Львівський державний університет безпеки життєдіяльності

Кафедра інформаційних технологій
та систем електронних комунікацій

Лекція

з дисципліни: «Клієнт-серверне програмування»

на тему: «*Bash Scripting: сценарій розгортання bash*»

ПЛАН ЛЕКЦІЇ

1. Командні оболонки ОС Unix.
2. Інтерпретатор bash.
3. Змінні в сценаріях для bash.
4. Програмування арифметичних виразів.
5. Умовні оператори.
6. Оператори порівняння
7. Оператори введення і виведення
8. Цикли та функції.
9. Об'єднання команд.

1. Командні оболонки ОС Unix.

Досить часто в процесі роботи з комп'ютером потрібно часто повторювати одні і ті ж команди Linux. Операційна система дозволяє записати необхідну послідовність команд в спеціальний файл, який називається сценарієм оболонки. Далі цей сценарій можна виконувати подібно виконанню звичайної команди Linux, набравши ім'я файла. Такий принцип організації файлів існує в MS DOS – це так звані командні файли.

Створювати сценарії надають спеціальні програми – оболонки, які виступають посередником між користувачем та операційною системою. Існують як текстові, так і графічні оболонки.

У системах Unix використовуються різні командні оболонки (command shells), які називаються також командними процесорами або інтерпретаторами команд. Командна оболонка забезпечує взаємодію між користувачем та ядром системи.

Серед них найбільш відомі і поширені:

• sh (Bourne shell) - оболонка Борна (не дуже зручна в роботі);

• csh (C-shell) - оболонка С (зручніша у порівнянні з sh, але несумісна з нею по командній мові);

• ksh (Korn shell) - оболонка Корна (містить потужну командну мову, засновану на мові sh, та розвинені засоби інтерактивної роботи);

• bash (Bourne-Again Shell) - оболонка «Борна» (зручна для інтерактивної роботи, створена на основі sh і багато в чому з нею сумісна).

Тип оболонки, як правило, можна визначити за останнім символом запрошення:

- знак долара «$» вказує на sh-сумісну оболонку (sh, bash, ksh),

- знак амперсанда «&» відповідає оболонці csh.

Однак у привілейованого користувача незалежно від командного процесора, який використовується, останнім символом запрошення зазвичай буває знак решітки «#».

Основними функціями командних оболонок є:

• організація діалогу з користувачем (введення команд);

• виконання внутрішніх команд;

• запуск зовнішніх програм;

• виконання командних файлів.

Командні мови в різних оболонках розрізняються, а стандартною прийнято вважати командну мову оболонки bash.

При вході в командний інтерпретатор відкривається вікно, в якому відображається: root – ім'я облікового запису користувача, ksp – ім'я комп'ютера, «~» - каталог виконання команди (домашній каталог).

Команди Т, як правило, складаються з назви програми, ключа і значення. В загальному вигляді виглядають так:

назва_ програми [-ключ] [значення].

назва_програми - це ім'я виконуваного файлу з каталогів, записаних у змінну $PATH (/bin, /sbin, /usr/bin, /usr/sbin, /usr/local/bin, /usr/local/sbin та ін.);

[ключ] - опції програми, які може приймати виконувана програма;

[значення] - даний параметр може приймати в якості аргументу цифри, текст, спеціальні символи і навіть змінні.

2. *Інтерпретатор bash.*

Весь цей час ми працювали у системі через командний інтерпретатор bash. Або іншими словами, ми взаємодіяли із системою через інтерпретатор bash. Це так званий інтерактивний режим роботи із системою. Крім цього можна взаємодіяти із системою за допомогою скриптів. Скрипт — це текстовий файл, у якому послідовно виконуються якісь дії (програми, команди, інші скрипти). Якщо необхідно щодня виконувати якісь дії (наприклад архівацію даних та копіювання їх на резервний сервер), то зручно записати такі дії у вигляді скрипта і потім викликати лише ім'я скрипта (або автоматизувати роботу за допомогою демона cron).

Наприклад, необхідно виконати таку послідовність команд:

mkdir dir1

cp file1.txt /home/user/dir1

cd dir1

Для того, щоб оформити вказані команди у вигляді сценарію, необхідно спочатку вказати назву оболонки, в рамках якої вони будуть виконані. Якщо у сценарії використовуються лише команди операційної системи, як у нашому випадку, вибір оболонки не є суттєвим. Тому виберемо найбільш поширену оболонку, яка практично завжди є в більшості дистрибутивів Linux – оболонку bash. Таким чином, першим рядком сценарію має бути такий запис:

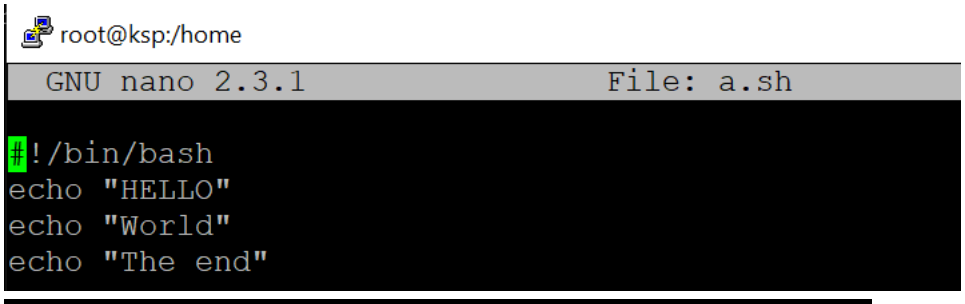
#!/bin/bash

Цей рядок є по суті ознакою того, що даний файл відноситься до сценаріїв. А далі вже записуються команди, які мають бути виконані.

Як bash розуміє, що перед ним скрипт, а не простий текстовий файл? Якщо пам'ятаєте, то в Linux немає поняття розширення файлу, і хоча для зручності деякі називають файли скриптів у вигляді *. sh, для інтерпретатора це не має жодного значення. Основний покажчик знаходиться усередині файлу. Коли ви запускаєте програму, виконуєте команду чи скрипт, bash починає аналізувати перші символи файлу у першому рядку. Якщо перші символи у файлі будуть #!, Bash робить наступне. Бере решту символів після #! і до кінця рядка (символу переходу на новий рядок), через пропуск додає ім'я останньої команди (тобто ім'я скрипта) і виконує отриману команду.

Найчастіше після #! вказується повний шлях до інтерпретатора для якого написано скрипт. Наприклад, якщо текст, що міститься в скрипті, написаний мовою python, то перший рядок буде виглядати як #!/bin/ python.

Після створення файла сценарію (наприклад, під іменем a.sh) необхідно перевірити, чи надано право виконати цей сценарій даному користувачеві.




```
-rw-r--r--. 1 root  root  53 Feb 22 10:04 a.sh
```

Як правило, для того, щоб перетворити будь-який власний файл у виконуваний файл, рядовий користувач повинен надати собі таке право за допомогою команди chmod:

chmod u+x a.sh

```
-rwxr-xr-x. 1 root  root  53 Feb 22 10:04 a.sh
```

Далі сценарій можна запустити на виконання із командного рядка:

./a.sh

```
[root@ksp home]# ./a.sh
HELLO
World
The end
```

або

bash a.sh

1. *Змінні в сценаріях для bash.*

Мова програмування оболонки підтримує три основних типи змінних:

- змінні середовища;

- вбудовані змінні;

- змінні користувача.

Змінні середовища надаються системою, їх не потрібно визначати, але до них можна звертатись. Вони часто використовуються для відстеження поточних даних (наприклад, поточного робочого каталогу). Зазвичай такі змінні позначаються за допомогою великих літер. Це допомагає користувачам розрізняти змінні середовища в інших контекстах. Значення деяких із них, наприклад PATH, можна змінювати в програмі оболонки. Основні змінні середовища наведені в таблиці 1.

Таблиця 1. Змінні середовища

Змінна середовища	Значення
USER	поточний користувач
PWD	поточна директорія

HOSTNAME	ім’я хоста(комп’ютера)
PS1	це підказка, яку shell показує, коли виводить командний рядок
OLDPWD	попередня робоча директорія. Використовується оболонкою для того, щоб повернутися в попередній каталог при виконанні команди cd -
HOME	домашня директорія поточного користувача
SHELL	шлях до оболонки поточного користувача (наприклад, bash або zsh).
EDITOR	заданий за замовчуванням редактор. Цей редактор викликається у відповідь на команду edit.
LOGNAME	ім’я користувача, яке використовується для входу в систему
PATH	шляхи до каталогів, у яких здійснюватиметься пошук викликаних команд. При виконанні команди система проходитиме по даним каталогам у вказаному порядку і вибере перший з них, в якому знаходитиметься виконуваний файл шуканої команди.
LANG	поточні налаштування мови та кодувань
TERM	тип поточного емулятора терміналу
MAIL	місце зберігання пошти поточного користувача
LS_COLORS	задає кольори, що використовуються для виділення об’єктів (наприклад, різні типи файлів у виводі команди ls будуть виділені різними кольорами).

Якщо ви виконаєте команду printenv або env без будь-яких аргументів, то вони покажуть список всіх змінних середовища:

$ printenv

Команди printenv та env виводять тільки змінні середовища. Якщо ви хочете отримати список всіх змінних, включаючи змінні (та функції) оболонки, то можете використати команду set:

$ set

Щоб знайти всі змінні, що містять вказаний рядок, використовуйте команду grep:

$ printenv | grep [ІМ'Я_ЗМІННОЇ]

Вбудовані змінні також надаються системою, але їх не можна змінювати. До цих змінних відносяться, зокрема, такі:

Parameter	Description
$0	Назва поточного сценарію оболонки
$1-$9	Позиційні параметри з 1 по 9
${10}	Позиційний параметр 10
$#	Кількість позиційних параметрів
$*	Усі позиційні параметри, «$*» — це один рядок
$@	Усі позиційні параметри, “$@” – це набір рядків
$?	Повернути статус останньої виконаної команди
$$	Ідентифікатор поточного процесу

Змінні користувача визначаються самим користувачем під час написання сценарію оболонки. В своєму власному сценарії користувач може довільно використовувати та змінювати їх.

Ім'я змінної bash має обов'язково починатися з літери. Далі можна використати і цифри. У bash змінні не мають типу, тому оголошувати змінну попередньо не потрібно, а відразу можна надавати значення:

```
#!/bin/bash

# Змінна

S1=Hello!

S2=”Hello World!”
```

Система самостійно “здогадується” про тип змінної за тим значенням, яке їй присвоюється. Це легко зробити, оскільки в сценарії можна оперувати тільки цілими числами або текстовими рядками. В такому випадку одна і та

ж змінна в один момент часу може слугувати для збереження цілого числа, а через деякий час – для збереження рядка. Однак це не рекомендується робити.

Значок # ("шарп") - є початком коментаря. Рядки, що починаються з символу #, інтерпретатор пропускатиме при обробці скрипту:

Для присвоєння змінній користувача цілого значення або одного текстового слова досить записати оператор присвоєння в загальноприйнятій формі, наприклад

n=5

mas=0

str1=process

Характерною особливістю сценаріїв є те, що для доступу до значення змінної, перед її іменем ставиться знак долара ($). Наприклад, для того, щоб змінній m присвоїти значення змінної n необхідно записати

m=$n

Також зверніть увагу, що навколо знака = не повинно бути пробілів.

Щоб отримати доступ до вмісту змінної, перед іменем змінної ставлять символ $:

echo $S1

echo $S2

Якщо текстовий рядок складається з кількох слів, тобто містить пропуски, тоді використовуються лапки, наприклад

str2=’long text string’

Для присвоєння текстових рядків використовуються також і подвійні лапки. В цьому випадку забезпечується підстановка значень змінних всередині рядків. Наприклад, якщо записати

str2=”long text string”

str3=”Value of str2 is $str2”

тоді значенням змінної str3 буде

Value of str2 is long text string

Різниця між одинарними і парними лапками у shell: в середині парних лапок діє підстановка значень змінних.

Змінні можуть бути локальними та глобальними. Локальні описуються словом local. Розглянемо наступний приклад:

```
#!/bin/bash
STR=Hello
function echoworld {
local STR=World
echo $STR
}
echo $STR
echo world
echo $STR
```

Результат виконання команди буде наступним:

```
Hello
World
Hello
```

Локальні - це змінні, які визначені лише для поточної сесії. Вони будуть зразу стерті після завершення сесії, будь то віддалений доступ або емулятор терміналу. Вони не зберігаються ні в яких файлах, а створюються і видаляються за допомогою спеціальних команд.

2. *Програмування арифметичних виразів.*

Для програмування арифметичних виразів можна застосувати такі знаки операцій:

+ сума,
- різниця,
* множення,
/ ділення,
% ділення з остачею.

Арифметичні вирази можна записати двома способами:

а) з використанням оператора let;
б) з використанням оператора expr.

Перший спосіб найбільш простий і зрозумілий, наприклад:

```
let y=$a*$b-$c
```

Арифметичний вираз може бути обчислений як з використанням квадратних дужок $[вираз] або з використанням подвійних круглих дужок $((вираз))

ari.sh
```
#!/bin/bash
let X=10+2*4
echo $X
echo "$((10+40))"
val=$[20+40]
echo "$[10*$val]"
```

Оператор expr розглядає свої аргументи як арифметичний або логічний вираз. В такому випадку потрібно враховувати деякі додаткові особливості такого запису, наприклад, символи арифметичних операцій відділяти від операндів пропуском, символ операції множення а також і весь вираз брати в лапки:

y=’expr $a ‘*’ $b ‘+’ $c’

3. *Умовні оператори*.

Конструкція if

Для перевірки умов у скрипті призначена конструкція if. Загальний синтаксис конструкції if у спрощеному вигляді, наступний:

if <вираз>
then <оператори 1>
else <оператори 2>
fi

Умовні оператори можуть бути вкладеними, наприклад:

if <вираз 1>
then <оператори 1>
else if <вираз 2>
then <оператори 2>
else <оператори 3 >
fi
fi

Ключове слово fi означає закінчення одного умовного оператора, тому їх кількість у вкладеному умовному операторі повинна дорівнювати кількості ключових слів if. Існує також форма скороченого запису вкладеного умовного оператора, коли достатньо лише одного ключового слова fi:

if <вираз 1>
then <оператори 1>
elif <вираз 2>
then <оператори 2>
else <оператори 3>
fi

Ключові елементи конструкції це if, then, else, fi. Крапка з комою потрібна тільки в тому випадку, якщо на одному рядку розташовано більше одного ключового елемента конструкції if. Якщо кожен ключовий елемент буде розташований на новому рядку, то крапка з комою не потрібна.

Розглянемо наступний практичний приклад:

```
#!/bin/bash
STR="Hello world"
if [ "$STR"  =  "Hello world" ]
then
echo YES
else
echo NO
fi
```

Конструкцію if з прикладу можна написати в командному рядку та виконати в інтерактивному режимі. Виглядатиме рядок ось так:

STR=”Hello world”; if [«$STR» = «Hello world»]; then echo YES; else echo NO; fi

Так само як і в скрипті, але є точки з комою, які відокремлюють команди і ключові елементи. Ще раз зверніть увагу на прогалини у [ ]. Щоб не забувати про це, пам'ятайте, що [це просто команда (на зразок будь-якої іншої команди linux), а після команди завжди йде пробіл і далі ключі або параметри.

6. *Оператори порівняння.*

Оператори порівняння в bash бувають арифметичними (для порівняння чисел) та операторами порівняння рядків.

Порівняння чисел

Для порівняння двох чисел можуть використовуватись такі операції:

a –eq b визначення рівності чисел a і b;

a –ne b визначення нерівності чисел a і b;

a –gt b визначення того, чи число a більше числа b ;

a –ge b визначення того, чи число a більше або дорівнює числу b;

a –lt b визначення того, чи число a менше числа b;

a –le b визначення того, чи число a менше або дорівнює числу b.

```
[user2@ksp ~]$ STR1=8; STR2=9; [ "$STR1" -gt "$STR2" ]; echo $?
1
[user2@ksp ~]$ STR1=8; STR2=9; [ "$STR1" -lt "$STR2" ]; echo $?
0
```

Порівняння рядків

Для порівняння двох рядків можуть використовуватись такі операції:

str1 = str2 визначення рівності рядків str1 і str2;

str1 != str2 визначення нерівності рядків str1 і str2;

-n перевірка ненульової довжини рядка;

-z перевірка нульової довжини рядка.

Практичний приклад. Наберіть і виконайте в консолі наступний рядок (не забувайте про прогалини!):

STR1=aaa; STR2=abc; [«$STR1» = «$STR2»]; echo $?

В результаті ви отримаєте число 1, яке виведе команда echo$? Конструкція $? містить числовий результат виконання попередньої команди. Попередня команда була [«$STR1» = «$STR2»] у разі виконання умови команда повернула б 0, але оскільки умова не виконується, результат виконання команди відмінний від нуля.

```
[user2@ksp ~]$ STR1=aaa; STR2=aaa; [ "$STR1" = "$STR2" ]; echo $?
0
```

```
[user2@ksp ~]$ STR1=aaa; [ -n "$STR1" ]; echo $?
0
```

Логічне порівняння

Логічні операції використовуються для порівняння виразів логічних операцій NOT, AND і OR:

! логічна операція NOT над логічним виразом;

-a логічна операція AND над двома логічними виразами;

-o логічна операція OR над двома логічними виразами.

Файлові операції порівняння

Такі операції можуть використовуватись для перевірки файлів:

-d перевірка того, чи є файл каталогом;

-f перевірка того, чи є файл звичайним файлом;

-r перевірка того, чи є право доступу для читання файла;

-w перевірка того, чи є право доступу для запису у файл;

-x перевірка того, чи є право доступу для виконання файла;

-s перевірка того, чи є файл з ненульовою довжиною.

```
[user2@ksp home]$ [ -d a1 ]; echo $?
1
[user2@ksp home]$ [ -f a1 ]; echo $?
0
```

7. *Оператори введення і виведення*

Для деяких видів скриптів (наприклад інсталяційних), необхідно, щоб користувач вводив якісь значення, і вони могли бути використані далі в скрипті. Для вирішення цього завдання в bash існує команда read. У найпростішому випадку команда записується так: read P1 P2 PN, де P1 … PN – це імена параметрів через пропуск. Команда read зчитує дані із потоку введення та записує їх у зазначені змінні. Як роздільник для даних команда read використовує значення яке зберігається системною зміною bash - IFS. Як правило, це пробіл або табуляція. Розглянемо приклад скрипту name.sh:

#!/bin/bash

echo «Введіть ваші ім’я та прізвище через пробіл і натисніть Enter:»

read P1 P2
echo «Ваше ім’я: $P1»
echo «Ваше прізвище: $P2»

```
 GNU nano 2.3.1                File: name.sh

#!/bin/bash
echo "Введіть ваше ім'я та прізвище і натисніть Enter:"
read P1 P2
echo "Ваше ім'я: $P1"
echo "Ваше прізвище: $P2"
```

І результат виконання:

```
[root@ksp home]# ./name.sh
Введіть ваше ім'я та прізвище і натисніть Enter:
Микола Гнатюк
Ваше ім'я: Микола
Ваше прізвище: Гнатюк
```

Для виведення повідомлень на екран дисплея використовується оператор echo.

Наприклад,

echo This is message

Якщо в сценарії необхідно вивести значення змінної, тоді використовуються подвійні лапки, наприклад

echo “result is $y”

Користуючись операторами введення та виведення можна написати найпростіший сценарій для обчислення арифметичних виразів:

```
#!/bin/bash
a=3
b=5
echo “Введіть значення змінної x”
read x
let y=($a+$b)*$x
echo “result is $y”
```

8. *Цикли та функції.*

Ми розглядаємо ті елементи bash, які допоможуть нам розуміти скрипти операційної системи. Такими елементами безумовно є цикли та функції.

Цикл for

Цикл for у bash має два види. Розглянемо спочатку класичний варіант for. Загальний вигляд наступний:

for змінна *in* послідовність значень
do
команди
done

Між елементами for i in визначається змінна, яка по черзі приймає значення з послідовності значень заданої між in і do. Між do і done знаходяться команди, які виконуються кожного разу, коли змінна змінює своє значення. Цикл припиняє роботу, коли змінна прийме останнє значення з послідовності. Значення в послідовності задаються через пропуск.

Практичний приклад:

```
#!/bin/bash
for i in 1 2 3 a b c
do
echo i=$i
done
```

Якщо запустити такий скрипт на виконання, отримаємо наступний результат:

```
i=1
i=2
i=3
i=a
i=b
i=c
```

Приклад сценарію для знаходження суми елементів одновимірного масиву:

```
#!/bin/bash
mas=’3 7 12 5 8’
sum=0
for var in $mas
do
let sum=$sum + $var
done
echo “result is $sum”
```

Повернемося до другого виду for. Часто в скриптах можна зустріти так званий С-подібний варіант for, який використовується для циклів на основі чисел. Розглянемо відразу приклад:

```
#!/bin/bash
for ((i=1; i<6; i++))
do
echo i=$i
done
```

Приклад сценарію з використанням циклу for для знаходження максимального значення серед елементів одновимірного масиву:

```
#!/bin/bash
mas[0]=3
mas[1]=7
mas[2]=12
mas[3]=5
mas[4]=8
```

```
max=mas[0]
for((i=0; i<5; i++))
do
if [ $max –lt ${mas[i]} ]
then let max=${mas[i]}
fi
done
echo “result is $max”
```

Цикл while

Загальний вигляд:

while вираз
do
команди
done

Цикл виконується поки умова, що перевіряється у виразі, вірна. Як тільки вираз повертає хибно, виконання циклу припиняється.

Практичний приклад:

```
#!/bin/bash
i=1
while [ $i -lt 7 ]
do
echo $i
let i=i+1 #або можна написати i++
done
```

Цикл until

Схожий на while з тією лише різницею, що в ньому команди всередині циклу виконуються тоді, коли умова не виконується. Синтаксис такий самий тільки замість while використовується until.

```
until <вираз>
do
<оператори>
done
```

Приклад:

```
#!/bin/bash
i=1
until [ $i -gt 6 ]
do
echo «until $i»
i=$[ $i+1 ]  #або можна написати let i=i+1
done
```

Функції

Як і в мовах високого рівня, окремі частини сценаріїв можна записувати у вигляді функцій. Формат визначення функції такий:

```
func() {
<оператори>
}
```

Виклик функції, якій передаються параметри param1, param2, param3 :

```
func param1 param2 param3
```

Можна також передати параметри у вигляді одного рядка, наприклад, $@. Функція може інтерпретувати параметри за тими же принципами, за якими виконується інтерпретація позиційних параметрів, що передаються сценарію оболонки.

Наприклад, для обчислення виразу

y=(a+b)/c та y=(a+b)*d

можна використати дві функції:

```
#!/bin/bash
a=9; b=5; c=7; d=2
calc1() {
let y=($a+$b)/$c; echo "Result is $y"
}
calc2() {
let y=($a+$b)*$d; echo "Result is $y"
}
echo "input x"
read x
if [ $x -eq 5 ]
then calc1
else calc2
fi
```

9. *Об'єднання команд*

Розглянемо об'єднання команд у bash, про конструкцію case та деякі інші команди.

case

case зручніше використовувати для розгалуження, коли значення необхідно перевіряти на точну відповідність і може приймати три і більше значень. Замість case можна було б використовувати if, але такі конструкції виглядають громіздко і їх не так зручно читати в скриптах.

Загальний синтаксис команди case наступний:

```
case var in
S1) <оператори 1>;;
S2) <оператори 2>;;
S3) < оператори 3>;;
*) <оператори 4>;;
esac
```

В залежності від того, чи збігається значення змінної var із значенням S1, S2 або S3, виконуються відповідно <оператори 1>, <оператори 2> або <оператори 3>. Якщо вказаного збігу немає, тоді виконуються <оператори 4>.

Об'єднання команд

Якщо в одному рядку скрипту ми пишемо два ключові слова якоїсь конструкції, то їх потрібно розділяти символом ;. Крапка з комою і є найпростішим способом об'єднання команд (ключові слова теж сприймаються bash як команди). Команди записані таким чином виконуватимуться послідовно незалежно від результату виконання попередньої команди. Приклади нижче можна набирати як у командному рядку, так і в скрипті.

a=127; echo $a; b=172; echo $b; let c=$a+$b; echo $c

127

172

299

Як бачите з прикладу, на одному рядку розміщено 6 команд, які були виконані послідовно.

Для паралельного запуску команд використовують вже знайомий вам амперсанд. Такі команди будуть запущені у фоновому режимі та виконуватимуться паралельно.

sleep 10& sleep 10& echo Hello!

```
[root@ksp home]# sleep 10& sleep 10& echo Hello!
[1] 1437
[2] 1438
Hello!
```

Після виконання будь-яка команда повертає числовий код результату виконання команди. Якщо це 0 - значить команда виконана успішно, якщо це число відмінне від нуля - значить, команда завершилася з помилкою. Можна будувати виконання послідовності команд з огляду на цей момент. І тому існують такі елементи: подвійний амперсанд — && і подвійний “пайп” — ||. Якщо об'єднання команд відбувається лише через &&, кожна наступна команда буде виконуватися лише у разі успішного завершення попередньої команди. Як тільки якась команда ланцюжка поверне код повернення відмінний від нуля, наступні після нього команди не будуть виконані. Наприклад скористаємося командами true (завжди повертає 0) і false (завжди повертає 1).

true && echo «1» && false && echo «2»

```
[root@ksp home]# true && echo "1" && false && echo "2"
1
[root@ksp home]# true && echo "1" && true && echo "2"
1
2
```

У прикладі команда true повернула 0 і наступна команда echo “1″ була виконана і теж повернула 0, після чого була виконана команда false, яка

повернула 1 та виконання команд припинилося (команда echo 2 не виконалася).

Команда || працює з точністю до навпаки. Тобто наступна команда буде виконана тільки в тому випадку, якщо попередня команда повернула код повернення відмінний від нуля (правило діє в такому вигляді, якщо всі команди об'єднані тільки ||). Якщо команда повернула 0, виконання подальшого ланцюжка команд переривається. Якщо у прикладі вище замінити && на ||, то виконається лише команда true, яка поверне 0 і подальші команди не будуть виконані.

```
[root@ksp home]# true || echo "1" || false || echo "2"
[root@ksp home]#
```

Якщо замінити true на false, то отримаємо такий самий результат:

false || echo «1» || false || echo «2»

```
[root@ksp home]# false || echo "1" || false || echo "2"
1
```

Розглянемо приклад об'єднання команд і результат їх виконання:

false || echo «1» || false && echo «2»

```
[root@ksp home]# false || echo "1" || false && echo "2"
1
2
```

У даному випадку команда echo "2" виконується. Вся справа в тому, що насправді відбувається перевірка всіх операторів об'єднання, які беруть участь у ланцюжку. Якщо команда повернула 1, а всі подальші оператори об'єднання && — то й не буде виконана жодна команда. Але якщо в ланцюжку зустрінеться оператор ||, то команда після нього буде виконана і перевірка йтиме далі.

Розглянемо докладніше цей ланцюжок:

1) Виконується команда false та повертає код виконання 1

2) Далі стоїть оператор ||, отже наступна команда виконуватиметься

3) Виконується команда echo "1" та повертає код 0

4) Далі стоїть оператор ||, отже наступна команда не виконується

5) Перевіряється наступний оператор – це оператор &&, отже наступна команда (echo “2″) буде виконана.

Подивіться ще кілька прикладів:

true || echo «1» && false || echo «2»

```
[root@ksp home]# true || echo "1" && false || echo "2"
2
```

true && echo «1» || false && echo «2»

```
[root@ksp home]# true && echo "1" || false && echo "2"
1
2
```

true || echo «1» || echo «2» || echo «3» || false && echo «4» && echo «5» || echo «6»

```
[root@ksp home]# true || echo "1" || echo "2" || echo "3" ||
 false && echo "4" && echo "5" || echo "6"
4
5
```

Команди можна об'єднувати в блоки за допомогою круглих дужок:

true || echo «1» && (echo «2»; echo «3» || echo «4») && echo «5» || echo «6»

```
[root@ksp home]# true || echo "1" && (echo "2"; echo "3" ||
 echo "4") && echo "5" || echo "6"
2
3
5
```

або фігурних:

true || echo «1» && { echo «2»; echo «3» || echo «4»; } && echo «5» || echo «6»

```
[root@ksp home]# true || echo "1" && { echo "2"; echo "3" ||
  echo "4"; } && echo "5" || echo "6"
2
3
5
```

У другому випадку фігурні дужки обов'язково мають бути між пробілами, і після останньої команди має йти крапка з комою.

Відмінності між круглими та фігурними дужками в тому, що команди у фігурних дужках виконуються в рамках того самого процесу, що і скрипт, а команди у круглих дужках виконуються як би у своєму підпроцесі.

На завершення лекції ще кілька команд.

Якщо зі скрипту необхідно запустити виконуваний файл, достатньо написати ім'я файлу (якщо він розташований у каталозі описаному в змінній PATH) або повний шлях до файлу, якщо він розташований у специфічному каталозі:

/home/dir2/a.sh

У цьому випадку буде запущено файл a.sh, а виконання поточного скрипту буде призупинено та відновлено після завершення роботи викликаного скрипту. Якщо потрібно, щоб при виклику нового скрипта або програми поточний скрипт завершував роботу, викликати його потрібно за допомогою команди exec.

1
2
3 …
exec /home/dir2/a.sh
echo «1»
echo «2»

При такому варіанті запуску команди echo 1 і echo 2 не будуть виконані, оскільки після exec /home/dir2/a.sh скрипт завершить свою роботу.

Також роботу скрипту можна будь-де перервати командою exit. Бажано вказати код результату виконання скрипта. Наприклад, exit 0.

Питання для самоконтролю

1. Яка різниця між компіляторами та інтерпретаторами? До якого типу відноситься мова оболонки bash?
2. Що ви знаєте про змінні в оболонці bash?
3. Що таке командна оболонка?
4. Які функції виконує командна оболонка?
5. Що таке середовище оточення?
6. Що таке змінні оточення?
7. Які типи змінних оточення вам відомі?
8. Які основні змінні середовища?
9. Які основні змінні оболонки?
10. Які Ви знаєте оператори введення-виведення?
11. Як відбувається порівняння чисел, рядків та файлів в сценаріях?
12. Які умовні оператори та оператори циклів Ви знаєте?

Лекцію склав доцент кафедри ІТтаСЕК Борзов Ю.О.