

Лабораторная работа #2.2

Командный язык и скрипты Shell

Часть 2 – Скрипты, создание проекта, диалоги



РАЗРАБОТАЛ

СЕРГЕЙ СТАНКЕВИЧ (SERHEY STANKEWICH, MINSK, BELARUS)

ЛАБОРАТОРНАЯ РАБОТА #2.2

Командный язык и скрипты SHELL

Часть 2 – Скрипты, создание проекта, диалоги

Цель работы

Закрепить на практике принципы создания проектов с помощью скриптов SHELL, освоить средства примитивного графического интерфейса в Linux-скриптах.

Краткие теоретические сведения.

Создание программы с помощью скриптов SHELL

Приступим к созданию программы. Цель данного проекта – показать, как можно использовать разные возможности командной оболочки для создания полноценных программ.

Мы напишем генератор отчетов. Он будет выводить разнообразную информацию о системе и ее состоянии в формате HTML, благодаря чему ее можно будет просматривать в веб-браузере. Создание программ выполняется в несколько этапов, на каждом из которых добавляются новые функции и возможности.

Начальный этап создания программы представлен в разделе упражнений «Упражнение 2». По окончании первого этапа наша программа будет воспроизводить минимальную HTML-страницу.

Затем используем различные возможности командного интерпретатора и скриптов SHELL для создания полноценной функциональной программы.

Использование переменных

Переменные позволяют хранить в файле сценария информацию, например – результаты работы команд для использования их другими командами.

Существуют два типа переменных, которые можно использовать в bash-скриптах:

- Переменные окружения
- Пользовательские переменные

Присваивание значений любым переменным производится так:

ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ

где *переменная* – это имя переменной, а *значение* – строка.

В отличие от некоторых других языков программирования, командная оболочка не заботится о типах значений, присваиваемых переменным, она все значения интерпретирует как строки.

Значение переменной можно получить, используя знак доллара (\$).

Название *переменная* подразумевает значение, которое может изменяться, в процессе работы приложения. Переменные, значение которых не изменяется, называются *константами*. Командная оболочка не различает константы и переменные, эти термины используются в основном для удобства программиста. *Типичное соглашение* – использовать буквы верхнего регистра для обозначения констант и буквы нижнего регистра для истинных переменных.

grade=5
PERSON="Adam"

Переменные окружения

Окружение (environment) – это набор переменных и их значений, с помощью которых можно передавать в программу какие-нибудь данные общего назначения. Окружение, это набор специфичных для конкретного пользователя пар ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ.

Увидеть, что хранится в окружении, можно при помощи встроенной в bash команды `set` или программы `printenv`. Команда `set` выводит переменные обоих видов – командной оболочки и окружения, тогда как `printenv` выводит только окружение.

Иногда в командах оболочки нужно работать с некими системными данными. Вот, например, как вывести домашнюю директорию текущего пользователя:

echo "Home for the current user is: \$HOME"

Обратите внимание на то, что мы можем использовать системную переменную `$HOME` в двойных кавычках, это не мешает системе её распознать.

Перечень переменных окружения представлен в разделе «Дополнительная информация».

Пользовательские переменные

В дополнение к переменным среды, `bash`-скрипты позволяют задавать и использовать в сценарии собственные переменные. Подобные переменные хранят значение до тех пор, пока не закончится выполнение сценария.

```
grade=5
PERSON="Adam"
echo "$PERSON is a good boy, he is in grade $grade"
```

Подстановка команд

Одна из самых полезных возможностей `bash`-скриптов — это возможность извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария. Сделать это можно двумя способами:

- С помощью значка *обратного апострофа* «```»
- С помощью конструкции `$()`

Используя первый подход, проследите за тем, чтобы вместо *обратного апострофа* не ввести *одинокую кавычку*. Команду нужно заключить в два таких знака:

```
mydir=`pwd`
```

При втором подходе то же самое записывают так:

```
mydir=$(pwd)
```

Встроенные документы

Для вывода текста как правило применяется команда `echo`. Однако существует еще один, третий метод, который называется *встроенным документом* (here document), или *встроенным сценарием* (here script). **Встроенный документ** — это дополнительная форма перенаправления ввода/вывода, которая передает

текст, встроенный в сценарий, на стандартный ввод команды. Действует это перенаправление так:

```
команда << индикатор
текст
индикатор
```

где *команда* — это имя команды, принимающей указанный текст через стандартный ввод, а *индикатор* — это строка, отмечающая конец встроенного текста.

Рассмотрим фрагмент скрипта:

```
...
cat << _EOF_
<HTML>
  <HEAD>
    <TITLE>$TITLE</TITLE>
  </HEAD>
  <BODY>
    <H1>$TITLE</H1>
    <P>$TIME_STAMP</P>
  </BODY>
</HTML>
_EOF_
```

Вместо команды `echo` в сценарии используются команда `cat` и встроенный документ. На роль индикатора была выбрана строка `_EOF_` (означает *end-of-file* — конец файла, распространенное соглашение), и она отмечает конец встроенного текста. Обратите внимание, что строка-индикатор должна находиться в отдельной строке, одна, и за ней не должно следовать никаких пробелов.

Но какие преимущества дало использование встроенного документа здесь? Практически никаких, кроме того, что кавычки внутри встроенных документов теряют свое специальное значение для командной оболочки.

Командная оболочка не обращает никакого внимания на кавычки. Она интерпретирует их как обычные символы. Благодаря этому мы свободно

вставляем кавычки во встроенные документы. Этим обстоятельством можно воспользоваться при разработке программ составления отчетов

Встроенные документы можно использовать с любыми командами, принимающими данные со стандартного ввода. В следующем примере встроенный документ используется для передачи последовательности команд программе **ftp**, чтобы загрузить файл с удаленного FTP-сервера:

```
#!/bin/bash
# Сценарий загрузки файла через FTP
FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz
ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l $REMOTE_FILE
```

Если заменить оператор перенаправления << на <<-, командная оболочка будет игнорировать начальные символы табуляции во встроенном документе:

```
#!/bin/bash
# Сценарий загрузки файла через FTP
FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz
ftp -n <<- _EOF_
    open $FTP_SERVER
    user anonymous me@linuxbox
cd $FTP_PATH
    hash
    get $REMOTE_FILE
    bye
_EOF_
ls -l $REMOTE_FILE
```

Благодаря этому во встроенный документ можно добавить отступы для удобного чтения.

Функции командной оболочки

Функции — это «мини-сценарии», находящиеся внутри другого сценария, которые работают как автономные программы. Функции имеют две синтаксические формы.

Первая выглядит так:

```
function имя {  
    команды  
    return  
}
```

где *имя* — это имя функции, а *команды* — последовательность команд внутри функции.

Вторая форма выглядит так:

```
имя () {  
    команды  
    return  
}
```

Область видимости переменных

Каждая переменная имеет определенную *область видимости* (scope). Область видимости представляет участок программы, в рамках которого можно использовать переменную.

Переменные бывают *глобальными* и *локальными*.

Глобальные переменные существуют и доступны в любой точке программы. В некоторых случаях это безусловно полезное свойство осложняет использование функций. Внутри функций иногда желательно использовать локальные переменные. **Локальные** переменные доступны только внутри функции, в которой они определены, и прекращают свое существование по окончании выполнения функции. Локальные переменные объявляются внутри функции при помощи ключевого слова **local** перед именем переменной.

Поддержка локальных переменных позволяет программисту использовать переменные с именами, которые уже определены в сценарии, глобально или в других функциях, не беспокоясь о возможных *конфликтах имен*.

Управление потоком выполнения посредством ветвления программы

Очень часто необходим некий способ «изменить направление» выполнения сценария, опираясь на результаты проверки. То есть нужен способ, обеспечивающий *ветвление* программы. Эта задача решается использованием управляющих конструкций: **if-then**, **if-then-else**, а также **case**.

Код завершения (exit status)

Команды (включая сценарии и функции, написанные нами) по завершении работы возвращают системе значение, которое называют **кодом завершения** (exit status). Это значение – целое число в диапазоне от 0 до 255 – сообщает об успешном или неуспешном окончании команды. По соглашениям значение 0 служит признаком успешного завершения, а любое другое – не успешного. Командная оболочка поддерживает переменную (**\$?**), посредством которой можно определить код завершения. Например:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

В первый раз команда `ls` выполнилась благополучно, переменная «**\$?**» равна 0 (0 всегда служит признаком успешного выполнения). Во второй раз команда `ls` сообщила об ошибке, а переменная «**\$?**» равна 2, указывающее, что команда столкнулась с ошибкой.

Одни команды используют разные коды завершения, чтобы сообщить о характере ошибки, другие, столкнувшись с любой ошибкой, просто возвращают значение 1. Страницы справочного руководства `man` часто включают раздел с заголовком «Exit Status» («Коды завершения»), с описанием кодов ошибок.

Команда test

Чаще всего с инструкцией `if` используется команда **test**. Команда `test` может выполнять различные проверки и сравнения. Она имеет две эквивалентные формы:

test выражение

и более популярную

[выражение]

где выражение возвращает истинное (true) или ложное (false) значение. Команда `test` возвращает код завершения 0, если выражение истинно, и код завершения 1, если выражение ложно.

В команду `test` можно вставлять выражения для проверки: сравнения числовых значений; сравнения строковых значений; для проверки файлов.

Выражения команды `test` для проверки файлов

`-d file` Проверяет, существует ли файл, и является ли он директорией.
`-e file` Проверяет, существует ли файл.
`-f file` Проверяет, существует ли файл, и является ли он файлом.
`-r file` Проверяет, существует ли файл, и доступен ли он для чтения.
`-s file` Проверяет, существует ли файл, и не является ли он пустым.
`-w file` Проверяет, существует ли файл, и доступен ли он для записи.
`-x file` Проверяет, существует ли файл, и является ли он исполняемым.
`file1 -nt file2` Проверяет, новее ли `file1`, чем `file2`.
`file1 -ot file2` Проверяет, старше ли `file1`, чем `file2`.
`-O file` Проверяет, существует ли файл, и является ли его владельцем текущий пользователь.
`-G file` Проверяет, существует ли файл, и соответствует ли его идентификатор группы идентификатору группы текущего пользователя.

Выражения команды `test` для проверки строк

`str1 = str2` Проверяет строки на равенство, возвращает истину, если строки идентичны.
`str1 != str2` Возвращает истину, если строки не идентичны.
`str1 < str2` Возвращает истину, если `str1` меньше, чем `str2`.
`str1 > str2` Возвращает истину, если `str1` больше, чем `str2`.
`-n str1` Возвращает истину, если длина `str1` больше нуля.
`-z str1` Возвращает истину, если длина `str1` равна нулю.

Выражения команды `test` для проверки чисел

`n1 -eq n2` Возвращает истинное значение, если `n1` равно `n2`.
`n1 -ge n2` Возвращает истинное значение, если `n1` больше или равно `n2`.
`n1 -gt n2` Возвращает истинное значение, если `n1` больше `n2`.
`n1 -le n2` Возвращает истинное значение, если `n1` меньше или равно `n2`.
`n1 -lt n2` Возвращает истинное значение, если `n1` меньше `n2`.
`n1 -ne n2` Возвращает истинное значение, если `n1` не равно `n2`.

Более современная версия команды `test` – составные команды

Составные команды, действуют как улучшенная замена для команды `test`. Она имеет следующий синтаксис:

[[выражение]] и ((арифметическое выражение или целое число))

где выражение возвращает истинное (true) или ложное (false) значение.

Команда `[[]]` очень похожа на команду `test` (она поддерживает те же выражения), но добавляет новое выражение для проверки строк:

строка1 =~ регулярное_выражение

возвращающее истинное значение, если **строка1** соответствует *расширенному* регулярному выражению. Это открывает широкие перспективы для проверки корректности данных.

Объединение выражений логическими операторами

Для более сложных вычислений существует возможность объединения выражений с помощью *логических операторов*. Команды `test` и `[[]]` поддерживают три логические операции. Это И (AND), ИЛИ (OR) и НЕ (NOT).

Логические операторы

Операция	test	[[]] и (())
И	-a	&&
ИЛИ	-o	
НЕ	!	!

Проверка логическими операторами:

```
if [[ "$INT" =~ ^-[0-9]+$ ]]; then
    if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then ...
```

Таже проверка с помощью `test`:

```
if [ $INT -ge $MIN_VAL -a $INT -le $MAX_VAL ]; then ...
```

Проверка логическими операторами:

```
if [[ "$INT" =~ ^-[0-9]+$ ]]; then
    if [[ ! (INT -ge MIN_VAL && INT -le MAX_VAL) ]]; then ...
```

Таже проверка с помощью `test`:

```
if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \) ]; then ...
```

Операторы управления – еще один способ ветвления

Bash поддерживает два оператора управления, которые используются для ветвления. Операторы `&&` (И) и `||` (ИЛИ) действуют подобно логическим операторам в составной команде `[[]]`. Они имеют следующий синтаксис:

команда1 && команда2

и

команда1 || команда2

Важно понимать, как они действуют. В последовательности с оператором `&&` первая команда выполняется всегда, а вторая – только если первая завершилась успехом.

В последовательности с оператором `||` первая команда выполняется всегда, а вторая – только если первая завершилась неудачей

Практические примеры:

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

Последовательность создаст каталог с именем `temp` и, если эта операция завершится успехом, каталог `temp` будет назначен текущим рабочим каталогом.

```
[me@linuxbox ~]$ [ -d temp ] || mkdir temp
```

Последовательность проверит существование каталога `temp`, и только если проверка не увенчается успехом, будет выполнена команда его создания

Такие конструкции очень удобно использовать для обработки ошибок в сценариях.

Чтение ввода с клавиатуры

Современным программам характеризуются – *интерактивностью*, то есть возможность взаимодействия с пользователем.

Встроенная команда **read** используется для чтения единственной строки со стандартного ввода. Эту команду можно использовать для чтения ввода с клавиатуры или, в случае перенаправления, строки данных из файла. Команда имеет следующий синтаксис:

read [-параметры] [переменная...]

где *параметры* – это один или несколько параметров, а *переменная* – имя одной или нескольких переменных для сохранения введенного значения. Если имя переменной не указано, строка с данными сохраняется в переменной `REPLY`.

Использование новой для нас возможности приема ввода с клавиатуры влечет за собой дополнительную проблему: необходимость **проверки введенных данных**. Зачастую неожиданности возникают в форме ввода ошибочных данных. Проверки должны выполняться для любых вводимых данных, чтобы обезопасить программу от недопустимых значений.

Правильные интерактивные сценарии предлагают пользователю ввести элемент данных и затем последовательно анализирует его содержимое. В таких сценариях используется множество идей: функции `[]`, `(())`, операторы управления `&&` и `if`, а также разумную дозу регулярных выражений `healthy`.

Для организации интерактивной работы используются *меню*. Программы, управляемые системой меню, выводят список возможных вариантов и предлагают пользователю выбрать один из них.

Однако, при организации меню программа может быть неудобна для пользователя. Например, программа выполняет только один выбранный вариант и завершается. Хуже того, в случае ошибочного выбора программа заканчивается с выводом сообщения об ошибке, не давая возможности повторить попытку. Пользоваться программой намного удобнее, если она снова и снова выводит меню и предлагает сделать выбор, пока пользователь не выберет пункт, соответствующий выходу из программы.

С помощью организации *циклов* можно реализовать многократное выполнение участков программ.

Циклы

Командная оболочка поддерживает три составные команды для организации циклов. Это команды: **while**, **until**, **for**.

Цикл **for** отличается от циклов **while** и **until** поддержкой средств обработки последовательностей. Это очень полезная возможность. Как следствие, цикл **for** пользуется большой популярностью среди создателей сценариев для `bash`.

В некоторые версии `bash` добавлена вторая форма синтаксиса команды **for**: форма в стиле языка C.

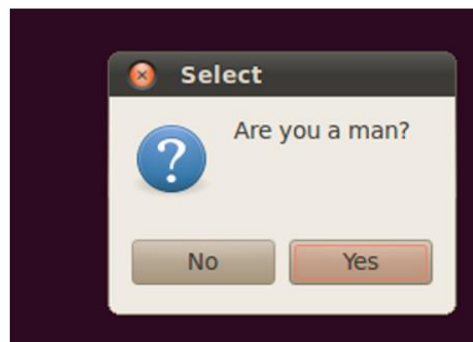
Простой графический интерфейс диалога пользователя с программой

Создание графического интерфейса диалога позволяет организовать общение скрипта с пользователем и придать тем самым ему вид настоящей программы. Диалоговое окно отображается командой **gdialog**, где прописываются

различные опции. Утилита **gdialog**, и другие подобные ей (zenity, Xdialog, whiptail), не являются предустановленными программами дистрибутива Linux, поэтому требуют дополнительной установки. Классической литературы и справочной документации по использованию подобных утилит нет, так как они являются кустарным продуктом свободных разработчиков.

```
#!/bin/sh
```

```
gdialog --title "Select" --yesno "Are you a man?" 9 18  
case "$?" in  
  0) gdialog --infobox "Good day, Boy" 5, 20;;  
  1) gdialog --infobox "Good day, Girl" 5, 20;;  
  *)gdialog --infobox "Good day" 5, 20;;  
  
esac  
sleep 1  
exit 0
```



На основе представленных выше инструментов командной оболочки мы на практике создадим небольшую программу.

Цель данного проекта – показать, как можно использовать разные возможности командной оболочки для создания программ и, что особенно важно, для создания *хороших* программ.

*Best of LUCK with it, and remember to HAVE FUN while you're learning :)
Sergey Stankewich*

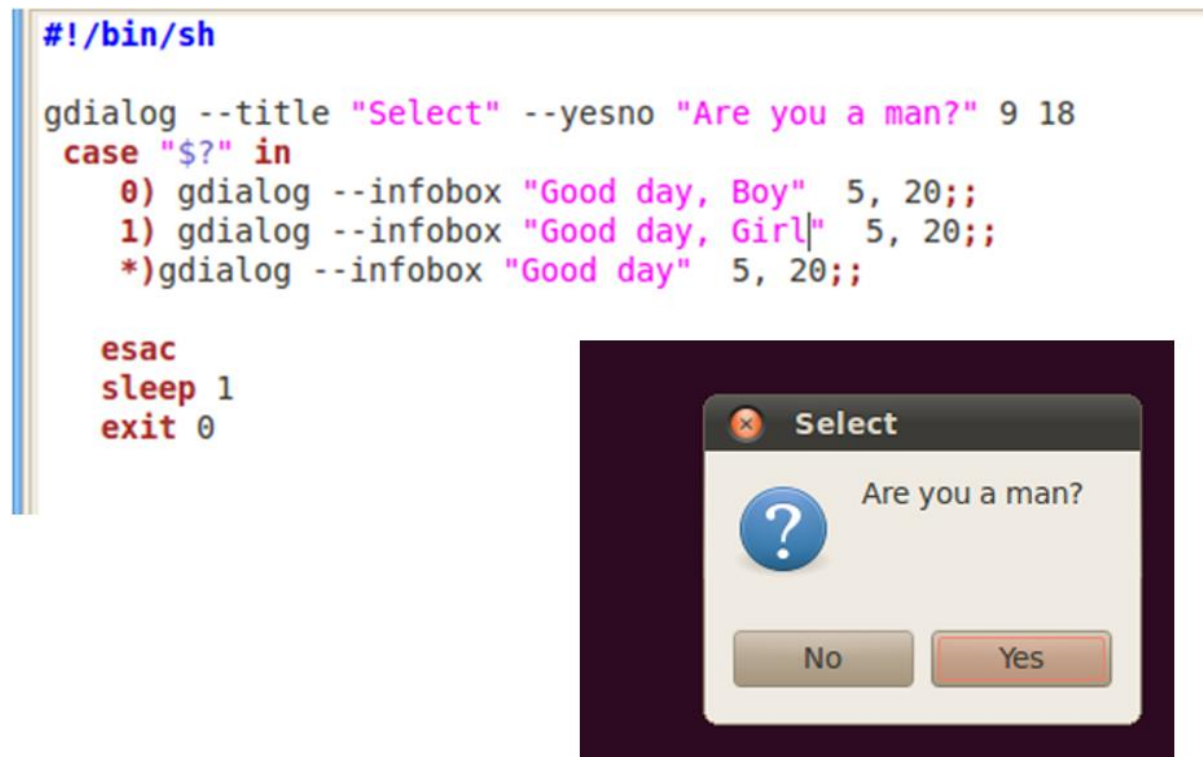


УПРАЖНЕНИЯ

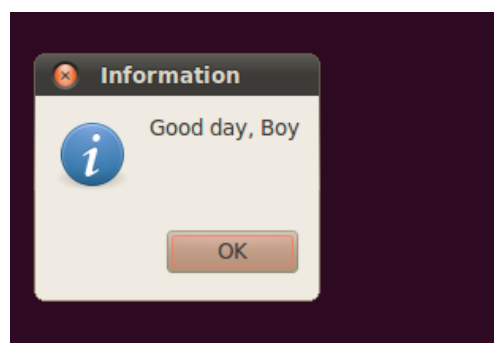
Упражнение 1

Создание простых графических диалогов

Напишем следующий скрипт. Пусть его имя – **script_dia1**. Выполним его. На экране мы увидим диалоговое окно:



Нажмем кнопку Yes:



Обратим внимание на обработку опции, указанной пользователем через case:

```
case "$?" in
    gdialog --infobox "Good Day, Boy" 5,20;;
    ...
```

В первом случае мы выводим меню типа yesno. Переменной с необычным именем \$? Присваивается значение нажатой клавиши (Yes – это 0, No – это 1).

Затем с помощью оператора **case** проверяем, что было нажато и реагируем.

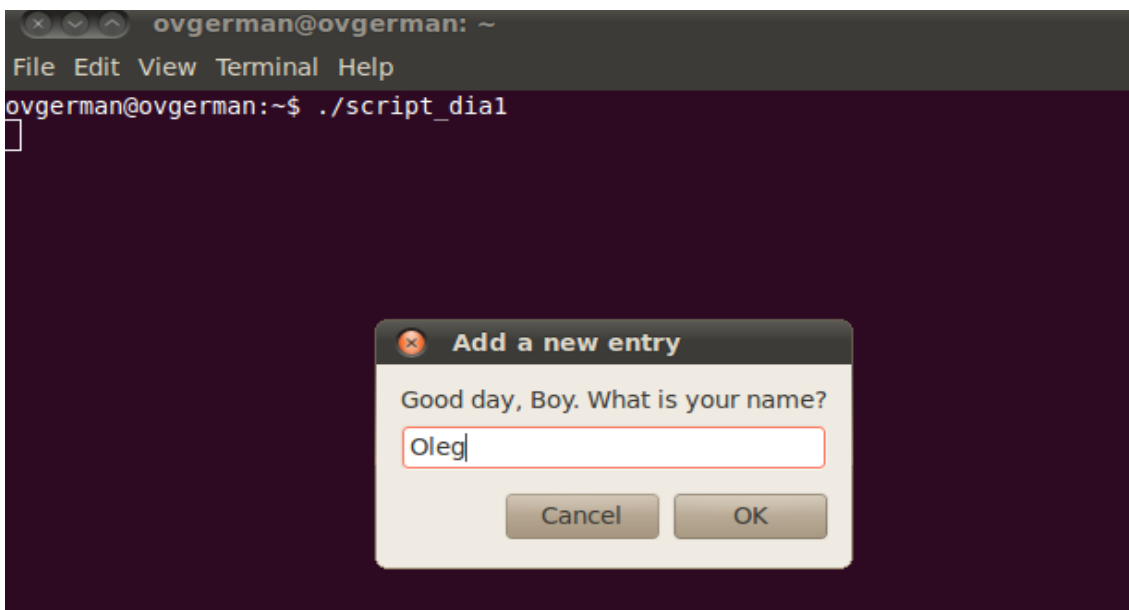
Немного изменим скрипт

```
#!/bin/sh

gdialog --title "Select" --yesno "Are you a man?" 9 18
case "$?" in
    0) gdialog --inputbox "Good day, Boy. What is your name?" 5 20 2> _1.txt
        Q_N=$(cat _1.txt)
        echo "Hie, $Q_N";;
    1) gdialog --infobox "Good day, Girl" 5, 20;;
    *) gdialog --infobox "Good day" 5, 20;;

esac
sleep 1
exit 0
```

Здесь мы предлагаем ввести имя и направляем это имя во временный файл. Временные файлы начинаются со знака подчеркивания. Наш временный файл `_1.txt`. Затем содержимое временного файла мы присваиваем переменной `Q_N`. После мы выводим на экран приветствие. Вот, как все это выглядит




```

ovgerman@ovgerman: ~
File Edit View Terminal Help
ovgerman@ovgerman:~$ ./script_dial
Nie, Oleg
ovgerman@ovgerman:~$

```

Для создания диалога в скрипте используются следующие конструкции

Команда **read**

```

echo "продолжать? (y/n): ";
read yesno;                # считывание строки (до ввода ENTER) в переменную yesno
if [ $yesno = "y" -o $yesno = "Y" ]; then
    echo Yes
else
    echo No
fi

```

```

echo "продолжать? (y/n): ";
read -n 1 yesno;            # посимвольное считывание ввода в переменную yesno
if ! $( echo $yesno | grep -q "y"); then
    echo "скрипт завершается"
    exit 1
fi

```

Команда **select**

```

select fname in *; do      # команда select организует простое меню и размещает ввод пользователя в $fname
    echo you picked $fname \($REPLY\)
    break;
done

```

Более сложный диалог представляет собой диалог типа меню. Вот пример программы

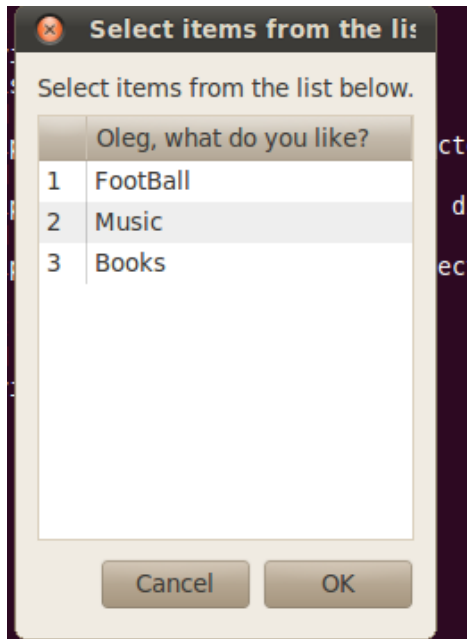
```

#!/bin/sh

gdialog --title "Select" --yesno "Are you a man?" 9 18
case "$?" in
    0) gdialog --inputbox "Good day, Boy. What is your name?" 5 20 2>_1.txt
        Q_N=$(cat _1.txt)
        gdialog --clear
        gdialog --menu "$Q_N, what do you like?" 15 30 4 1 "FootBall" 2 "Music" 3 "Books" 1>_2.txt
        Q_A=$(cat _2.txt)
        case "$Q_A" in
            1|1*|FootBall|2*|Music|1|"1"|FootBall) echo "Good, Oleg";;
            *) echo "Strange, Oleg";;
        esac;;
    1) gdialog --infobox "Good day, Girl" 5, 20;;
    *) gdialog --infobox "Good day" 5, 20;;
esac
sleep 1
exit 0

```

Здесь два диалога. Один внутри другого. Диалог меню показан на скриншоте ниже:



Техника выбора пункта меню такая же, как и выше. Номер пункта помещается в файл.

Упражнение 2

Программа генератор отчетов в формате HTML

Программа будет выводить разнообразную информацию о системе и ее состоянии в формате HTML, благодаря чему ее можно будет просматривать в веб-браузере.

Прежде всего, определим, как выглядит формат правильно сформированного HTML-документа

```
<HTML>
  <HEAD>
    <TITLE>Заголовок страницы</TITLE>
  </HEAD>
  <BODY>
    Тело страницы.
  </BODY>
</HTML>
```

Введите этот текст в текстовом редакторе и сохраните в файле с именем **foo.html**, после мы сможем открыть его, введя следующий адрес URL в Firefox: **file:///home/username/foo.html**

На первом этапе создадим программу, которая будет выводить эту разметку HTML в стандартный вывод. Откройте текстовый редактор (здесь `vim`) и создайте файл с именем `~/bin/sys_info_page`:

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

А затем введите следующую программу:

```
#!/bin/bash
# Программа вывода страницы с информацией о системе
echo "<HTML>"
echo " <HEAD>"
echo "    <TITLE>Page Title</TITLE>"
echo " </HEAD>"
echo " <BODY>"
echo "    Page body."
echo " </BODY>"
echo "</HTML>"
```

После запуска на экране должен появиться текст HTML-документа, потому что команды `echo` в сценарии посылают свои строки в стандартный вывод.

Если объединить все команды `echo` в одну, это определенно упростит в будущем добавление новых строк в вывод программы. Поэтому изменим программу, как показано ниже:

```
#!/bin/bash
# Программа вывода страницы с информацией о системе
echo "<HTML>
    <HEAD>
        <TITLE>Page Title</TITLE>
    </HEAD>
    <BODY>
        Page body.
    </BODY>
</HTML>"
```

Строки в кавычках могут включать символы перевода строки и, соответственно, содержать несколько строк текста. Командная оболочка будет продолжать читать текст, пока не встретит закрывающую кавычку.

После сохранения файла сделайте его выполняемым и попробуйте запустить:

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
```

```
[me@linuxbox ~]$ sys_info_page
```

Запустите программу снова и перенаправьте вывод программы в файл `sys_info_page.html`, чтобы затем посмотреть результат в веб-браузере:

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
```

```
[me@linuxbox ~]$ firefox sys_info_page.html
```

На втором этапе проекта разработки программы-отчета внесем в нее добавление некоторых данных. Теперь, когда программа способна сгенерировать минимальный документ, добавим в отчет немного данных.

Для этого мы используем такие возможности командного интерпретатора, как *переменные* и *константы* с присвоение им значений, а также преимущества *встроенных документов*. Для этого внесите следующие изменения:

```
#!/bin/bash
# Программа вывода страницы с информацией о системе
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"
cat << _EOF_
<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
        <P>$TIME_STAMP</P>
    </BODY>
</HTML>
_EOF_
```

С увеличением размеров и сложности программ их становится все труднее проектировать, программировать и сопровождать. Практически к любому сложному проекту с успехом можно применить методологию деления больших и сложных задач на более мелкие и простые. В этом нам поможет такой гибкий механизм командной оболочки как, функции.

Третий этап, функции командной оболочки.

В настоящий момент наш сценарий генерирует документ HTML, выполняя следующие шаги:

1. Открыть страницу
2. Открыть заголовок страницы
3. Установить название страницы
4. Закрыть заголовок страницы
5. Открыть тело страницы
6. Вывести заголовок на странице
7. *Вывести текущее время*
8. *Закрыть тело страницы*
9. Закрыть страницу

На следующем этапе разработки мы добавим несколько задач между шагами 7 и 8:

- **Продолжительность непрерывной работы системы и степень ее загрузки** — это интервал времени, прошедшего с момента последней загрузки системы, и среднее число задач, выполняемых процессором в настоящее время для нескольких отрезков времени.
- **Дисковое пространство** — информация об использовании дискового пространства на системных устройствах хранения.
- **Объем домашних каталогов** — объем дискового пространства, занятого каждым пользователем.

Если бы у нас были команды, решающие перечисленные задачи, мы бы просто добавили их в сценарий, воспользовавшись механизмом подстановки результатов команд:

```
#!/bin/bash
# Программа вывода страницы с информацией о системе
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"
cat << _EOF_
<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
```

```

        <P>$TIME_STAMP</P>
        $(report_uptime)
        $(report_disk_space)
        $(report_home_space)
    </BODY>
</HTML>
_EOF_

```

Добавим в наш сценарий *минимальные* определения функций (заглушки):

```

#!/bin/bash
# Программа вывода страницы с информацией о системе
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    return
}
report_disk_space () {
    return
}
report_home_space () {
    return
}

cat << _EOF_
<HTML>
    <HEAD>
    ...

        <P>$TIME_STAMP</P>
        $(report_uptime)
        $(report_disk_space)
        $(report_home_space)
    </BODY>
</HTML>
_EOF_

```

Функция *должна содержать* хотя бы одну команду. Команда `return` (которая является необязательной) помогает удовлетворить это требование.

Постоянное опробование сценария

В процессе разработки программ необходимо постоянно проверять их работоспособность. Запуская и тестируя программы как можно чаще, мы сможем выявить ошибки на самых ранних этапах разработки. Это существенно упрощает задачу отладки. Например, если после внесения небольших изменений и очередного запуска программы обнаружится ошибка, источник проблемы почти наверняка будет находиться в последних изменениях. Добавив пустые функции, которые на языке программистов называются заглушками, мы смогли проверить работоспособность программы на ранней стадии. Создавая заглушку, неплохо было бы включить в нее что-то, что давало бы обратную связь, позволяющую программисту оценить.

Запустите нашу программу:

```
[me@linuxbox ~]$ sys_info_page
```

Внимательно рассмотрите полученный результат.

После этого организуем **обратную связь** с программой. Изменим функции, добавив в них сообщения для обратной связи:

```
report_uptime () {
    echo "Function report_uptime executed."
    return
}
report_disk_space () {
    echo "Function report_disk_space executed."
    return
}
report_home_space () {
    echo "Function report_home_space executed."
    return
}
```

И запустим сценарий еще раз:

```
[me@linuxbox ~]$ sys_info_page
```

Что изменилось? Мы получили обратную связь. Можно с уверенностью сказать, что наши три функции выполняются как надо.

Теперь, когда каркас функций готов и работает, самое время добавить в них некий код.

```
report_uptime () {
    cat <<- _EOF_
```

```

        <H2>System Uptime</H2>
        <PRE>$(uptime)</PRE>
        EOF_
    return
}

report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>
        <PRE>$(df -h)</PRE>
        _EOF_
    return
}

```

Наконец, определим функцию **report_home_space**:

```

report_home_space () {
    cat <<- _EOF_
        <H2>Home Space Utilization</H2>
        <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    return
}

```

Для решения поставленной задачи мы использовали команду **du** с параметрами **-sh**. Однако это не полное решение задачи. Даже при том, что его можно использовать в некоторых системах (например, в Ubuntu), кое-где оно работать не будет.

Причина в том, что во многих системах для домашних каталогов выбираются разрешения, не позволяющие читать их содержимое другим пользователям, что является вполне разумной мерой предосторожности. В этих системах функция **report_home_space** в том виде, в каком она написана здесь, будет работать, только если запустить сценарий с *правами суперпользователя*.

Лучшее, что можно сделать в такой ситуации, — корректировать поведение сценария в соответствии с привилегиями пользователя, запустившего его. Мы займемся этим на следующем этапе.

Четвертый этап – ветвление программы.

На предыдущем этапе мы столкнулись с проблемой. Как помочь сценарию адаптировать свое поведение в зависимости от привилегий пользователя, запустившего его? Для решения проблемы нам необходим некий способ

«изменить направление» выполнения сценария, опираясь на результаты проверки. Выражаясь языком программистов, нам нужен способ, обеспечивающий ветвление программы.

На этом этапе мы можем использовать такие конструкции как: **if-then**, **if-then-else**, а также **case**. Кроме этого, мы можем применить такие механизмы ветвления как: составные команды, объединение выражений, операторы управления.

На предыдущем этапе в нашей программе возникла проблема: как сценарию **sys_info_page** определить, имеет ли текущий пользователь права на чтение всех домашних каталогов? После знакомства с инструкцией **if** эту проблему можно решить, добавив следующий код в функцию **report_home_space**:

```
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}
```

Здесь проверяется вывод команды **id**. Если вызвать команду **id** с параметром **-u**, она выведет числовой идентификатор действующего пользователя. Суперпользователю всегда присваивается числовой идентификатор 0. Зная это, мы сконструировали два разных вложенных документа: один пользуется преимуществом привилегий суперпользователя, а другой ограничивается домашним каталогом текущего пользователя.

Дальнейшее совершенствование нашей программы проведем в следующем упражнении.

Упражнение 3

Интерактивность программы

В сценариях, написанных нами до сих пор, отсутствует одно свойство, характерное для многих компьютерных программ, *интерактивность*, то есть возможность взаимодействия с пользователем.

Часто для организации интерактивной работы используются **меню**. Можно сконструировать программу **sys_info_page**, реализующую решение задач, перечисленных в меню.

```
#!/bin/bash
# read-menu: программа вывода системной информации,
#           управляемая с помощью меню
clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

"read -p "Enter selection [0-3] > "
if [[ $REPLY =~ ^[0-3]$ ]]; then
    if [[ $REPLY == 0 ]]; then
        echo "Program terminated."
        exit
    fi
    if [[ $REPLY == 1 ]]; then
        echo "Hostname: $HOSTNAME"
        uptime
        exit
    fi
    if [[ $REPLY == 2 ]]; then
        df -h
        exit
    fi
    if [[ $REPLY == 3 ]]; then
        if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else

```

```

        echo "Home Space Utilization ($USER)"
        du -sh $HOME
    fi
    exit
fi
else
    echo "Invalid entry." >&2
    exit 1
fi

```

Этот сценарий делится на две логические части. Первая часть выводит меню и вводит выбор пользователя. Вторая часть идентифицирует выбор и выполняет соответствующие действия.

Команда `exit` в этом сценарии препятствует выполнению ненужного кода после завершения затребованного действия. Наличие нескольких точек выхода из программы вообще считается *дурным тоном* (логику работы такой программы труднее понять), но в данном сценарии нас это устраивает.

Программа работает, но неудобна в использовании. Она выполняет только один выбранный вариант и закрывается. С помощью *циклов* реализуем многократное выполнение участков программ.

Наша программа **sys_info_page** выросла и усложнилась. Ниже приводится полный листинг программы с выделенными последними изменениями:

```

#!/bin/bash
# sys_info_page: программа вывода страницы с информацией о системе
PROGNAME=$(basename $0)
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"
report_uptime () {
    cat <<- _EOF_
        <H2>System Uptime</H2>
        <PRE>$(uptime)</PRE>
    _EOF_
    return
}
report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>

```

```

    <PRE>$(df -h)</PRE>
    _EOF_
    return
}

report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh /home/*)</PRE>
        _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh $HOME)</PRE>
        _EOF_
    fi
    return
}

usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

write_html_page () {
    cat <<- _EOF_
    <HTML>
        <HEAD>
            <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
            <H1>$TITLE</H1>
            <P>$TIME_STAMP</P>
            $(report_uptime)
            $(report_disk_space)
            $(report_home_space)
        </BODY>
    </HTML>
    _EOF_
    return
}

```

обработка параметров командной строки

```

interactive=
filename=
while [[ -n $1 ]]; do
    case $1 in
        -f | --file)      shift
                        filename=$1
                        ;;
        -i | --interactive) interactive=1
                        ;;
        -h | --help)      usage
                        exit
                        ;;
        *) usage >&2
            exit 1
            ;;
    esac
    shift
done

```

интерактивный режим

```

if [[ -n $interactive ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e $filename ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case $REPLY in
                Y|y) break
                ;;
                Q|q) echo "Program terminated."
                    exit
                    ;;
                *) continue
                ;;
            esac
        fi
    done
fi

```

вывод страницы html

```

if [[ -n $filename ]]; then
    if touch $filename && [[ -f $filename ]]; then
        write_html_page > $filename
    fi
fi

```

```
else
    echo "$PROGNAME: Cannot write file '$filename'" >&2
    exit 1
fi
else
    write_html_page
fi
```

У нас уже получился неплохой сценарий, но он еще не закончен.

Улучшения данной программы проведем в задании №2.

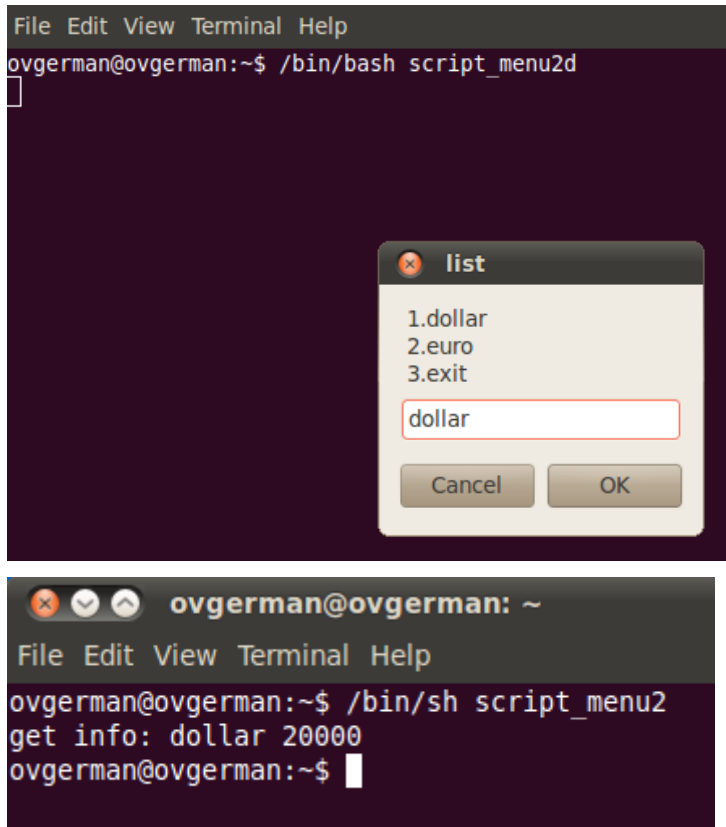
We hope you enjoy working with Linux!



ЗАДАНИЯ

Задание 1

1. Сделать пример с двумя вложенными диалогами типа YesNo.
2. Сделать пример, запрашивающий сначала имя человека, а потом профессию. Вывести имя + профессию, прочитанные в диалоге.
3. Вывести **список** с названиями валют. После выбора валюты система должна вывести ее котировку.



Можно реализовать диалог с помощью **if ... then**:

```
#!/bin/sh
touch f1
echo "dollar 20000">f1
echo "euro 22000">>f1
echo "rus 300">>f1
gdialog --inputbox "Your currency?" 5 20 2>_1.txt
z=$(cat _1.txt)
if [ "$z" = "dollar" ]
then
echo "dollar = 20000rub"
elif [ "$z" = "euro" ]
then
echo "euro = 23000rub"
else
echo "we do not know for $z"
fi
```

Здесь следует обратить внимание на проверку условия. Запомните, если выполняем сравнение строк, то в квадратных скобках строки записываем в двойных кавычках. При этом (!) нужно разделять все слова пробелами как слева, так и справа. Наконец, слово `then` договоримся писать на новой строке. Заметим, что диалог отображается списком, поскольку указаны символы перехода на новую строку в команде `dialog`.

4. Измените предыдущую программу так, вместо списка валют предлагалось оконное **меню** валют. Чтобы программа работала в цикле. Для выхода из цикла нужно вместо названия валюты вводить `exit`.
5. Измените предыдущую программу так, для выхода из программы в меню была кнопка закрытия программы «`exit`».

Работа диалогов должна быть подтверждена скриншотами в отчетах.

Задание 2

Создайте копию проекта представленного в упражнении №3. Внесем изменения в новом проекте, добавив вывод информации о домашнем каталоге каждого пользователя и включив в вывод общее число файлов и подкаталогов в каждом из них:

```
report_home_space () {
    local format="%8s%10s%10s\n"
    local i dir_list total_files total_dirs total_size user_name
    if [[ $(id -u) -eq 0 ]]; then
        dir_list=/home/*
        user_name="All Users"
    else
        dir_list=$HOME
        user_name=$USER
    fi
    echo "<H2>Home Space Utilization ($user_name)</H2>"
    for i in $dir_list; do
        total_files=$(find $i -type f | wc -l)
        total_dirs=$(find $i -type d | wc -l)
        total_size=$(du -sh $i | cut -f 1)
        echo "<H3>$i</H3>"
        echo "<PRE>"
        printf "$format" "Dirs" "Files" "Size"
        printf "$format" "----" "-----" "-----"
```



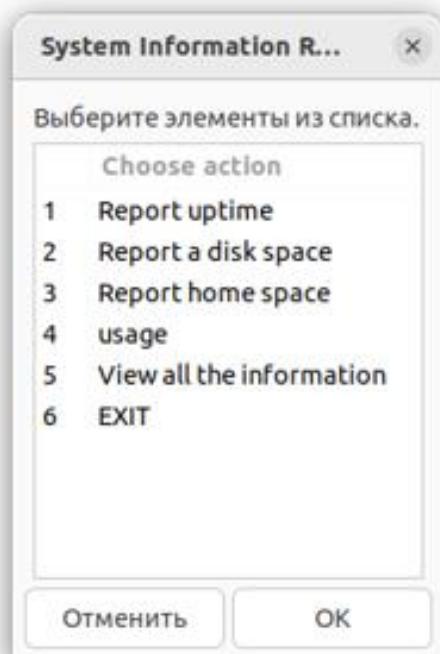
```
        printf "$format" $total_dirs $total_files $total_size
        echo "</PRE>"
    done
    return
}
```

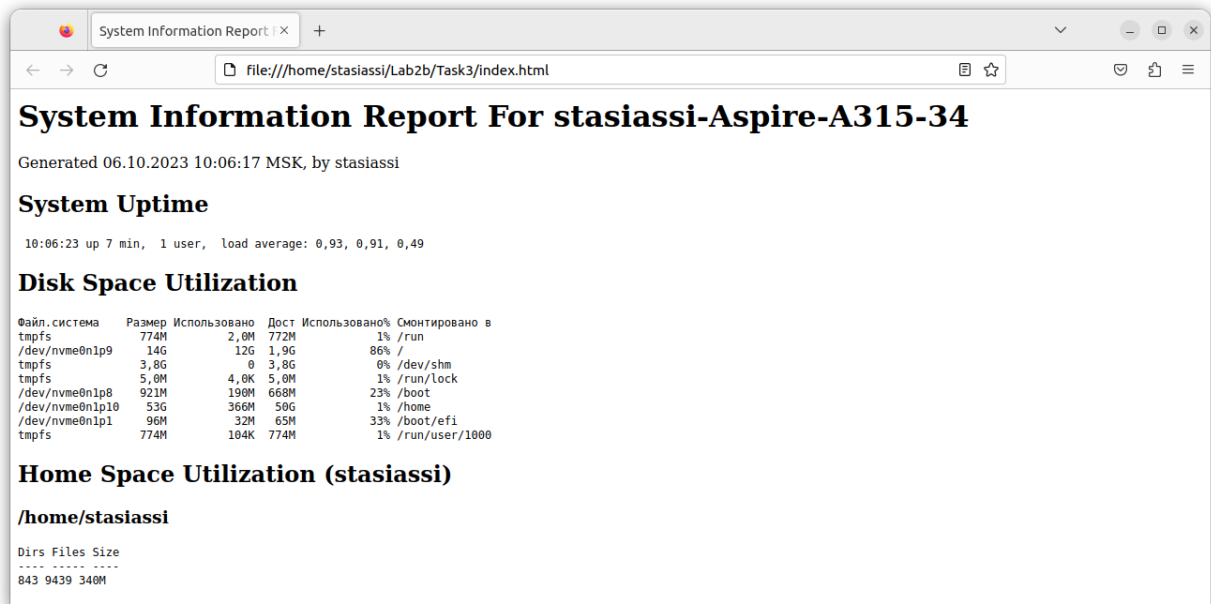
В этой новой версии проверяется наличие привилегий суперпользователя, но вместо того, чтобы выполнить полный набор операций в каждой из ветвей `if`, здесь устанавливаются некоторые переменные, которые затем используются в цикле `for`. В функции использованы несколько локальных переменных и команда `printf` для форматирования части вывода.

Задание 3

Доведите проект до совершенства. Для сценария выполненного в задании №2 создайте графическое диалоговое окно, с интерактивным меню выбора функций. Информация должна автоматически передаваться в HTML-файл и открываться в браузере.

Результат программы должен выглядеть примерно вот так.





System Information Report For stasiassi-Aspire-A315-34

Generated 06.10.2023 10:06:17 MSK, by stasiassi

System Uptime

10:06:23 up 7 min, 1 user, load average: 0,93, 0,91, 0,49

Disk Space Utilization

Файл. система	Размер	Использовано	Дост	Использовано%	Смонтировано в
tmpfs	774M	2,0M	772M	1%	/run
/dev/nvme0n1p9	14G	12G	1,9G	86%	/
tmpfs	3,8G	0	3,8G	0%	/dev/shm
tmpfs	5,0M	4,0K	5,0M	1%	/run/lock
/dev/nvme0n1p8	921M	190M	668M	23%	/boot
/dev/nvme0n1p10	53G	366M	50G	1%	/home
/dev/nvme0n1p1	96M	32M	65M	33%	/boot/efi
tmpfs	774M	104K	774M	1%	/run/user/1000

Home Space Utilization (stasiassi)

/home/stasiassi

Dirs	Files	Size
----	-----	-----
843	9439	340M

«Easy things should be easy and hard things should be possible»
«Простые вещи должны быть простыми, а сложные вещи должны быть
возможными»



Контрольные вопросы:

Возможности командной оболочки и скриптов Shell

- 1) Что такое *переменный* и *константы*, чем они отличаются? Как их различает командная оболочка?
- 2) Как командная оболочка определяет тип значения переменных?
- 3) Какое *типичное соглашение о написании имен переменных* и констант принято программистами? Для чего это делается?
- 4) Что такое *переменные окружения*? Назовите команды с помощью которых можно получить значения переменных окружения?
- 5) Что такое *пользовательские переменные* и для чего они используются?
- 6) Что такое *область видимости* переменных? Какие типы области видимости переменных существуют?
- 7) Что такое *конфликт имен переменных* и как решается эта проблема?
- 8) Что такое *функции* командной оболочки?
- 9) Что такое «код завершения программы»?
- 10) Какой командой и каким средством в скриптах обеспечивается интерактивность программы?

Графический интерфейс диалога

- 11) С помощью каких утилит реализуется средства примитивного графического интерфейса в Linux-скриптах?
- 12) Укажите какими интернет-источниками вы пользовались при изучении и разработке графического диалога.
- 13) Какие существуют типы окон примитивного графического интерфейса Linux-скриптов?
- 14)

Дополнительная информация

Подробно песочница представлена к книге: Шоттс У. «Командная строка Linux. Полное руководство.» — СПб.: Питер, 2017. — 480 с.: ил. — (Серия «Для профессионалов»). на страницах 134-145, 346-405, 433-435, 440 – 441.

Робачевский А. М. Операционная система UNIX®. - СПб.: 2002. - 528 ил.

Иванов программирование (окружение)

Переменные окружения

\$DIRSTACK	содержимое вершины стека каталогов
\$EDITOR	текстовый редактор по умолчанию
\$EUID	эффективный UID. Если вы использовали программу su для выполнения команд от другого пользователя, то эта переменная содержит UID этого пользователя.
\$UID	содержит реальный идентификатор, который устанавливается только при логине.
\$FUNCNAME	имя текущей функции в скрипте.
\$GROUPS	массив групп, к которым принадлежит текущий пользователь
\$HOME	домашний каталог пользователя
\$HOSTNAME	ваш hostname
\$HOSTTYPE	архитектура машины.
\$LC_CTYPE	внутренняя переменная, которая определяет кодировку символов
\$OLDPWD	прежний рабочий каталог
\$OSTYPE	тип ОС
\$PATH	путь поиска программ
\$PPID	идентификатор родительского процесса
\$SECONDS	время работы скрипта(в сек.)
\$#	общее количество параметров переданных скрипту
\$*	все аргументы переданные скрипту(выводятся в строку)
\$@	тоже самое, что и предыдущий, но параметры выводятся в столбик
\$!	PID последнего запущенного в фоне процесса
\$\$	PID самого скрипта

* Переменные окружения могут обличаться в разных дистрибутива.

Интернет источники

