BASH SCRIPTING

Everything you need to know about Bash

For beginners

Rufus Stewart

Introduction

Is a program that works to receive commands from you and then ask the kernel to execute these commands. Ie acts as an intermediary between you and the operating system or in other words it represents the outer layer or interface of the system, it takes orders from you and gives you the result. Then it shows the inductor again, indicating termination, and waiting for you to enter more commands. This is why the shell is called a command interpreter, which is similar to command.com in windows. It also contains a programming language similar to high-level languages is very strong through which you can solve any problem you face. Also, one of the important features is the possibility of making a file containing a set of programs to create another program and this enables you to produce what is known as script, and there are many types of shell, but the most famous and used are the two main types:

· **The Bourne shell includes the sh, ksh, bash**

· **The C shell includes csh, tcsh**

If you are using a Bourne shell, the default inductor for you as a user is $.

What is shell propmt?

The inductor, whether it is% or $, depends on the type of shell you are using.When the prompt appears, it means that the shell is waiting for you to type the commands to execute.If you have entered the system as the root of any system administrator, the prompt will appear. The following are the advantages and disadvantages of each type of shell quickly:

The first type includes:

The Bourne shell and symbolized by sh The Korn shell and symbolized by ksh The Bourne againshell and symbolized by the bash

· The second type includes:

C shell TENEX / TOPS C shell TCSH

The beginning of the shell was with the Bourne shell where stephen R.Bourne wrote it on the unix system in the mid 1970's at the AT&T lab. Therefore, some call it the shell, "The Shell," because it first appeared. This shell is usually located in the path / bin / sh As mentioned earlier, the shell is a command interpreter and also contains a programming language using syntax formula ALGOL language, and contains the following things:

Variables, functions, program control commands, repeating commands, processing commands, mathematical expressions, etc.

All the Bourne-type types share these features, but the disadvantage of this original type is the difficulty in using it. There is no auto-complete feature, and there is no save command that has been executed.

Ten years later, in 1980, Bill Joy wrote the C shell, avoiding the difficulty found in the previous type and also making the syntax formula from ALGOL to C language.

Some of the features found in it:

The property of completing the text, save previously written commands. Usually we find this shell inside the / bin / csh path

Its disadvantages are: Weakness in supporting I / O operations lacks functions, so it is not recommended for programming, only for use.

The TENEX / TOPS C shell, symbolized by tcsh, is an improved version of the cshell where additions of many features, including the use of the arrows (up - down) to view the commands that have been executed as well as the arrow (right - left) to modify these commands. At this time, after two types of shell were available, David Korn in AT&T wrote the Korn shell and symbolized by ksh

It combines the ease of the C shell, as well as the Bourne shell, making ksh the preferred choice for many users.

In general, the ksh is compatible with the first type, which is the sh. There are 3 copies of this paralysis:

The official symbolized by ksh which we talked about and the public domain and symbolized by pdksh and finally the version of Desktop and symbolized by dtksh. Shell programmers often use the first and second versions. The third dtksh is for Shell users. The ksh is usually found in the / bin / ksh or / usr / bin / ksh paths.Other versions can be downloaded from the net if it does not come with your system.

Finally, the Bourne again shell, which is currently used in most distributions where the developer Brian fox collected the previous features and put it in it so it is now preferred and is usually found in the path / bin / bash or path / bin / sh where it became the default shell.

When does paralysis begin and end?

When you log on to the system, getty starts to work where you are asked to type your username and passwords and then passes it to the login program that checks the entered data, by comparing it with / etc / passwd and it looks like this:

SudaNix: x: 500: 500: ahmad: / home / SudaNix: / bin / bash

In the case of matching, the paralysis in the file will be executed, in this case is / bin / bash and when you exit logging Shell ends work and to change the type of paralysis change it by typing the command: / bin / csh Here changed to Cshell and exit

· Interactive and non-interactive paralysis !!

When you see the prompt prompt, this means that immobilization works interactively interactive, I mean interactively, that is, you are expected to write commands to execute This is the usual normal situation where you enter the system, then the immobilization begins to work .. Then you write some commands, and exit the system Shell is suspended.

It is contradicted by non-interactive paralysis:

It means that there is no interaction between you and Shell, that is, you will not be waiting for you to write any commands, where he executes commands saved inside a file and when it reaches the end of the file stops working and this is known as the script. Two basic concepts are:

Commands

It is a program that you can run by typing its name and then press enter.

Example: The date command shows the day, the date, the hour. Note that the $ prompt is reappeared and this indicates that the program has finished executing.

Another example is the command, which shows the names of all users of this system in addition to some information and there are several types of commands, namely, simple commands, complex, complex.

Simple commands

It is clear from its name that it is simple, which contains only the name of the program, such as the two previous date, who as well as many of the commands that I will address later.

Complex commands

A simple program name plus one or more arguments. An example to make it clear:

As mentioned, we give you information about all users and their access times to the system, but by typing it: The username you are working on and the time you access the system. So am and i are intermediaries who interfere with who adjusts his behavior. Most of the existing commands accept many arguments that change their default behavior and we will see that later.

Compound commands

They are commands that combine simple and complex commands. They are separated by semicolon; Example command: date; who am i;

This complex command executes the simple command date first and then executes the complex command whoam i and commands are separated using a semicolon and if not placed, ie the command is written as: date who am i will happen an error .. and will not execute an error message because this interpreter I think that date is the order or program, and who am i are arguments for him and this is wrong

It is worth mentioning is that you can write a semicolon at the end of any command, whether complex or simple or complex example command: date; The result is the same as the date

One final note on orders:

It is when writing any command and before the execution of the shell to make sure it is in a specific path PATH in the case of the command or the program in which it executes, and in the absence of it gives you the error command not found

What is shell script?

It is a file with a set of commands to perform a specific purpose, where the shell executes them in an interactive way you will see the definition through the following example: For example, the joiner box contains tools (screwdrivers, screws, wood, hammer ... etc). He can build chairs, tables, cabinets, etc. Using the same tools, but with a certain composition, he can build anything he wants.

This example is very similar to shell scripting. To build anything you want, you must use the right tools and call them commands or programs, which are mentioned above.

Write the first shell script: Open any document - Text file - and type the following:

date; who;

Save the file by any name and let it be sudanix

Now you run this script, type:

/ bin / bash sudanix

Now the result will appear which is the output of the date command and the output of the command who.

To convert a script into a scriptable executable ??

One of the most important steps, in order to make the file executable .. Any we execute by name only. she:

Convert the file into an executable file by command

chmod a + x ./sudanix

But we have to make sure which shell will be used. Because we don't guarantee that all shell users are working on bash, we will add this line to the start of the script.

#! / bin / bash

Note that this line at the beginning of the script. Specifically in the first line !! Otherwise, the script will be as follows:

#! / bin / bash

date; who;

Now to execute this script write it on the following form with a note to be in the same folder that contains the script.

./sudanix

The bash shell is the default shell of the GNU system

The GNU project, referring to GNU's Not UNIX, provides tools for the management of Unix-like systems, free operating systems that comply with Unix standards. One of these tools is bash, a shell coinciding with the first sh shell written by Stephen Bourne, and has useful properties from Korn and C shells - their abbreviations ksh and csh respectively. It is designed to comply with IEEE POSIX P1003.2 / ISO 9 945.2 standard for shells and software.

It also features improved sh for both programming and interactive use, such as command line editing, unlimited history of previous commands, job control, and shell functions.), Aliases, indexed arrays of unlimited sizes, and arithmetic integers in any base from 2 to 64. bash can also handle most sh text scripts without modification.

The Bash Initiative, like the rest of GNU's projects, was originally launched to preserve the freedom to use, study, copy, modify, and republish programs, and to promote and disseminate those rights, as these are the catalysts for creativity. In any other coincidence.

The advantages are found only in bash

Invocation

There are many multi-character options that you can use, as well as single-character options that can be set using the set command that is located by default in the shell. In this documentation we will show some other popular options, and you can find the complete list in the info guide, in Bash features and then Invoking Bash.

Startup files in Bash

Startup files are scripts that Bash reads and executes at startup. The following explanation describes the different ways to start shells, and the startup files that are read accordingly.

Invoke as an interactive login shell, or with login--

Interactivity here means that you can enter commands, and shell does not work because of the activation of the text code. "Login shell" means that you enter the shell only after authenticating with the system, often by entering your username and password.

Read files:

etc / profile /

bash_profile./~, bash_login./~, or profile./~: Reads the first readable file.

bash_logout./~ when you log out.

Error messages are printed if configuration files exist but cannot be read, and if a file does not exist, bash looks for the next file.

Call as an interactive shell without logging in

A non-login shell means that you don't have to authenticate with the system, as if you open Terminal by clicking an icon or item in a list, it's a non-login shell.

Read files:

bashrc./~

This file is usually referenced in bash_profile / ~:

if [-f ~ / .bashrc]; then. ~ / .bashrc; fi

See conditional structures in Bash for more if structure

The call is interactive

All text codes use non-interactive shells, and are programmed to perform specific tasks and cannot be programmed to perform other tasks otherwise.

Read files:

Selected by BASH_ENV

PATH is not used in this file for searching. If you want to use it, please indicate the full path and file name.

Call by order from within sh

Bash tries to behave like the old sh software written by Steve Bourne, while trying to comply with POSIX standards.

Read files:

etc / profile /

profile./~

The ENV variable can refer to additional startup information.

POSIX mode

This option is activated using the set command:

set -o posix

Or by invoking the bash program with the - posix option, bash will then try to comply with POSIX standards for shells. Also, initializing the POSIXLY_CORRECT variable produces the same result.

Read files:

Specified by the ENV variable.

Remote paging

Files read when bash starts with the rshd command:

bashrc /. ~

Avoid using r-tools

Be aware of the dangers of using unsafe tools such as rlogin, telnet, rsh, and rcp, as they transmit sensitive data over the network without encryption. If you need tools to remotely administer the system, transfer files, etc., use a Secure SHell Protocol (ssh) protocol, which is available for free from http://www.openssh.org. You will also find different programs for non-Unix-like systems, see your local software mirror.

The call when the UID is not identical to the EUID

No files are read at startup in this case.

Interactive shells

What is interactive chance?

An interactive shell reads and writes at a user's terminal as follows:

Input and outputs are connected to the terminal, and interactive bash behavior starts when the bash command is called without adding any non-option arguments, except if the option is a string that is read, or when the shell is triggered to read a standard input, This allows the initialization of positional parameters, see Bash environment.

Is this coincidence interactive?

To find out if a shell is interactive, look at the content of the special parameter -, if it has an "i", the shell is interactive:

hsoub: ~> echo $ -

himBH

In the non-reactive shell, the PS1 inductor is not initialized, ie it goes back to zero.

Interactive shell behavior

Bash reads startup files.

Function control is enabled by default.

Inductors are configured, and the PS2 environment variable is activated for multi-line commands, usually set to <, which is also what you get when the shell assumes that you entered an incomplete command, such as forgetting quotes or one or more items that cannot be neglected in a command. etc.

By default, commands are read from the command line using readline.

Bash interprets the ignoreeof option when you receive End Of File (or EOF for short) instead of exiting directly.

Command history and log expansion are enabled by default, and the log is saved in a file referred to by HISTFILE when the shell closes. By default, HISTFILE refers to bash_history./~.

Alias expansion is automatically activated.

The SIGTERM signal is ignored in the absence of traps.

In the absence of traps, SIGINT is captured and handled, so pressing Ctrl + C, for example, won't get you out of the interactive shell you're in.

SIGHUP signals can be set to be sent to all functions at exit, with the huponexit option.

The commands are executed as soon as they are read.

Periodically inspect mail bash.

Bash can be set to close when you encounter unreferenced variables. This behavior is disabled in interactive mode.

When shell commands encounter redirect errors, they will not cause the shell to close or exit.

Repeated errors for special built-in commands while using them in POSIX mode do not cause an accident. See the commands included in the shell for more on the included commands.

Failure to exec will not cause the exit of chance.

Syntactic errors of the Parser will not cause the shell to close.

Simple check of arguments cd is activated by default.

Auto-logoff after a period specified in the TMOUT variable has elapsed, is also enabled.

For more explanation, see Variables in Bash, More Bash options, Capture signals in Bash for more signals, and Bash expansions that discuss the different expansions that occur when you enter an order.

Conditional Expressions

Conditional expressions are used via the combined [] command, the included commands and test, and the built-in commands. Expressions can be mono or binary.

There are also text operators (string operators) and numeric comparison operators. These two types are binary operators, requiring two elements to perform the operation on.

If the FILE argument of an original is dev / fd / N /, the file descriptor N is selected, but if it is in one of those dev / stdin / or dev / stdout / or dev / stderr /, the file descriptor 0, 1 , Or 2 respectively.

See the detail description for conditional structures for more conditional expressions, and see redirection and file descriptions for more file descriptors.

Arithmetic expressions in shell Shell Arithmetic Expressions

Coincidence allows arithmetic expressions to be evaluated as one of the expansions of the shell, or by using the let command, and evaluated as fixed width integers without overflow, except that dividing by 0 is captured and notified as an error. The rules of the C programming language on factors, their precedence and their associativity apply to their counterparts here, see bash environment.

Alternative commands

Alternative commands allow a word to be replaced by a text of several words when used as the first word of a simple command. The shell keeps a list of alternative commands that can be created or canceled using alias and unalias.

Bash reads at least a full line of inputs before executing any commands on that line. Alternative commands are also expanded when the command is read and not executed, so if the characters of one of the alternate commands on a line match the letters of another command in the same line, the command that the alternate command holds is only executed when the next line of input is read.

Similarly, alternative commands are expanded when a function description is read, not when executed, because the function description itself is a compound command, so alternative commands in a function are only available after the function is executed. We will discuss alternative orders in detail in the alternative orders.

Chapter One

Create and run the Bash program

Writing and naming

If you have a series of commands that you execute frequently and want to shorten the execution time or execute them automatically, you can put them in an empty file written in a particular format and read by chance as a program that you execute when you call it by typing the name of the file it contains, and these programs are called shell programs Scripts).

These programs can be used to automate tasks using the cron tool or in boot and shutdown procedures in Unix systems, where daemons and services are defined in init scripts.

To create a shell program, open an empty file in your favorite text editor.It doesn't matter what type of editor you're using.Most of them are doing the job we want now, writing a Bash shell program.Use VIM if you like, emacs,

gedit, dtpad, etc.

But what might make you use a sophisticated editor like VIM or emacs, is that you can configure them to recognize the texts you write and highlight them in different colors - including texts and shell commands that we will write - and that reduces the mistakes that beginners usually make, such as omitting closing parentheses () Or forget about semicolons; .

Highlight texts in VIM

Use one of the following three commands to highlight Text Highlighting in the VIM editor:

syntax enable: or sy enable: or syn enable:

It is useful to add that setting to a vimrc file. In order to install it by default.

Place the Unix commands in the file you created as if you were going to type those commands on the command line (see Execute Commands in Bash). These commands can be shell functions, embedded commands, Unix commands, etc. Now give that file a name that describes the tasks it executes and make sure that the name does not conflict with pre-existing commands.To ensure this, the names of shell programs often end in .sh, but this does not prevent the presence of other text codes in the system with the same name you choose, search in the system for files Or text codes with the same name using file and program search commands like which or whereis:

which -a script_name

whereis script_name

locate script_name

script1.sh

In this example, we will use the echo command to print on-screen notifications that tell the user what will happen before each task is executed. It is important to tell the user what is happening to avoid panic because the user feels that the program is "doing nothing". See detailed user notification in writing interactive Bash programs.

The script1.sh file

#! / bin / bash

clear

echo "The script starts now."

```
echo "Hi, $ USER!"
echo

echo "I will now fetch you a list of conntected users:"
echo
w
echo

echo "I'm setting two variables now."
COLOR = "black"
VALUE = "9"
echo "This is a string: $ COLOR"
echo "And this is a number: $ VALUE"
echo

echo "I'm giving you back your prompt now."
echo
```

You may have to train yourself to write shell programs by typing this example into the image yourself.It is also useful to create a / ~ scripts folder to save your text codes in, then add that folder to the contents of the PATH variable as follows:

```
export PATH = "$ PATH: ~ / scripts"
```

Use an editor that supports text highlighting for shell commands if you are a beginner.This feature is supported in editors such as VIM - gvim - (x) emacs-kwrite and other editors, so make sure your editor supports Text Highlighting by looking at the Document Highlighting feature. That editor.

Various inductors

The inductors used in this explanation may differ to simulate real-life situations rather than using the $ inductor, which is frequently used in explanations. However, the root user prompt will remain constant in the annotation, and will be #.

Program implementation

The program is executable for those entitled to it, that is, to whom the program owner grants permission to execute it, make sure that you get the permissions you want while adjusting the permissions of the program, and then you can run the program as you run any command in chance:

wiki @ hsoub: ~ / scripts $ chmod u + x script1.sh

wiki @ hsoub: ~ / scripts $ ls -l script1.sh

-rwxrwxr-x 1 wiki wiki 331 Jul 31 03:12 script1.sh

wiki @ hsoub: ~ / scripts $ script1.sh

The script starts now.

Hi, wiki!

I will now fetch you a list of connected users:

03:12:48 up 3:42, 1 user, load average: 1.33, 1.51, 1.67

USER TTY FROM LOGIN @ IDLE JCPU PCPU WHAT

wiki tty7: 0 23:30 3: 42m 12:51 0.63s / usr / lib / gnome-session / gnome

I'm setting two variables now.

This is a string: black

And this is a number: 9

I'm giving you back your prompt now.

wiki @ hsoub: ~ / scripts $ echo $ COLOR

wiki @ hsoub: ~ / scripts $ echo $ VALUE

wiki @ hsoub: ~ / scripts $

This is the most common way to implement a shell script, preferably in a subshell. The variables, functions, and aliases that were created in that sub shell will be defined for this session from Bash for that sub shell only. As it was when that shell closed and control of the mother coincided again, everything is cleaned and all changes made by the program to chance are canceled. If you do not put the scripts folder in the PATH variable and it is not the current folder. You can also execute the program with this command - script_name indicates the name of your program -:

./script_name.sh

The program may be implemented in a particular shell, although we do this if we want to obtain certain behavior, such as checking if the program is working in another shell or printing traces for debugging:

rbash script_name.sh

sh script_name.sh

bash -x script_name.sh

The shell specified in any of the above commands will act as a sub-shell of your current shell and execute the program. You may do this when you want the program to run with specific options or under circumstances not specified in it. If you want to execute the program within the same shell as you are, you must first use the source command:

source script_name.sh

source =.

The source command included in the Bash shell is synonymous with a command. In Bourne coincidence.

Shell does not need any execute permissions in that case, and commands are executed in the current context of the shell, so any changes that occur to your current environment will be visible and remaining after the program is finished executing:

wiki @ hsoub: ~ / scripts $ source script1.sh

wiki @ hsoub: ~ / scripts $ echo $ COLOR

black

wiki @ hsoub: ~ / scripts $ echo $ VALUE

9

wiki @ hsoub: ~ / scripts $

In which coincidence will the program work?

You must specify the shell that will execute the program if you want to execute it in a sub-shell. The shell in which you wrote the program may not be the default shell of your system, so your program commands may produce errors when executed in the wrong shell.

The first line in the program specifies the type of shell, and the first two characters in that line must be # !, followed by the path of the shell that will execute the program. Also, do not start your program with a blank line as blank lines are read as lines as well, and all programs in this explanation will start with the following line:

#! / bin / bash

The preceding line indicates that the Bash is present in the bin /.

Add comments

Be aware that your software may be seen and read by others.

The comments make it easier for you to modify the program and go back to what you wrote again. What's new if you haven't recorded comments about what you did, how you did it, and why you chose those orders or modifications?

Copy the script1.sh file in the example of creating and running a Bash program named commented-script1.sh, and we will modify its content so that the comments describe what the commands do. Be aware that anything encountered by chance after the # sign in any line is ignored, and only appears when you open the source file. For the program:

#! / bin / bash

This program scans peripheral content, greets the user and gives information about users

Callers at program execution time. The two variables used as examples are formatted and presented

clear # Clear terminal contents immediately

echo "The script starts now."

echo "Hi, $ USER!" # The dollar sign is used to fetch the content of a variable

echo

echo "I will now fetch you a list of connected users:"

echo

w # Show who is signed in and what to do

echo

echo "I'm setting two variables now."

COLOR = "black" # Set a local variable to the shell

VALUE = "9" # Set a local variable to the shell

echo "This is a string: $ COLOR" # Display the variable content

echo "And this is a number: $ VALUE" # View the variable content

```
echo

echo "I'm giving you back your prompt now."
echo
```

The first lines in shell programs are usually comments that explain the nature of the tasks the program executes, and then each piece of code is accompanied by one or more comments to illustrate. You will find an example in the init.d folder of your system, as init codes are well documented because they are readable and modified by any Linux user.

Chapter II

Bash environment

The Bash environment can be set up at the system level as a whole or for each user individually, and various configuration files are used to control shell behavior. These files contain shell options, variable settings, function definitions, and other Bash shell builders that make it easier for us to create a comfortable and fulfilling work environment. .

You can choose names for variables as you like, with the exception of reserved names for Bourne, Bash, and special parameters. Bash uses a quoting system to exclude special meanings from one or more characters if no special processing is needed, since many characters have two or more meanings depending on the environment.

Bash also uses multiple methods to extend command-line entries in order to determine which commands to execute.

Shell initialization files

System-wide Configuration files

Etc / profile /

Bash reads the instructions in etc / profile / if invoked interactively via the login option - or if called as sh, these instructions usually set the PATH - USER - MAIL - HOSTNAME - HISTSIZE variables, which are shell variables.

The umask value in etc / profile / is also set on some systems; in others this file contains pointers to other configuration files such as:

/ Etc / inputrc file, which is a system-wide configuration file for the readline command, where you can configure the bell style for the command line.

The / etc / profile.d / folder, which contains files that set the behavior of specific programs at the system level.

All settings you want to apply to all users in your system must be present in that -etc / profile / - file, and may look similar to the following:

```
# / etc / profile

# System wide environment and startup programs, for login setup

PATH = $ PATH: / usr / X11R6 / bin

# No core files by default
ulimit -S -c 0> / dev / null 2> & 1

USER = "` id -un` "
LOGNAME = $ USER
MAIL = "/ var / spool / mail / $ USER"

HOSTNAME = `/ bin / hostname`
HISTSIZE = 1000

# Keyboard, bell, display style: the readline config file:
if [-z "$ INPUTRC" -a! -f "$ HOME / .inputrc"]; then
    INPUTRC = / etc / inputrc
fi

PS1 = "\ u @ \ h \ W"
```

```
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC
PS1

# Source initialization files for specific programs (ls, vim, less, ...)
for i in /etc/profile.d/*.sh; do
    if [-r "$ i"]; then
        . $ i
    fi
done

# Settings for program initialization
source /etc/java.conf
export NPX_PLUGIN_PATH = "$ JRE_HOME / plugin / ns4plugin /: / usr /
lib / netscape / plugins"

PAGER = "/ usr / bin / less"

unset i
```

This configuration file sets some basic shell variables in addition to some variables required by any user running Java or any Java application in their browser. See variables in Bash.

See conditional structures in Bash for more if if conditionals used in the previous example, and repeating tasks in Bash that discuss iterative loops such as for.

The Bash source contains sample profile files for general or individual use, and those files - the file in the example above - undergo several changes before they are used in your environment.

File etc / bashrc /

It is probably best to put Bash configuration files in this file if you are working on systems with different types of shells, for example other shells - Bourne shell - read the etc / profile / file.

Errors generated by shells that do not understand the syntax of the Bash shell can be avoided by separating configuration files for the different types of shells. Initialize shell initialization at login.

You may find that the / etc / profile file in your system contains only the shell environment and program startup settings, while / etc / bashrc / contains system-wide definitions of shell functions and their alternative commands. The / etc / bashrc file may also be referenced within the / etc / profile file or in the user's shell configuration files.

The source contains samples of bashrc files, and you'll find a copy in usr / share / doc / bash-2.05b / startup-files /. Here's part of what is in the bashrc file that comes with the Bash documentation:

```
alias ll = 'ls -l'
alias dir = 'ls -ba'
alias c = 'clear'
alias ls = 'ls --color'

alias mroe = 'more'
alias pdw = 'pwd'
alias sl = 'ls --color'

pskill ()
{
    local pid

    pid = $ (ps -ax | grep $ 1 | grep -v grep | gawk '{print $ 1}')
    echo -n "killing $ 1 (process $ pid) ..."
    kill -9 $ pid
    echo "slaughtered."
}
```

It contains useful aliases as well as generic aliases, as they ensure that the command you enter is executed even if you misspell it, see Creating and Deleting Alternate Orders. This file also contains the pskill function, see functions in Bash for more explanation of functions.

User settings files

Can't find those files on my computer !?

You may not find these files in your home folder.

File bash_profile./~

This file is best used if you set up individual user environments. Users here may add additional options or change the default settings:

```
wiki ~> cat .bash_profile
############################################################
###############
# #
```

```
# .bash_profile file #
# #
# Executed from the bash shell when you log in. #
# #
############################################################
###############

source ~ / .bashrc
source ~ / .bash_login
case "$ OS" in
  IRIX)
    stty sane dec
    stty erase
    ;;
# SunOS)
# stty erase
# ;;
  *)
    stty sane
    ;;
Esac
```

This user, with the above settings, sets the backspace button to sign in on different operating systems, in addition to the bashrc files. And bash_login. User-specific information has been read.

File bash_login./~

This file contains specific settings that are usually implemented only when you log on to the system. In the following example, we will use it to set the umask value and display a list of connected users upon logging in. This user will also see a calendar in the current month:

```
############################################################
####################
# #
# Bash_login file #
# #
# commands to perform from the bash shell at login time #
# (sourced from .bash_profile) #
```

```
# #
##########################################################
####################
# file protection
umask 002 # all to me, read to group and others
# miscellaneous
w
cal `date +` `% m "` `date + '% Y' "
```

This file will be read in the absence of bash_profile./~.

Profile./~

The profile./~ is read in the absence of bash_profile./~ and bash_login./~, which may contain the same settings that other shells can access. Note that these shells may not understand the linguistic structure of the Bash shell.

File bashrc./~

It is now common to see a non-login shell because the user logs in from the graphical interfaces, and the user does not enter their data in that case - username and password - the authentication process does not take place, and Bash searches for the bashrc./~ file, so that file is Mechanism in the read files at logon, which means you don't need to enter the same settings in multiple files.

In the following example of a bashrc file. For a user, some alternative commands have been specified and variables set for specific programs, after reading the / etc / bashrc file:

```
wiki ~> cat .bashrc
# /home/franky/.bashrc

# Source global definitions
if [-f / etc / bashrc]; then
        . / etc / bashrc

fi

# shell options

set -o noclobber

# my shell variables

export PS1 = "\ [\ 033 [1; 44m \] \ u \ w \ [\ 033 [0m \]"
```

```
export PATH = "$ PATH: ~ / bin: ~ / scripts"

# my aliases

alias cdrecord = 'cdrecord -dev 0,0,0 -speed = 8'
alias ss = 'ssh octarine'
alias ll = 'ls -la'

# mozilla fix

MOZILLA_FIVE_HOME = / usr / lib / mozilla
LD_LIBRARY_PATH = / usr / lib / mozilla: / usr / lib / mozilla / plugins
MOZ_DIST_BIN = / usr / lib / mozilla
MOZ_PROGRAM = / usr / lib / mozilla / mozilla-bin
export MOZILLA_FIVE_HOME LD_LIBRARY_PATH MOZ_DIST_BIN
MOZ_PROGRAM

# font fix
alias xt = 'xterm -bg black -fg white &'

# BitchX settings
export IRCNAME = "frnk"

# THE END
wiki ~>
```

You'll find more examples in the Bash package, and remember that existing file templates may need modifications before you can use them in order to work in your environment.

File bash_logout./~

This file contains specific instructions for logging off. In the following example, it is a single statement "clear" to leave the window empty after closing, and is useful in remote connections:

```
wiki ~> cat .bash_logout
############################################################
####################
# #
# Bash_logout file #
# #
# commands to perform from the bash shell at logout time #
```

# #

##
######################

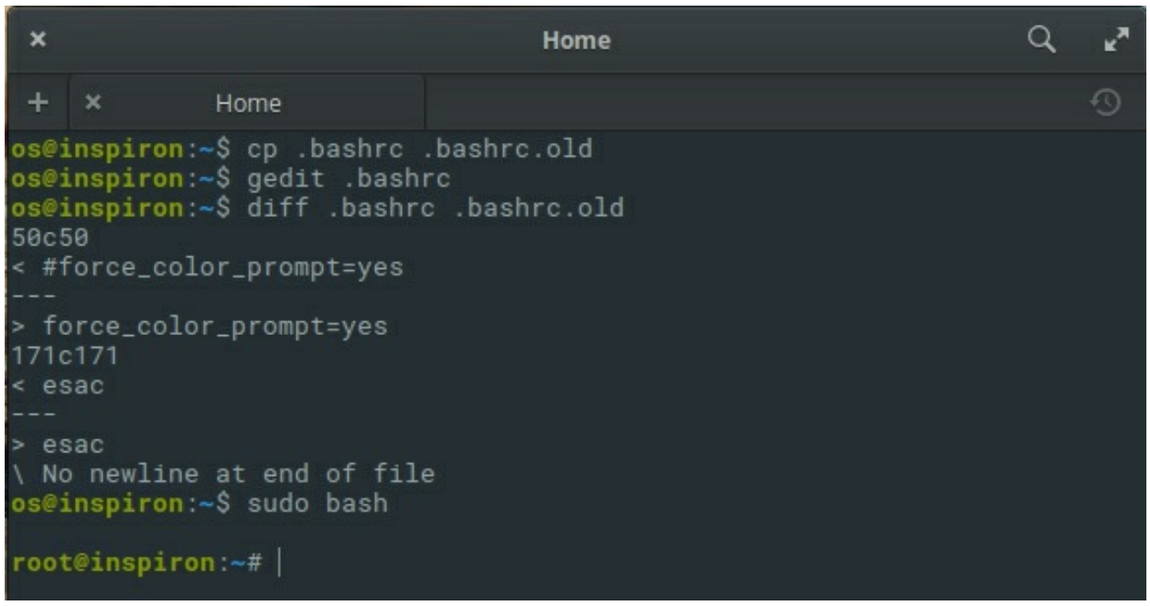
clear

wiki ~>

Changing shell settings files

The user must reconnect to the system when modifying any of the files mentioned above in order to activate these modifications, or as another method that uses the source command on the file he edited. Modifications are applied to the current shell session when interpreting Bash in this way:

Figure 3.1: Different inductors for each user.



Most Bash programs are implemented in special environments where sub-processes do not acquire variables unless the parent shell makes them a source (sourcing). Making a file with shell commands a source is one way to apply modifications to your environment and adjust variables in the current shell.

This example also explains the use of different inductors for each user, and in our case the # sign means that you are now in the root user stage, you should be wary of every command you enter as the device will not hesitate to execute it even if it means destroying files from the system core, Unlike the $ prompt of the average user who has limited authority to execute the commands he or she enters.

Note that the source resourcefile is the same as the resourcefile. .

If you suspect these many settings and see settings you don't know where they originate, use the echo command just as you do to refine Bash programs (see Refine parts of Bash programs). For example, you can add a line like this:

echo "Now executing .bash_profile .."

or

echo "Now setting PS1 in .bashrc:"

export PS1 = "[some value]"

echo "PS1 is now set to $ PS1"

Variables in Bash

Types of variables

Shell variables are capitalized, and Bash maintains a list of two variables:

Global Variables
You will find general variables or environment variables in all shells, and env or printenv commands can be used to display environment variables. These programs come in the sh-utils package. Here's an example of a printenv command output:

wiki ~> printenv

CC = gcc

CDPATH = .: ~: / usr / local: / usr: /

CFLAGS = -O2 -fomit-frame-pointer

COLORTERM = gnome-terminal

CXXFLAGS = -O2 -fomit-frame-pointer

DISPLAY =: 0

DOMAIN = hq.garrels.be

e =

TOR = vi

FCEDIT = vi

FIGNORE = .o: ~

G_BROKEN_FILENAMES = 1

GDK_USE_XFT = 1

GDMSESSION = Default

GNOME_DESKTOP_SESSION_ID = Default

GTK_RC_FILES = / etc / gtk / gtkrc: /nethome/wiki/.gtkrc-1.2-gnome2

GWMCOLOR = darkgreen

GWMTERM = xterm

HISTFILESIZE = 5000

history_control = ignoredups

HISTSIZE = 2000

HOME = / nethome / wiki

HOSTNAME = hsoub.hq.garrels.be

INPUTRC = / etc / inputrc

IRCNAME = wiki

JAVA_HOME = / usr / java / j2sdk1.4.0

LANG = en_US

LDFLAGS = -s

LD_LIBRARY_PATH = / usr / lib / mozilla: / usr / lib / mozilla / plugins

LESSCHARSET = latin1

LESS = -edfMQ

LESSOPEN = | /usr/bin/lesspipe.sh% s

LEX = flex

LOCAL_MACHINE = hsoub

LOGNAME = wiki

LS_COLORS = no = 00: fi = 00: di = 01; 34: ln = 01; 36: pi = 40; 33: so = 01; 35: bd = 40; 33; 01: cd = 40; 33; 01: or = 01, 05,37,41: mi = 01, 05,37,41: ex = 01,32: *. cmd = 01,32: *. exe = 01,32: *. com = 01,32: *. btm = 01,32: *. bat = 01,32: *. sh = 01,32: *. csh = 01,32: *. tar = 01,31: *. tgz = 01; 31: *. arj = 01,31: *. taz = 01,31: *. lzh = 01,31: *. zip = 01,31: *. z = 01,31: *. Z = 01,31: *. gz = 01,31: *. Bz2 = 01; 31: *. Bz = 01,31: *. Tz = 01,31: *. Rpm = 01; 31: *. Cpio = 01; 31: *. Jpg = 01; 35: *. Gif =

01; 35: *. Bmp = 01; 35: *. Xbm = 01,35: *. Xpm = 01; 35: *. Png = 01; 35: *. Tif = 01; 35:

MACHINES = hsoub

MAILCHECK = 60

MAIL = / var / mail / wiki

MANPATH = / usr / man: / usr / share / man /: / usr / local / man: / usr / X11R6 / man

MEAN_MACHINES = hsoub

MOZ_DIST_BIN = / usr / lib / mozilla

MOZILLA_FIVE_HOME = / usr / lib / mozilla

MOZ_PROGRAM = / usr / lib / mozilla / mozilla-bin

MTOOLS_FAT_COMPATIBILITY = 1

MYMALLOC = 0

NNTPPORT = 119

NNTPSERVER = news

NPX_PLUGIN_PATH = / plugin / ns4plugin /: / usr / lib / netscape / plugins

OLDPWD = / nethome / wiki

OS = Linux

PAGER = less

PATH = / nethome / wiki / bin.Linux: / nethome / wiki / bin: / usr / local / bin: / usr / local / sbin: / usr / X11R6 / bin: / usr / bin: / usr / sbin: / bin: / sbin :.

PS1 = \ [\ 033 [1; 44m \] wiki is in \ w \ [\ 033 [0m \]

PS2 = More input>

PWD = / nethome / wiki

SESSION_MANAGER = local / hsoub.hq.garrels.be: /tmp/.ICE-unix/22106

SHELL = / bin / bash

SHELL_LOGIN = - login

SHLVL = 2

SSH_AGENT_PID = 22161

SSH_ASKPASS = / usr / libexec / openssh / gnome-ssh-askpass

SSH_AUTH_SOCK = / tmp / ssh-XXmhQ4fC / agent.22106

START_WM = twm

TERM = xterm

TYPE = type

USERNAME = wiki

USER = wiki

_ = / usr / bin / printenv

VISUAL = vi

WINDOWID = 20971661

XAPPLRESDIR = / nethome / wiki / app-defaults

XAUTHORITY = / nethome / wiki / .Xauthority

XENVIRONMENT = / nethome / wiki / .Xdefaults

XFILESEARCHPATH = / usr / X11R6 / lib / X11 /% L /% T /% N% C% S: / usr / X11R6 / lib / X11 /% l /% T /% N% C% S: / usr / X11R6 / lib / X11 /% T /% N% C% S: / usr / X11R6 / lib / X11 /% L /% T /% N% S: / usr / X11R6 / lib / X11 /% l /% T / % N% S: / usr / X11R6 / lib / X11 /% T /% N% S

XKEYSYMDB = / usr / X11R6 / lib / X11 / XKeysymDB

XMODIFIERS = @ im = none

XTERMID =

XWINHOME = / usr / X11R6

X = X11R6

YACC = bison -y

Local Variables

Local variables will only be found in the current shell, and the set command - without options - can be used to display a list of all functions and variables, including environment variables, and output is categorized according to the current position and is also presented in a reusable format.

Here's an example of a diff file created from comparing the outputs of the printenv and set commands, excluding functions that can also be displayed using the set command:

wiki ~> diff set.sorted printenv.sorted | grep "<" | awk '{print $ 2}'

BASE = / nethome / wiki / .Shell / hq.garrels.be / hsoub.aliases

BASH = / bin / bash

BASH_VERSINFO = ([0] = "2"

BASH_VERSION = '2.05b.0 (1) -release'

COLUMNS = 80

DIRSTACK = ()

DO_FORTUNE =

EUID = 504

GROUPS = ()

HERE = / home / wiki

HISTFILE = / nethome / wiki / .bash_history

HOSTTYPE = i686

IFS = $ '

LINES = 24

MACHTYPE = i686-pc-linux-gnu

OPTERR = 1

OPTIND = 1

OSTYPE = linux-gnu

PIPESTATUS = ([0] = "0")

PPID = 10099

PS4 = '+

PWD_REAL = 'pwd

SHELLOPTS = braceexpand: emacs: hashall: histexpand: history: interactive-comments: monitor

THERE = / home / wiki

UID = 504

Classify variables based on their content

Variables can be categorized according to the type of content the variable holds, and accordingly the variables are divided into four types:

Text variables.

Numerical variables.

Static variables.

Matrix Variables.

You'll find more explanation about these types in more about variables in Bash, but now we'll work with numerical and textual values for our variables.

Create variables
Variable names are case-sensitive, and the variable is capitalized by default, but local variables can be given lowercase names.In any case, you have the freedom to use the names you want or put small and large letters within the name of a single variable. Variables can also contain numbers provided that the variable name does not begin with a number:

prompt> export 1number = 1

bash: export: `1number = 1 ': not a valid identifier

Use the following code to set the value of a variable in the shell:

VARNAME = "value"

Be careful, however, do not put spaces around the quotation marks as this will cause errors, so it's a good idea to usually acquire text content quotes when you assign values to variables, as this will reduce the likelihood of an error. Here are some examples of using case, numbers, and spaces:

wiki ~> MYVAR1 = "2"

wiki ~> echo $ MYVAR1

2

wiki ~> first_name = "Wiki"

wiki ~> echo $ first_name

Wiki

wiki ~> full_name = "Wiki Hsoub"

wiki ~> echo $ full_name

Wiki Hsoub

wiki ~> MYVAR-2 = "2"

bash: MYVAR-2 = 2: command not found

wiki ~> MYVAR1 = "2"

bash: MYVAR1: command not found

wiki ~> MYVAR1 = "2"

bash: 2: command not found

wiki ~> unset MYVAR1 first_name full_name

wiki ~> echo $ MYVAR1 $ first_name $ full_name

<- no output ->

wiki ~>

Export variables

The variable that is generated as seen in the example above is only available in the current shell, it is a local variable, and the sub-processes of the current shell will not recognize this variable.

In order to pass the variables to the sub-shell, we must export them using the export command first. The variables that are exported are referred to as environment variables.

export VARNAME = "value"

The sub-shell can modify the variables you have gained from the parent shell, but these changes will not affect the parent shell. Here's an example that explains:

wiki ~> full_name = "Wiki Hsoub"

wiki ~> bash

wiki ~> echo $ full_name

wiki ~> exit

wiki ~> export full_name

wiki ~> bash

wiki ~> echo $ full_name

Wiki Hsoub

wiki ~> export full_name = "Osama H. Damarany"

wiki ~> echo $ full_name

Osama H. Damarany

wiki ~> exit

wiki ~> echo $ full_name

Wiki Hsoub

wiki ~>

In the example above, we have the full_name variable with a value of Wiki Hsoub, and when we try to read that value in a sub-shell, we get a null answer (the echo command shows a -null string- value).

Now we get out of the sub shell and export the full_name variable in the parent shell, then open a new sub shell and ask for the value of the full_name

variable again, this time we get its value (Wiki Hsoub), because the variable is visible to the sub shell after it is exported.

We can change the value of the variable inside the sub-shell as you see in the example to (Osama H. Damarany), but when we get out of that shell and go back to the mother shell to ask for the value of the variable itself, we get the first value (Wiki Hsoub), did not affect its value in the mother shell By changing it inside a sub-shell.

The difference between * $ and @ $

Using * $ can cause bugs or vulnerabilities in your programs, and almost every time a programmer uses * $, it means @ $.

Topical operators are words that follow the name of the shell program. These parameters are placed in the variables $ 1, $ 2, $ 3, and so on. The variables are added to an internal array as long as needed.

```
#! / bin / bash

# positional.sh
# This script reads 3 positional parameters and prints them out.

POSPAR1 = "$ 1"
POSPAR2 = "$ 2"
POSPAR3 = "$ 3"

echo "$ 1 is the first positional parameter, \ $ 1."
echo "$ 2 is the second positional parameter, \ $ 2."
echo "$ 3 is the third positional parameter, \ $ 3."
echo
echo "The total number of positional parameters is $ #."
```

After execution you can add any number of arguments:

```
wiki ~> positional.sh one two three four five
one is the first positional parameter, $ 1.
two is the second positional parameter, $ 2.
```

three is the third positional parameter, $ 3.

The total number of positional parameters is 5.

wiki ~> positional.sh one two

one is the first positional parameter, $ 1.

two is the second positional parameter, $ 2.

is the third positional parameter, $ 3.

The total number of positional parameters is 2.

See Evaluate these parameters in conditional structures in Bash, and use the embedded Shift command in Bash. Here are some examples of other special parameters:

wiki ~> grep dictionary / usr / share / dict / words

dictionary

wiki ~> echo $ _

/ usr / share / dict / words

wiki ~> echo $$

10662

wiki ~> mozilla &

[1] 11064

wiki ~> echo $!

11064

wiki ~> echo $ 0

bash

wiki ~> echo $?

0

wiki ~> ls doesnotexist

ls: doesnotexist: No such file or directory

wiki ~> echo $?

1

wiki ~>

In the previous example, the user wiki starts by entering a grep command that results in a value of _, and the process ID is 10662. It retains the process ID in the background, and the shell running here is Bash, and when something goes wrong? Keeps an exit code different from 0 (zero).

Recycle shell programs using variables

Variables help you apply shell scripts in other environments or for other purposes, but they make the program easier to read. See the following example for a simple program that takes a backup of a user's home wiki folder on a remote server:

#! / bin / bash

 # This program copies my home folder

cd / home

 # This command creates the archive

tar cf /var/tmp/home_wiki.tar wiki> / dev / null 2> & 1

 # Delete the old bzip2 file

#Redirect errors because this produces some errors if the archive does not exist, and then create a new zip file.

rm /var/tmp/home_wiki.tar.bz2 2> / dev / null

bzip2 /var/tmp/home_wiki.tar

 # Copy the file to another host - we have ssh keys to complete it without interference.

scp /var/tmp/home_franky.tar.bz2 bordeaux: / opt / backup / franky> / dev / null 2> & 1

 # Create a time stamp in the log file.

date >> /home/wiki/log/home_backup.log

echo backup succeeded >> /home/wiki/log/home_backup.log

First, you'll make mistakes if you manually rename files and folders every time you need to. Second, suppose the wiki wants to give that program to osama, so osama will need to make some modifications before he can use it to create a backup of his home folder. You can say like this if the wiki wants to use that program to back up other folders. To make recycling easier, make all files, journals, usernames, server names, etc. variable, so you only need to modify the value once without having to check the entire program to see

where a lab is.

#! / bin / bash

 # This program creates a backup copy of my home folder.

 # Change the values of the variables for your program to work:

BACKUPDIR = / home

BACKUPFILES = wiki

TARFILE = / var / tmp / home_wiki.tar

BZIPFILE = / var / tmp / home_wiki.tar.bz2

SERVER = bordeaux

REMOTEDIR = / opt / backup / wiki

LOGFILE = / home / wiki / log / home_backup.log

cd $ BACKUPDIR

 # This creates the archive

tar cf $ TARFILE $ BACKUPFILES> / dev / null 2> & 1

 # Delete the old bzip2 file

#Redirect errors because this produces some errors if the archive does not exist, and then create a new zip file.

rm $ BZIPFILE 2> / dev / null

bzip2 $ TARFILE

 # Copy the file to another host - we have ssh keys to complete it without interference.

scp $ BZIPFILE $ SERVER: $ REMOTEDIR> / dev / null 2> & 1

 # Create a timestamp in a logfile.

date >> $ LOGFILE

echo backup succeeded >> $ LOGFILE

Large folders and low speed internet capacity

Anyone can accommodate the example above where we used a small and host folder on the same subnet.However, the speed of doing this depends on

your folder size, internet speed, and server location, it can take a long time to create backups using this method.

The solution that works for large folders is to use rsync to keep folders in both respects synchronized.

Quote characters in Bash

Utility quote typefaces

Many quotes and words have special meanings and functions in one form or another, and the quotation is used to eliminate that particular meaning or function.The quotation marks in Bash disrupt the treatment given to the special character, and prevent the treatment of special words according to the meanings they carry, as well as the expansion of coefficients.

Character escape

Escape characters are used to cancel the special meaning of a single character, and the backslash is used as an escape character in Bash. It retains the immediate meaning of the next character except for the new line character, if a new line character follows the slash it specifies the completion of the current line if it is longer than the width The slash is removed from the input and ignored.

In the following example, we create a variable named date and set it to a value. The echo command displays that value when the date variable is invoked. The second time we use the \ character, the $ tag is ignored and the echo command prints the following as plain text:

wiki ~> date = 20021226

wiki ~> echo $ date

20021226

wiki ~> echo \ $ date

$ date

Single quotation marks

Single quotation marks " are used to retain the literal value of each character within the quotation marks, and a single quotation mark does not come between two single quotation marks even if it is preceded by a slash. We follow with the previous example:

hsoub ~> echo '$ date'

$ date

Double quotation marks

Double quotes are used to preserve literal values for all characters that fall between them except the $ sign, single back quotes, and backslashes.

The $ dollar sign and single back quotes `` retain their own functions within double quotes.

The backslash does not retain its own function within quotation marks unless it is followed by a $, `," or \ or new line, and the backslashes are omitted from the input if one of them is followed. They are left unchanged to be treated by the interpreter.

Two double quotation marks can also be used within two other double tags, see the following example for further clarification:

hsoub ~> echo "$ date"

20021226

hsoub ~> echo "` date``

Sun Apr 20 11:22:06 CEST 2003

hsoub ~> echo "I'd say: \" Go for it! \ ""

I'd say: "Go for it!"

hsoub ~> echo "\"

More input> "

hsoub ~> echo "\\"

\

Quote ANSI-C

Words in the " STRING '$' format are specially treated. The word is expanded to text (string), replacing the characters ignored by the slash as defined in the ANSI-C criteria. See the Bash documentation for more detail on backward slash escape sequences.

Locales

When a text is preceded by a double quotation mark with a $ sign, it is translated according to Locale, and the translated text is placed between the double quotation marks as well. If the locale is C or POSIX, the $ sign is ignored.

Expansions in Bash

Expanded swastika {}

The expansion of the square bracket is a mechanism by which arbitrary strings can be created. The patterns on which the expansion of the square bracket will be carried out are optional preamble followed by a series of texts separated by commas. {} Next comes an optional postscript as well. Initiation is placed at the beginning of each text within the bracket - after expansion - and then the footnote is also attached to the end of each text, in order from left to right.

Swastika extensions may overlap but that does not mean that the results can be sorted, but the order remains from left to right, see the following example for illustration:

hsoub ~> echo sp {el, il, al} l

spell spill spall

The expansion of the square bracket is carried out before any other expansion, and the result is retained any characters with special meanings in relation to other expansions, since this expansion is exclusively textual. Also, the text} $ is not valid for expanding the square bracket in order to avoid inconsistencies.

The expansion of the square bracket must contain square brackets without quotation marks around each other and a comma, at least one not in quotation marks, and any expansion of the square bracket, unlike that formula, is left unchanged.

Length Expansion (~)

If a word begins with a period ~ without quotation marks, all characters up to the first slash / without quotation marks - or all characters if there is no slash without quotation marks - are considered an introduction to the term (tilde-prefix). If tilde-prefix is a login name, if no prefix characters are enclosed in quotation marks, if that login name is blank, the duration is replaced by the value of the HOME variable, if the HOME variable is not set to any value, The home folder of the user who performs the shell is used in its place, otherwise the foreground is replaced by the For a home associated with the login name.

If the foreground is + ~, the value of the PWD variable is superseded. If the characters after the period in the foreground consist of a number N (may) preceded by + or -, the foreground is replaced by the corresponding element from the Directory Stack, and will be displayed as an argument in the dirs command called by the characters after the period in the foreground If the characters following the period consist of a number without a positive or negative sign before it, this number is assumed to be positive. Also, if the login name is not valid, or if the extension fails, the word is left unchanged.

Variable assignments are examined after: or = directly for prefixes that are not in quotation marks, and the duration is expanded in those cases, so file names with the duration character can be used in assignments that have variables for PATH, MAILPATH, and CDPATH, and coincidence sets the extended value.

Example:

hsoub ~> export PATH = "$ PATH: ~ / testdir"

Testdir / ~ will be expanded to $ HOME / testdir, so if $ HOME is var / home / hsoub /, the var / home / hsoub / testdir / folder is added to the contents of the PATH variable.

Coefficient of chance and variable expansion

The $ character is preceded by parameter expansion, command substitution and arithmetic expansion. The name of the parameter that is to be expanded, or its symbol, may be in square brackets and is optional, but it protects the variable that will be expanded from the characters that follow. It is part of the name. When square brackets are used, the second end of the arch is the first {not smuggled by a backslash \, between quoted text, within an embedded arithmetic expansion, a substituted command, or a coefficient expansion.

The basic image of the parameter expansion is $ {PARAMETER}, the value of PARAMETER is replaced, and the parentheses are necessary when PARAMETER is a local operand of more than one number or when PARAMETER is followed by a character that will not be interpreted as part of the name.

If the first parameter of the parameter PARAMETER is an exclamation point,

Bash uses the value of the variable consisting of the rest of the PARAMETER as the name of the variable.This variable is expanded and used in the rest of the substitution instead of the value of PARAMETER itself.This is known as indirect expansion, unlike the widespread direct expansion that occurs In the simplest case, such as the previous case or this case:

hsoub ~> echo $ SHELL

/ bin / bash

The following example is for indirect expansion:

hsoub ~> echo $ {! N *}

NNTPPORT NNTPSERVER NPX_PLUGIN_PATH

Note that this is different from * echo $ N, the {VAR: = value} $ structure allows you to create the variable mentioned if it is not already created, see the following example for illustration:

hsoub ~> echo $ HSOUB

hsoub ~> echo $ {HSOUB: = Hsoub}

Hsoub

Special transactions, among other localized transactions, may not be assigned in this way. We will discuss the use of square brackets in the treatment of variables.

Substitution of commands

Substitution of commands allows the output of a command to replace the command itself, and the substitution of commands occurs when an order is on this image: (command) $ or on this image using the apostrophe `:` command`

Bash performs the expansion by executing the command and replacing the substitution with the normal output of the command with the deletion of any new lines (trailing newlines). The new embedded lines are not deleted but may be deleted while words are split.

hsoub ~> echo `date`

Thu Feb 6 10:06:20 CET 2003

The backslash retains its literal meaning when using the old backquoted form of substitution, unless it is followed by a $ or `or` sign, and the substitution of the first backslash is not preceded by a backslash \ (backslash), when using the $ COMMAND formula, all characters in parentheses are made up of the command and none are treated specifically.

Command overrides can also overlap, and in order to create overlap while using the back quote formula, precede the slashes with back slashes. If the substitution appears in double quotation marks, and the file names expand on the results.

Computational expansion

The arithmetic expansion allows the evaluation and arithmetic of an arithmetic expression, and the formula for arithmetic expansion is $ (arithmetic expression). All units within the expression are subject to coefficient expansion, order substitution and citation removal, and computational substitutions can overlap.

Arithmetic expressions are evaluated in fixed-width integers without losing the surplus, although dividing by zero is picked up and defined as an error, and operators are almost the same as in C programming language.

Process substitution

Process substitution is supported in systems that support named pipes (FIFOs- First In First Out) or the dev / fd / file naming method, with the format: (LIST)> or (LIST) <.

The LIST process is carried out so that its inputs and outputs are connected to a named pipe (FIFO) or a file in dev / fd /, and the file name is passed as an intermediary to the current command as a result of the expansion. If LIST is used, writing to the file will provide the LIST with inputs. If LIST is used, the file passed as an intermediary must be read in order to obtain the LIST output. Note that spaces should not appear between the <or> tags and the left parenthesis, otherwise the structure will be interpreted as a redirect.

Process substitution is performed in conjunction with coefficient and variable expansion, command substitution and computational expansion whenever possible. See Forwarding and file descriptions.

Split words

The coincidence clears the results of transaction expansion, order substitution, and arithmetic expansions that do not fall within double quotes in search of a split of words. The shell treats each internal field separator or IFS $ character as a delimiter, and divides the results of the other expansions

into words about those characters. > <Table (tab)> <new line> " Specifically, any sequence of IFS characters works to identify words.

The sequence of white space characters and the table (tab) at the beginning and end of a word is ignored if the IFS value is other than the default value, as long as the white space character is in the IFS value (white space character of the IFS variable).

Any character in the IFS variable is not a white space character, it specifies a field, as well as any white space characters in any nearby IF (IF- Internal Field), and the sequence of white space characters of the IFS separator is also treated as a delimiter.If the IFS value is empty, no word division occurs.

Explicit blank arguments (" " or " ") are retained, and the non-quoted implicit arguments resulting from the expansion of non-value coefficients are deleted, and empty text is produced and retained when a coefficient with no quotation marks is expanded.

Expansion and division of words

If there is no expansion, no division will be implemented.

Expands file names

Bash clears each word after the word has been split unless the f- option is checked for characters * and? F) (see refinement of parts of the shell), if none of these characters appear, the word is a PATTERN pattern and is replaced by a list of file names corresponding to the pattern alphabetically.

If no matching name exists and the nullglob shell option is disabled, the word is left intact. If the nullglob option is set and there are no matches, the word is deleted. If the nocaseglob option is enabled, the match is executed without regard to the case of the alphabet.

When a pattern is used to generate a filename, it is typeface. At the beginning of the filename or immediately after a slash, it must be explicitly matched unless the dotglob option is selected. When matching a file name, the slash must always match explicitly. May not be treated. Particularly in some cases.

The GLOBIGNORE variable may be used to restrict the set of file names that match a pattern, even if GLOBEIGNORE is set to a value that is omitted from the list of matching file names. Each name matches a style in GLOBIGNORE.

They are ignored. And .. as filenames even if GLOBIGNORE is set, but in any case, setting GLOBIGNORE activates the dotglob option, so all other filenames beginning with. , And to get old behavior to ignore file names that begin with. , Make *. One of the styles in GLOBIGNORE. The dotglob option is disabled when GLOBEIGNORE is unset.

Alternative commands in Bash

Utility alternative commands

The Alias command allows a single word to be placed in place of a string, if used as the first word of a simple command. The shell keeps a list of alternate commands that can be set and desensitized by the alias and unalias commands. Type alias at the command line to see a list of alternate commands defined for the current shell:

hsoub: ~> alias

alias .. = 'cd ..'

alias ... = 'cd ../ ..'

alias .... = 'cd ../../ ..'

alias PAGER = 'less -r'

alias Txterm = 'export TERM = xterm'

alias XARGS = 'xargs -r'

alias cdrecord = 'cdrecord -dev 0,0,0 -speed = 8'

alias e = 'vi'

alias egrep = 'grep -E'

alias ewformat = 'fdformat -n / dev / fd0u1743; ewfsck '

alias fgrep = 'grep -F'

alias ftp = 'ncftp -d15'

alias h = 'history 10'

alias fformat = 'fdformat / dev / fd0H1440'

alias j = 'jobs -l'

alias ksane = 'setterm -reset'

alias ls = 'ls -F --color = auto'

alias m = 'less'

alias md = 'mkdir'

alias od = 'od -Ax -ta -txC'

alias p = 'pstree -p'

alias ping = 'ping -vc1'

alias sb = 'ssh blubber'

alias sl = 'ls'

alias ss = 'ssh octarine'

alias tar = 'gtar'

alias tmp = 'cd / tmp'

alias unaliasall = 'unalias -a'

alias vi = 'eval `resize`; vi'

alias vt100 = 'export TERM = vt100'

alias which = 'type'

alias xt = 'xterm -bg black -fg white &'

hsoub ~>

Alternate commands are used to determine the default version of an order on your system in more than one version, or to specify the default options for an order, and can be used to correct misspellings. The first word of each simple command is examined if it is not quoted - not in quotation marks - for an alternate command. The substitute command name and replacement text may contain shell entries such as shell metacharacters, except that the substitute command name may contain =.

The first word of the substitution text is also scanned for an alternative command, but the word that matches an alternate command expands as it is read again. recursively). The next word for the alternate command is checked inside the command text for an expansion of an alternate command if the last character of the alternate order value is a space character or a table (tab). Alternate commands are not extended if the shell is not interactive unless the expand-aliases option is set using the shopt command.

Create and delete alternate commands

Alternate commands are created using the alias command. To create an alternate command permanently, enter alias into one of your shell initialization files. If you have entered the alternative command on the command line, it will only be recognized within the current shell.

hsoub ~> alias dh = 'df -h'

hsoub ~> dh

Filesystem Size Used Avail Use% Mounted on

/ dev / hda7 1.3G 272M 1018M 22% /

/ dev / hda1 121M 9.4M 105M 9% / boot

/ dev / hda2 13G 8.7G 3.7G 70% / home

/ dev / hda3 13G 5.3G 7.1G 43% / opt

none 243M 0 243M 0% / dev / shm

/ dev / hda6 3.9G 3.2G 572M 85% / usr

/ dev / hda5 5.2G 4.3G 725M 86% / var

hsoub ~> unalias dh

hsoub ~> dh

bash: dh: command not found

hsoub ~>

Bash reads at least one full line of inputs before executing any commands on that line. Alternative commands are only expanded when a command is read and not executed, so defining an alternate command that appears on the same line as another does not have effect until the next line of Input. Commands that follow the alternative command definition on that line are not affected by the new alternative command. This behavior is also a problem when you perform functions. Alternative commands are expanded when the definition of a new function is read and not executed, since the function itself is a compound command. As a result, alternative commands specified in a function are not available even after the function is executed. But to be safe, place the alternate command settings on a separate line and do not use the alias command in bulk commands. Also, sub-processes do not inherit substitute commands, and the Bourne shell does not recognize alternative commands.

Faster functions

Alternative commands are sought after functions, so their resolution is slower, although easier to understand, and prefers the shell functions over alternative commands in almost every use.

Chapter III

Stereotypes in Bash

This chapter highlights the features included in Bash to match patterns and identify character classes and ranges. In addition to a breakdown of the regular expressions (Regular Expressions), they are powerful tools for selecting specific lines of files or from an output, and are used by many commands in Unix such as vim, perl, PotgreSQL database and others, and can be added in any language or application using external libraries, You may even find it on non-Unix systems, as Excel uses it for tables that come in the Windows Office package.

You will also learn about grep, which is indispensable in any Unix environment, as it has far more possibilities than we will explain in this

chapter where we use it as an example of stereotypes only.

Regular Expressions

Regular Expression is a method that describes a set of strings. These stereotypes are constructed analogously to arithmetic expressions using several coefficients to combine smaller expressions. The smallest modular structure is the one that matches a single character.Most of the characters, including all letters and numbers, are only typical expressions that match themselves, and any special character (metacharacter) that has a special meaning can be quoted with a backslash.

The difference between basic and extended stereotypes

The following special characters lose their meaning in Basic Regular Expression: , +, {, | , (,), And use the same characters prefixed with a backslash instead? \, + \, {\, | \, (\,) \. See your system documentation to see which commands use stereotypes that support extended expressions.

Grep command

The grep command searches input files for lines that contain matches for a particular pattern list.When it finds a match in a line, it copies the line to the standard output by default or to any other output that you require using the options that you may add to the command.

Although the grep command expects to match text, it has no restrictions on the length of the input line except for available memory, and can match arbitrary characters inside the line, although the last byte of the input file is not a newline, grep adds Automatically. Also, the characters of the new line within text cannot be matched since the new line is a separator of the style list. Here are some examples:

hsoub ~> grep root / etc / passwd

root: x: 0: 0: root: / root: / bin / bash

operator: x: 11: 0: operator: / root: / sbin / nologin

hsoub ~> grep -n root / etc / passwd

1: root: x: 0: 0: root: / root: / bin / bash

12: operator: x: 11: 0: operator: / root: / sbin / nologin

hsoub ~> grep -v bash / etc / passwd | grep -v nologin

sync: x: 5: 0: sync: / sbin: / bin / sync

shutdown: x: 6: 0: shutdown: / sbin: / sbin / shutdown

halt: x: 7: 0: halt: / sbin: / sbin / halt

news: x: 9: 13: news: / var / spool / news:

mailnull: x: 47: 47 :: / var / spool / mqueue: / dev / null

xfs: x: 43: 43: X Font Server: / etc / X11 / fs: / bin / false

rpc: x: 32: 32: Portmapper RPC user: /: / bin / false

nscd: x: 28: 28: NSCD Daemon: /: / bin / false

named: x: 25: 25: Named: / var / named: / bin / false

squid: x: 23: 23 :: / var / spool / squid: / dev / null

ldap: x: 55: 55: LDAP User: / var / lib / ldap: / bin / false

apache: x: 48: 48: Apache: / var / www: / bin / false

hsoub ~> grep -c false / etc / passwd

7

hsoub ~> grep -i ps ~ / .bash * | grep -v history

/home/hsoub/.bashrc:PS1="\[\033[1;44m\font>$USER is in \ w \ [\ 033 [0m \] ""

The hsoub user displays lines that contain the root text of / etc / passwd.

It then displays the line numbers that contain that text.

Then look at the third command which users do not use bash, but accounts that use nologin shell are not displayed.

It then calculates the number of calculations encountered by bin / false /.

The last command line displays lines beginning with bash./~ from all files in the user's home folder hsoub, except for matches containing history text, including excluding matches from the bash_history./~ file since it may contain the same section Script both in its lower case (lowercase) and major case (uppercase). Also note that searching for ps text, not ps.

Grep command and typical expressions

If you are on an operating system other than Linux

We use the grep command for GNU in these examples, which supports extended stereotypes. The grep command from GNU is the default on Linux, see if you are running on non-Linux systems, which version you have using the V- option. You can also download the GNU grep from https://gnu.org/directory.

Axes of lines and words

From the previous example, we will now show the lines beginning with root text:

hsoub ~> grep ^ root / etc / passwd

root: x: 0: 0: root: / root: / bin / bash

If we want to see any calculations for which shells are not assigned, we will look for lines that end with the letter:, as follows:

hsoub ~> grep: $ / etc / passwd

news: x: 9: 13: news: / var / spool / news:

To find out if the PATH variable has been exported in the bashrc./~ file, first choose export lines and then look for lines that start with PATH text so that MANPATH and other possible paths are not displayed:

hsoub ~> grep export ~ / .bashrc | grep '\ <PATH'

export PATH = "/ bin: / usr / lib / mh: / lib: / usr / bin: / usr / local / bin: / usr / ucb: / usr / dbin: $ PATH"

Similarly, <\ matches the end of a word, and if you want a separate word "any text between two spaces" it is best to use w- as in the following example where it displays information about the root partition -root- on the hard drive:

hsoub ~> grep -w / / etc / fstab

LABEL = // ext3 defaults 1 1

If this option is not used, all lines of the file system table will be displayed.

Character Types

The bracket expression is a list of characters between the square brackets [], which corresponds to any single character in that list.If the first character in the list is a caret ^ it matches any character that is not in the list. [0123456789] Any single typeface. The expression range within a square bracket [] consists of two characters separated by a dash - and corresponds to any single character that is classified between two characters including the use of locale's collating sequence and character set. For example, in the default locale for C, [a-d] is equal to [abcd].

Many locales classify characters in the dictionary order, and in those localities [a-d] is not equal to [abcd]; it may equal [aBbCcDd]. For the traditional interpretation of square bracket expressions, you can use the locale for C by setting the environment variable LC_ALL to C.

Finally, some of the character categories listed are predefined within square bracket expressions, see the grep info pages or man directory pages for more information on those predefined expressions. Also, look at the following example where all lines containing either y or f will be displayed.

hsoub ~> grep [yf] / etc / group

sys: x: 3: root, bin, adm

tty: x: 5:

mail: x: 12: mail, postfix

ftp: x: 50:

nobody: x: 99:

floppy: x: 19:

xfs: x: 43:

nfsnobody: x: 65534:

postfix: x: 89:

Wildcards

use . In order to match a single character, if you want to get a list of all five-letter English dictionary words that start with c and end with h, here's an example - useful in solving crosswords! -:

hsoub ~> grep '\ <c ... h \>' / usr / share / dict / words

catch

clash

cloth

coach

couch

cough

crash

crush

If you want to display lines that contain a typeface. Use the F-option to get it, and use the * character to match multiple characters, see the following example, which selects all words beginning with c and ending with h from the system dictionary:

hsoub ~> grep '\ <c. * h \>' / usr / share / dict / words

caliph

cash

catch

cheesecloth

cheetah

--output omitted--

If you want to find a * character inside a file or an output, use single quotes around it. In the following example, it attempts to find the star character in etc / profile / without the quotation marks, it does not return any lines, but when used, it gets a result of what it wants in the output. :

hsoub ~> grep * / etc / profile

hsoub ~> grep '*' / etc / profile

for i in /etc/profile.d/*.sh; do

Match patterns using Bash features

Character Ranges

Unlike grep and regular expressions, we have many instances of pattern matching that you can make directly in the shell without having to use an external program. Match any single text or character, respectively, and in order to match these two characters specifically, enclose double quotation marks around each:

hsoub ~> touch "*"

hsoub ~> ls "*"

*****

You can use square brackets to match any character or character set within those brackets if the character pairs are separated by a dash - see the following example, which shows all files beginning with one of the following characters: a, b, c, x, y, z in the user's home folder hsoub :

hsoub ~> ls -ld [a-cx-z] *

drwxr-xr-x 2 hsoub hsoub 4096 Jul 20 2002 app-defaults /

drwxrwxr-x 4 hsoub hsoub 4096 May 25 2002 arabic /

drwxrwxr-x 2 hsoub hsoub 4096 Mar 4 18:30 bin /

drwxr-xr-x 7 hsoub hsoub 4096 Sep 2 2001 crossover /

drwxrwxr-x 3 hsoub hsoub 4096 Mar 22 2002 xml /

If the first typeface in parentheses is! Or ^ it will match any character that is not in parentheses, and to match the dash -, enter it as the first or last character in the set, and the classification depends on the current locale and the value of the LC_COLLATE variable, if set to a value, note that other locale may be interpreted [a-cx-z] as [aBbCcXxYyZz] If the classification is in the dictionary order, and if you want to make sure the range is interpreted in the traditional way, set LC_COLLATE or LC_ALL to C.

Character Types

Character classes can be specified in square brackets according to the linguistic structure [: CLASS:] where CLASS is defined within the POSIX standard and has one of the following values: alnum - alpha - ascii - blank - cntrl - digit - graph - lower - print - punct - space - upper - word - xdigit.

Examples:

hsoub ~> ls -ld [[: digit:]] *

drwxrwxr-x 2 hsoub hsoub 4096 Apr 20 13:45 2 /

cathy ~> ls -ld [[: upper:]] *

drwxrwxr-- 3 hsoub hsoub 4096 Sep 30 2001 Nautilus /

drwxrwxr-x 4 hsoub hsoub 4096 Jul 11 2002 OpenOffice.org1.0 /

-rw-rw-r-- 1 hsoub hsoub 997376 Apr 18 15:39 Schedule.sdc

Several extended pattern matching operators are defined when the extglob option is activated using the shopt command, see the info directory pages, basic shell features section> shell expansions> filename expansion> Pattern

matching.

Chapter IV

Conditional structures in Bash

In this chapter, we will learn how to build conditional strings in our programs so that we can take different actions based on the success or failure of commands.These procedures will be determined using the if statement.This will allow you to perform mathematical and textual comparisons, and to test the exit codes, inputs, and files needed by the program.

Commands in shell programs are usually preceded by an if / then / fi test to prevent the output from being created for the program to run in the background or through the cron tool, and more complex conditions are placed in the case statement.

The program tells the parent shell when the condition test is successful using the exit 0 state

Another number, and the program in the mother shell then performs the appropriate actions according to the return code.

- **Introduction to if in Bash**

Look at the conditional if statement, as well as then, the expressions and commands used with each, as well as checking files and shell options.

- **Advanced uses of the conditional if statement**

Detailed explanations of structures using if / then / else and if / then / elif / else.

- **Use the case structure in Bash**

A simple explanation of cases where case is used instead of if, with an example from an init code.

D You need to specify different strings of action within shell scripts based on the success or failure of the execution of a command, the conditional if tool comes for such cases, and the smallest syntax for the if command is the following:

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

The TEST-COMMANDS test command list is executed, and the CONSEQUENT-COMMANDS dependent command list is executed if its return status is zero. The return status is the exit status of the last executed order, or zero if no test order is met.

The TEST-COMMAND test command often involves numerical or textual comparisons, but this does not prevent a command that returns a zero state if it succeeds and another case when it fails. Unary Expressions are often used to test the status of a file. FILE for a primaries in dev / fd / N format. The file descriptor N is inspected. Stdin, stdout, and stderr files and their file descriptions, respectively, may also be used in these tests.

Expressions used with if

The following table contains a summary of the so-called basics that make up the TEST-COMMAND test order or the list of test commands in the conditional if structure. These basics are enclosed in square brackets to show the conditional expression test.

The [] test command evaluates conditional expressions using a set of rules based on the number of arguments. You will find more explanation of this in the Bash documentation. Also, the square bracket should be closed after the conditions are listed just as if the conditional should be closed with fi.

The commands that follow are then

CONSEQUENT-COMMANDS commands following the then statement can be any UNIX command, executable program, shell script, or shell statement, except of course the fi statement that is closed. Perhaps you should remember that then and fi are separate phrases in chance, so they are separated by a semicolon; When you type them in the command line. Also, the different parts of the if statement within a shell program are well separated, you'll find some examples below.

Scan files

The first example looks at the existence of the msgcheck.sh file:

hsoub ~> cat msgcheck.sh

#! / bin / bash

echo "This scripts checks the existence of the messages file."

echo "Checking ..."

if [-f / var / log / messages]

then

echo "/ var / log / messages exists."

fi

echo

echo "... done."

hsoub ~> ./msgcheck.sh

This scripts checks the existence of the messages file.

Checking ...

/ var / log / messages exists.

... done.

Check the chance options

To add your Bash settings files:

noclobber These lines will print a message if an option is set:

if [-o noclobber]

then

echo "Your files are protected against accidental overwriting using redirection."

fi

The environment

The above example will be executed when entered in the command line:

hsoub ~> if [-o noclobber]; then echo; echo "your files are protected against overwriting. "; echo; fi

your files are protected against overwriting.

hsoub ~>

However, if you use environment-based testing, you may get different results when you insert the same into a shell script. The program will open a new shell in which variables and options may not be automatically set.

Simple applications on the if statement

Test the exit status

Contains a variable? On the logoff status of the previously executed command (last operation in the interface). The following example shows a simple test:

hsoub ~> if [$? -eq 0]

More input> then echo 'That was a good job!'

More input> fi

That was a good job!

hsoub ~>

This example shows how TEST-COMMAND could be any Unix command that returns an exit state, and if if again returns a zero exit state:

hsoub ~> if! grep $ USER / etc / passwd

More input> then echo "your user account is not managed locally"; fi

your user account is not managed locally

hsoub> echo $?

0

hsoub>

The same result can be obtained as follows:

hsoub> grep $ USER / etc / passwd

hsoub> if [$? -ne 0]; then echo "not a local account"; fi

not a local account

hsoub>

Numerical comparisons

The examples below use numerical comparisons:

hsoub> num = `wc -l work.txt`

hsoub> echo $ num

201

hsoub> if ["$ num" -gt "150"]

More input> then echo; echo "you've worked hard enough for today."

More input> echo; fi

you've worked hard enough for today.

hsoub>

This code is executed with a cron every Sunday and sends you a reminder to remove trash cans even if the week number is even:

#! / bin / bash

 # date Calculate the week number using a command:

WEEKOFFSET = $ [$ (date + "% V")% 2]

 # Consider the conditional statement If we have a reminder, if not, the week will be even, send us a message

 # In case of any other result, the statement does nothing.

if [$ WEEKOFFSET -eq "0"]; then

echo "Sunday evening, put out the garbage cans." | mail -s "Garbage cans out" your@your_domain.org

fi

Text comparisons

Here's an example of text comparison to test your User ID:

if ["$ (whoami)"! = 'root']; then

echo "You have no permission to run $ 0 as non-root user."

exit 1;

fi

You can shorten this type of structure using Bash, see the following example that represents a compressed image from the previous test:

["$ (whoami)"! = 'root'] && (echo you are using a non-privileged account; exit 1)

Specifies the || What to do if the test is wrong, just like the && expression that explains what to do if the test is correct. Also, stereotypes can be used in

comparisons:

hsoub> gender = "male"

hsoub> if [["$ gender" == m *]]

More input> then echo "Pleasure to meet you, Sir."; fi

Pleasure to meet you, Sir.

Real programmers

Most programmers prefer to use the test command, which is equal to the use of square brackets for comparison. See the following example:

test "$ (whoami)"! = 'root' && (echo you are using a non-privileged account; exit 1)

Exit command

If you call the exit command in a sub-shell, it won't pass variables to the parent shell, use {and} instead of (and) if you don't want Bash to open a sub-shell.

If / then / else is built

The following example illustrates the structure that should be used to take an action or series of actions if the conditions of the if statement are met, and a series of other actions if not:

hsoub scripts> gender = "male"

hsoub scripts> if [["$ gender" == "f *"]]

More input> then echo "Pleasure to meet you, Madame."

More input> else echo "How come the lady hasn't got a drink yet?"

More input> fi

How come the lady hasn't got a drink yet?

hsoub scripts>

The difference between [] and [[]]

Unlike [],]] prevents words from being split into variable values, so if "VAR =" var with spaces you don't need to put $ VAR between double quotes in a test, although using quotes remains a better behavior anyway. It also prevents]] pathname expansion, so do not attempt to expand wildcards to file names. Also, using]], == and =! Interpret the sections on the right as glob patterns that are compared to the values on the left. For example, [[* value "== val"]].

The ALTERNATE-CONSEQUENT-COMMANDS alternative command list that follows the else statement may consist of any UNIX command that returns an exit state, just like the list of CONSEQUENT-COMMANDS dependent commands that follows the then statement. Here's another example that complements the example described in the Exit-to-If Exit chapter in Bash:

hsoub ~> su -

Password:

[root @ alaraby root] # if! grep ^ $ USER / etc / passwd 1> / dev / null

> then echo "your user account is not managed locally"

> else echo "your account is managed from the local / etc / passwd file"

> fi

your account is managed from the local / etc / passwd file

[root @ alaraby root] #

In the previous example, we changed the user to the root user to see the effect of else, because your root user is a local user, unlike your user account, as it may be subject to a centralized system, such as an LDAP server.

Examine the command line arguments
It is more appropriate to place values for variables on the command line rather than setting the variable to a value and then execute the program. We use the position parameters of $ 1, $ 2,…, N $ for this purpose, where # $ indicates the number of command-line arguments, and $ 0 indicates the name of the shell program . Here's a simple example:

```
hsoub @ alaraby: ~ / testdir $ cat penguin.sh
#! / bin / bash

# (Tux) This program allows you to serve food to the penguin
# The penguin will only be happy when a fish is given

if ["$ 1" == fish]; then
  echo "Hmmmm fish ... Tux is happy!"
else
  echo "Tux doesn't like that. Tux wants fish!"
fi

hsoub @ alaraby: ~ / testdir $ penguin.sh apple
Tux doesn't like that. Tux wants fish!
hsoub @ alaraby: ~ / testdir $ penguin.sh fish
Hmmm fish ... Tux is happy!
hsoub @ alaraby: ~ / testdir $
```

Here's another example that uses two local operators:

```
hsoub ~> cat weight.sh
#! / bin / bash

# This program prints a message about your weight if you give your weight in kilograms and your height in centimeters

weight = "$ 1"
height = "$ 2"
idealweight = $ [$ height - 110]

if [$ weight -le $ idealweight]; then
  echo "You should eat a bit more fat."
else
  echo "You should eat a bit more fruit."
fi

hsoub ~> bash -x weight.sh 55 169
+ weight = 55
+ height = 169
+ idealweight = 59
```

+ '[' 55 -le 59 ']'

+ echo 'You should eat a bit more fat.'

You should eat a bit more fat.

Test the number of arguments

The following example explains how to modify the previous example to print a message if more than two position operators are entered:

hsoub ~> cat weight.sh

#! / bin / bash

This program prints a message about your weight if you enter your weight in kilograms and your height in centimeters

if [! $ # == 2]; then

** echo "Usage: $ 0 weight_in_kilos length_in_centimeters"**

** exit**

fi

weight = "$ 1"

height = "$ 2"

idealweight = $ [$ height - 110]

if [$ weight -le $ idealweight]; then

** echo "You should eat a bit more fat."**

else

** echo "You should eat a bit more fruit."**

fi

hsoub ~> weight.sh 70 150

You should eat a bit more fruit.

hsoub ~> weight.sh 70 150 33

Usage: ./weight.sh weight_in_kilos length_in_centimeters

In the previous example, $ 1 referred to the first operand, $ 2 to the second, and so on, and stored the number of arguments in # $. See Using the exit statement with if to better print usage messages.

Test for a file

This test is used in many programs to ensure the implementation of these

programs. There is no point in calling a program if you know it will not be implemented:

```
#! / bin / bash

# This program gives data about a file.

FILENAME = "$ 1"

echo "Properties for $ FILENAME:"

if [-f $ FILENAME]; then
  echo "Size is $ (ls -lh $ FILENAME | awk '{print $ 5}')"
  echo "Type is $ (file $ FILENAME | cut -d": "-f2 -)"
  echo "Inode number is $ (ls -i $ FILENAME | cut -d" "-f1 -)"
  echo "$ (df -h $ FILENAME | grep -v Mounted | awk '{print" On ", $
1", \
which is mounted as the ", $ 6," partition. "} ')"
else
  echo "File does not exist."
fi
```

Note that the file is referenced using a variable which is the first argument of the program in that case.When we do not have arguments, the locations of the files are saved in variables at the beginning of the program, and their content is indicated by using those variables, so you do not need to modify the file name in Program only once.

The names of files that contain spaces

The previous example fails if the value of $ 1 can be parsed as multiple words, and if can be fixed with double quotation marks around the file name or with [] instead of].

If / then / elif / else

The full image of if is:

```
if TEST-COMMANDS; then
CONSEQUENT-COMMANDS;
elif MORE-TEST-COMMANDS; then
MORE-CONSEQUENT-COMMANDS;
else ALTERNATE-CONSEQUENT-COMMANDS;
```

fi

The TEST-COMMANDS test orders are executed first. If the return status is zero, the dependent commands are executed. One of the "MORE-CONSEQUENT-COMMANDS" commands that accompany it is executed and the command completes.

If else is followed by an ALTERNATE-CONSEQUENT-COMMANDS list and the last order status in the last condition of the if or elif statement is nonzero, the list of substitute orders is executed, and the return status is the exit status of the last executed order or zero if no condition is met Of the conditions given in the orders.

Example

Here's an example that you can put in your crontab file for daily execution:

hsoub /etc/cron.daily> cat disktest.sh

#! / bin / bash

 # This software performs a simple hard drive test.

space = `df -h | awk '{print $ 5}' | grep% | grep -v Use | sort -n | tail -1 | cut -d "%" -f1 -`

alertvalue = "80"

if ["$ space" -ge "$ alertvalue"]; then

** echo "At least one of my disks is nearly full!" | mail -s "daily diskcheck" root**

else

** echo "Disk space normal" | mail -s "daily diskcheck" root**

fi

Nested if statements

The if statement can be used within any other if any number of levels you want as long as you can control it. Here's an example that tests for leap years:

hsoub ~ / testdir> cat testleap.sh

#! / bin / bash

 # This program will inspect this year to see if we are in a leap year.

year = `date +% Y`

if [$ [$ year% 400] -eq "0"]; then

** echo "This is a leap year. February has 29 days."**

```
elif [$ [$ year% 4] -eq 0]; then
        if [$ [$ year% 100] -ne 0]; then
          echo "This is a leap year, February has 29 days."
        else
          echo "This is not a leap year. February has 29 days."
        fi
else
  echo "This is not a leap year. February has 29 days."
fi
```

```
hsoub ~ / testdir> date
Tue Jan 14 20:37:55 CET 2003

hsoub ~ / testdir> testleap.sh
This is not a leap year.
```

Logical operations
The previous program can be summarized using the logical operators AND (&&), and OR (||).

```
#! / bin / bash
# Check this program to know if we are in a leap year

year = `date +% Y`

if ((("$ year"% 400) == "0")) || ((("$ year"% 4 == "0") && ("$ year"% 100! = "0"))); then
  echo "This is a leap year. Don't forget to charge the extra day!"
else
  echo "This is not a leap year."
fi
```

We used the double brackets in the previous example to test a mathematical expression, see mathematical expansion in the separation of expansions in Bash, this is equivalent to let. Don't use square brackets here or try something like [400% year $] $, because square brackets here aren't really real. You might also want to use text editors that support different color schemes depending on the language you are writing, as they are useful for identifying errors in your code, one of which is gvim but you'll also find kwrite and others.

Use the exit statement with if

We first identified the exit statement, which terminates the entire execution of the program, and is often used if the request the user requested was wrong, the statement was not executed successfully, or if another error occurred. The exit statement takes an optional argument, which is the numerical symbol of the exit state, which is passed back to the parent shell and stored in the? $ Variable. When the value of the argument is zero, it means that the program was executed successfully, and any other value may be used by programmers to pass different messages to the parent shell in order to take different actions according to the failure or success of the subprocess.If no argument is given to the exit command, the parent shell uses the current value. For the? $ Variable.

Here is an example of penguin.sh, which is explained with a little modification so that it sends its exit status to the parent shell where feed.sh is present:

```
hsoub ~ / testdir> cat penguin.sh
#! / bin / bash

# (Tux) This program allows you to serve food to the penguin
# The penguin will only be happy when a fish is given
# We've also added dolphin and camel options.

if ["$ menu" == "fish"]; then
  if ["$ animal" == "penguin"]; then
    echo "Hmmmmmm fish ... Tux is happy!"
  elif ["$ animal" == "dolphin"]; then
    echo "Pweetpeettreetppeterdepweet!"
  else
    echo "* prrrrrrrt *"
  fi
else
  if ["$ animal" == "penguin"]; then
    echo "Tux doesn't like that. Tux wants fish!"
    exit 1
  elif ["$ animal" == "dolphin"]; then
    echo "Pweepwishpeeterdepweet!"
    exit 2
  else
```

```
    echo "Will you read this sign ?!"
    exit 3
  fi
fi
```

This program is called in the following example, which exports its menu and animal variables:

```
hsoub ~ / testdir> cat feed.sh
#! / bin / bash
# penguin.sh This program behaves according to the exit status it gives

export menu = "$ 1"
export animal = "$ 2"

feed = "/ nethome / anny / testdir / penguin.sh"

$ feed $ menu $ animal

case $? in

1)
  echo "Guard: You'd better give'm a fish, less they get violent ..."
  ;;
2)
  echo "Guard: It's because of people like you that they are leaving earth all the time ..."
  ;;
3)
  echo "Guard: Buy the food that the Zoo provides for the animals, you ***, how
do you think we survive? "
  ;;
*)
  echo "Guard: Don't forget the guide!"
  ;;
esac

hsoub ~ / testdir> ./feed.sh apple penguin
```

Tux doesn't like that. Tux wants fish!

Guard: You'd better give'm a fish, less they get violent ...

Exit status codes can be freely chosen as you see, and exit commands usually have a series of specific codes

Use the case structure

Simplified cases

It may be easy to use the if statements for its intuitive style, but that ease turns into confusion when you encounter a few different choices of possible actions to take. For such cases we use the case, and its linguistic structure is as follows:

case EXPRESSION in CASE1) COMMAND-LIST ;; CASE2) COMMAND-LIST ;; ... CASEN) COMMAND-LIST ;; esac

Each case in the previous expression matches a pattern, the commands in the COMMAND-LIST list are executed for the first match, and the pipe typeface | To separate multiple patterns, terminate a parameter (list of styles, each case is called a case and its commands, and each item must end with two semicolons; any case ends with esac. See the following example that explains Case statements To send more diverse warning messages with disktest.sh:

hsoub ~ / testdir> cat disktest.sh

#! / bin / bash

 # This program performs a simple hard disk space test.

space = `df -h | awk '{print $ 5}' | grep% | grep -v Use | sort -n | tail -1 | cut -d "%" -f1 -`

case $ space in

[1-6] *)

Message = "All is quiet."

;;

[7-8] *)

Message = "Start thinking about cleaning out some stuff. There's a partition that is $ space% full."

;;

9 [1-8])

Message = "Better hurry with that new disk ... One partition is $ space% full."

;;

99)

Message = "I'm drowning here! There's a partition at $ space%!"

;;

***)**

Message = "I seem to be running with an nonexistent amount of disk space ..."

;;

esac

echo $ Message | mail -s "disk report` date` "hsoub

hsoub ~ / testdir>

You have new mail.

hsoub ~ / testdir> tail -16 / var / spool / mail / hsoub

From hsoub @ alaraby Tue Jan 14 22:10:47 2003

Return-Path: <hsoub @ alaraby>

Received: from alaraby (localhost [127.0.0.1])

by alaraby (8.12.5 / 8.12.5) with ESMTP id h0ELAlBG020414

for <hsoub @ alaraby>; Tue, 14 Jan 2003 22:10:47 +0100

Received: (from hsoub @ localhost)

by alaraby (8.12.5 / 8.12.5 / Submit) id h0ELAltn020413

for hsoub; Tue, 14 Jan 2003 22:10:47 +0100

Date: Tue, 14 Jan 2003 22:10:47 +0100

From: Hsoub <hsoub @ alaraby>

Message-Id: <200301142110.h0ELAltn020413@alaraby>

To: hsoub @ alaraby

Subject: disk report Tue Jan 14 22:10:47 CET 2003

Start thinking about cleaning out some stuff. There's a partition that is 87% full.

hsoub ~ / testdir>

You could open your mail client to check the results, but this example was to show that the program sends complete mail messages including the To address, Subject, and From. You can find many examples of using case statements in your init code folder. Startup scripts use start and stop statements to run or stop system processes, you'll find a theoretical example below.

Example initscript

The init codes use the case statement to start and stop system services as well as to query them. In the following example, it is part of a program that starts the Anacron service, a daemon that executes commands periodically at intervals determined by days.

```
case "$ 1" in
        start)
            start
            ;;

        stop)
            stop
            ;;

        status)
            status anacron
            ;;
        restart)
            stop
            start
            ;;
        condrestart)
            if test "x`pidof anacron`"! = x; then
                stop
                start
            fi
            ;;
```

```
*)
    echo $ "Usage: $ 0 {start | stop | restart | condrestart | status}"
    exit 1
esac
```

The tasks to be performed, such as turning on and off the sprite, are defined within functions whose source is, in part, the etc / rc.d / init.d / functions / file,

Chapter V

Repetitive tasks in Bash

In this section, you will learn how repetitive commands can be merged into iterative loops, and how repeating loops are constructed using either or for or while statements until one or all of them are combined. The for loop executes a task a certain number of times. Until or while statement to specify when the loop should stop.

Loops can be interrupted or repeated using break and continue statements, and a file can be used as an input for a loop via the input forwarding coefficient. Loops can also read output from commands fed through a pipe.

You will also learn how the select structure is used to print lists in interactive scripts, and how to use a shift statement to execute a command argument in an iterative loop within a shell script.

Episode for in Bash

How to make a loop for iterative

The for loop is the first of the three iterative units of chance, which allows a list of values to be assigned, and a list of commands is executed for each of

these values.

The linguistic structure of this iterative cycle is as follows:

for NAME [in LIST]; do COMMANDS; done

If [in LIST] does not exist, @ $ in is replaced by it, and executed for commands (COMMANDS) once for each position parameter set to a value, see Variables in Bash and check the command line arguments.

The return status is the exit status of the last executed order, and if no order is executed because LIST does not expand to any element, the return status is zero.

The name (NAME) can be any variable name although i is frequently used, and the LIST list can be made up of any words, text, or numbers, which may be literal or generated by any command.

Similarly, the commands to be executed for each value (COMMANDS) can be any system commands, programs, shell statement, or scripts. When the first loop starts, the NAME variable is first set to the first item in the LIST, its value changes the second time to the second item in the list, and so on. The iteration loop ends when the NAME variable passes over all the values in the list.

Examples

Use the Replace commands to select menu items
The first example will be a command line, which shows the use of a for loop to create a backup of each .xml file, and you can work on your sources safely after the command is issued:

[wiki @ hsoub ~ / articles] ls * .xml

file1.xml file2.xml file3.xml

[wiki @ hsoub ~ / articles] ls * .xml> list

[wiki @ hsoub ~ / articles] for i in `cat list`; do cp "$ i" "$ i" .bak; done

[wiki @ hsoub ~ / articles] ls * .xml *

file1.xml file1.xml.bak file2.xml file2.xml.bak file3.xml file3.xml.bak

This example shows plain text files, potentially programs, inside the / sbin folder:

for i in `ls / sbin`; do file / sbin / $ i | grep ASCII; done

Use variable content to select menu items

The following is a specific program that converts system-compatible HTML files to PHP files. The conversion is done by deleting the first 25 lines and the last 21, and placing two tags from PHP that add two header and footer lines:

```
[wiki @ hsoub ~ / html] cat html2php.sh
#! / bin / bash
# php to html specific program to convert files
LIST = "$ (ls * .html)"
for i in "$ LIST"; do
    NEWNAME = $ (ls "$ i" | sed -e 's / html / php /')
    cat beginfile> "$ NEWNAME"
    cat "$ i" | sed -e '1,25d' | tac | sed -e '1,21d' | tac >> "$ NEWNAME"
    cat endfile >> "$ NEWNAME"
done
```

There is no way to know the line number at which deletion will begin until we reach the end because we do not count the lines here, and the problem is solved using tac, which reflects the lines within the file.

Basename command

It is best to use the basename command to change the html suffix to php. See the man's pages for more details.

Abnormal typefaces

If you expand the list to file names with special characters such as spaces and other abnormal characters, the solution is to use globbing in the shell as follows:

```
for i in $ PATHNAME / *; do
commands
done
```

While loop in Bash

How to make a while loop

The iterative while loop allows repeated execution of a list of commands as long as the command in a while loop is executed successfully (exiting zero). The linguistic structure of this episode is:

while CONTROL-COMMAND; do CONSEQUENT-COMMANDS; done

CONTROL-COMMAND, which will control the loop, denotes any command that fails or succeeds, while CONSEQUENT-COMMANDS, which will be repeated, may be any shell program or program or shell structure.

The loop comes out immediately after the control command fails, and if the loop is in a program of chance, the following command is done after the loop ends because of the failure of the control command.

The return status is the same as the exit status of the last dependent command (CONSEQUENT-COMMANDS), or is zero if no order is executed.

Examples

A simple example to use while

Here's a simple example of using a while loop:

```
#! / bin / bash

# This program opens four terminal windows.

i = "0"

while [$ i -lt 4]
do
    xterm &
    i = $ [$ i + 1]
done
```

While nested loops

The following example is written in order to copy images taken by a webcam to a folder on a web server. It takes a picture every five minutes, every hour a new folder with the images of that hour is created, and a folder each day with 24 subfolders is created. The program runs in the background:

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

```
# This program copies files from my home folder to a folder in a web server.
# (Use remote folder ssh and scp keys)
# A new folder is created every hour

PICSDIR = / home / hsoub / pics
WEBDIR = / var / www / hsoub / webcam

while true; do
DATE = `date +% Y% m% d`
HOUR = `date +% H`
mkdir $ WEBDIR / "$ DATE"

while [$ HOUR -ne "00"]; do
DESTDIR = $ WEBDIR / "$ DATE" / "$ HOUR"
mkdir "$ DESTDIR"
mv $ PICDIR / *. jpg "$ DESTDIR" /
sleep 3600
HOUR = `date +% H`
done
done
```

Notice the use of true, this means: Continue execution until the process is forcibly interrupted, by a kill command or Ctrl + c. The following example creates files, which can be used for simulation tests:

```
#! / bin / bash

# This program creates a file every five minutes

while true; do
    touch pic-`date +% s`.jpg
    sleep 300
done
```

Note that you use the date command to create different types of file and folder names,

Use the system

The example above was for explanation only, and periodic inspections can be performed using the cron tool. Do not forget to direct output and errors when using programs executed from your crontab file!

Use keyboard input to control a while loop

The following program can be interpreted by the user when entering ctrl + c:

```
#! / bin / bash
# This program prints examples and tips
FORTUNE = / usr / games / fortune

while true; do
echo "On which topic do you want advice?"
cat << topics
politics
startrek
kernelnewbies
sports
bofh-excuses
magic
love
literature
drugs
education
topics

echo
echo -n "Make your choice:"
read topic
echo
echo "Free advice on the topic of $ topic:"
echo
$ FORTUNE $ topic
echo

done
```

The here document is used to present different options for the user to choose from - here or heredoc is a way to feed a shell program with short content

without typing it in a separate file.

Calculate the average
This program calculates the average user input that is tested before it is executed. If the input is out of range, a message is printed. If the user presses the q key, the loop comes out:

```
#! / bin / bash

# Calculate the average of a series of numbers.

SCORE = "0"
AVERAGE = "0"
SUM = "0"
NUM = "0"

while true; do

  echo -n "Enter your score [0-100%] ('q' for quit):"; read SCORE;

  if (("$ SCORE" <"0")) || (("$ SCORE"> "100")); then
    echo "Be serious. Common, try again:"
  elif ["$ SCORE" == "q"]; then
    echo "Average rating: $ AVERAGE%."
    break
  else
    SUM = $ [$ SUM + $ SCORE]
    NUM = $ [$ NUM + 1]
    AVERAGE = $ [$ SUM / $ NUM]
  fi

done

echo "Exiting."
```

Notice how variables in the last lines are left without quotation marks for calculations.

Until loop in Bash

How to make a loop until

This iterative loop is similar to a while loop except that the loop here is

executed until the TEST-COMMAND checksum is executed successfully, and the loop will remain active as long as the execution of the test command fails. The syntax is similar to the syntax of a while loop:

until TEST-COMMAND; do CONSEQUENT-COMMANDS; done

The return status of this loop is the same as the exit status of the last command executed from CONSEQUENT-COMMANDS or zero if no command is executed. The TEST-COMMAND test command can be any command that can come up with a success or failure, and the dependent commands can be any of the Unix commands, scripts, or shell structures.

The semicolon can replace ";" With one or more new lines if needed

Examples

This program is an upgraded version of the example we mentioned in nested while loops, and this version inspects the existence of sufficient disk space, if you do not find it deletes images from previous months:

```
#! / bin / bash

# This program copies images from my home folder to a folder in a web server
# A new folder is created every hour
# If the images take up too much space to add new, the last backup is deleted.

while true; do
DISKFUL = $ (df -h $ WEBDIR | grep -v File | awk '{print $ 5}' | cut -d "%" -f1 -)

until [$ DISKFUL -ge "90"]; do

        DATE = `date +% Y% m% d`
        HOUR = `date +% H`
        mkdir $ WEBDIR / "$ DATE"

        while [$ HOUR -ne "00"]; do
                DESTDIR = $ WEBDIR / "$ DATE" / "$ HOUR"
                mkdir "$ DESTDIR"
                mv $ PICDIR / *. jpg "$ DESTDIR" /
                sleep 3600
                HOUR = `date +% H`
        done
```

DISKFULL = $ (df -h $ WEBDIR | grep -v File | awk '{print $ 5}' | cut -d "%" -f1 -)

done

TOREMOVE = $ (find $ WEBDIR -type d -a -mtime +30)

for i in $ TOREMOVE; do

rm -rf "$ i";

done

done

Note that the HOUR and DISKFUL variables start and use the options with the ls and date commands in order to obtain an accurate choice of the TOREMOVE command.

The Break and Continue commands

Break command

Break is used to exit the current iteration loop before its original exit date.This happens in cases where you don't know how many times the loop should execute, for example, depending on user input.

The following example shows an interrupted while loop, an example developed by wisdom.sh from an example of using keyboard inputs to control a while loop:

#! / bin / bash

```
# This program prints examples and tips
# You can go out now better.

FORTUNE = / usr / games / fortune

while true; do
echo "On which topic do you want advice?"
echo "1. politics"
echo "2. startrek"
echo "3. kernelnewbies"
echo "4. sports"
echo "5. bofh-excuses"
echo "6.Magic"
echo "7. love"
echo "8. literature"
echo "9. drugs"
echo "10. education"
echo

echo -n "Enter your choice, or 0 for exit:"
read choice
echo

case $ choice in
    1)
    $ FORTUNE politics
    ;;
    2)
    $ FORTUNE startrek
    ;;
    3)
    $ FORTUNE kernelnewbies
    ;;
    4)
    echo "Sports are a waste of time, energy and money."
    echo "Go back to your keyboard."
    echo -e "\ t \ t \ t \ t - \" Unhealthy is my middle name \ "Soggie."
```

```
    ;;
    5)
    $ FORTUNE bofh-excuses
    ;;
    6)
    $ FORTUNE magic
    ;;
    7)
    $ FORTUNE love
    ;;
    8)
    $ FORTUNE literature
    ;;
    9)
    $ FORTUNE drugs
    ;;
    10)
    $ FORTUNE education
    ;;
    0)
    echo "OK, see you!"
    break
    ;;
    *)
    echo "That is not a valid choice, try a number from 0 to 10."
    ;;
esac
done
```

Note the break command that terminates the loop only - not the program -. This can be seen clearly by adding an echo command at the end of the program, and the echo command will also be executed when you type an entry that causes a break command to be executed (when the user types "0"). The break command also allows, in the case of overlapping loops, to determine which loops come out, see the info guide pages for more.

Continue command

The continue statement repeats the attached for, while, until, or select loop, and when used in the for loop, the ruling variable --controlling variable - takes the value of the next item in the list. If used in a while or until loop, the execution returns again with the TEST-COMMAND command placed at the beginning of the loop.

Examples

In the following example, the filenames are converted to lower case, and the continue statement returns the loop if no characters need to be converted.

These commands do not take a lot of system resources, and although such problems can be solved using the sed and awk commands, however, it is useful to know this method of solution as it is useful when performing large tasks, and it will not be necessary if tests are entered In the correct positions within the program, saving system resources:

[wiki @ hsoub ~ / test] cat tolower.sh

#! / bin / bash

 # This program converts all uppercase file names to lowercase names.

LIST = "$ (ls)"

for name in "$ LIST"; do

if [["$ name"! = * [[: upper:]] *]]; then

continue

fi

ORIG = "$ name"

NEW = `echo $ name | tr 'A-Z' 'a-z'' '

mv "$ ORIG" "$ NEW"

echo "new name for $ ORIG is $ NEW"

done

This program has at least one vulnerability: it replaces existing files, and the noclobber command is useful only when a redirect occurs. Also, the b-option for the mv command provides more security, but that security is useful only in case of unintentional replacement, as you can see in the following example:

[wiki @ hsoub ~ / test] rm *

[wiki @ hsoub ~ / test] touch test Test TEST

[wiki @ hsoub ~ / test] bash -x tolower.sh

++ ls

+ LIST = test

Test

TEST

+ [[test! = * [[: upper:]] *]]

+ continue

+ [[Test! = * [[: Upper:]] *]]

+ ORIG = Test

++ echo Test

++ tr A-Z a-z

+ NEW = test

+ mv -b Test test

+ echo 'new name for Test is test'

new name for Test is test

+ [[TEST! = * [[: Upper:]] *]]

+ ORIG = TEST

++ echo TEST

++ tr A-Z a-z

+ NEW = test

+ mv -b TEST test

+ echo 'new name for TEST is test'

new name for TEST is test

[wiki @ hsoub ~ / test] ls -a

./ ../ test test ~

The tr command comes as part of the textutils package and can perform all kinds of conversions on files.

Chapter VI

Functions in Bash

Functions provide an easy way to group commands that need to be executed repeatedly.

Functions are similar to mini-scripts, so they generate exit codes or return codes.

What are functions?

Shell functions are a way of grouping commands for later execution using a single group name as a whole or routine.

The list of commands associated with the name of a function is executed when that function is called as a simple command, and the function is executed within the shell in which it was declared. The special commands that are included before the shell functions will appear while searching for commands, and the special commands that are included are:

break,:,., continue, eval, exec, exit, export, readonly, return, set, shift, trap, unset.

Functions Syntax

Functions use one of these two rules:

function FUNCTION {COMMANDS; }

or

FUNCTION () {COMMANDS; }

These rules define a shell function called FUNCTION. You don't need to use a function, but if you don't use it, you will need the parentheses () as you see in the second rule.

Common errors

The space brackets {} must be separated from the content of the function between them, otherwise the content of the function will be misinterpreted, and the function content must end with a semicolon; Or a new line.

Position coefficients in functions

Functions are similar to miniature codes, they can accept parameters and use variables that cannot be used outside of functions via the local command, and can return values to the shell they called.

Functions have a system for interpreting position coefficients, although the position coefficients that are passed to a function are not the same as those passed to a command or text code. When the function is executed, its arguments become positional coefficients during execution, and the # special parameter that expands to the number of position coefficients is updated to reflect the change occurring.

The position coefficient does not change to 0, and the FUNCNAME variable is set to the function name during execution. If a return command is executed in a function, the function is executed, and the command is executed after the function is called. The values of the position coefficients and the special parameter # after the completion of a function return to the values they were before the function was executed. Here's a simple example:

[ahmad @ ahmadpc ~ / test] cat showparams.sh

#! / bin / bash

echo "This script demonstrates function arguments."

echo

echo "Positional parameter 1 for the script is $ 1."

echo

test ()

{

echo "Positional parameter 1 in the function is $ 1."

RETURN_VALUE = $?

echo "The exit code of this function is $ RETURN_VALUE."

}

test other_param

[ahmad @ ahmadpc ~ / test] ./showparams.sh parameter1

This script demonstrates function arguments.

Positional parameter 1 for the script is parameter1.

Positional parameter 1 in the function is other_param.

The exit code of this function is 0.

[ahmad @ ahmadpc ~ / test]

Note that the return value or the exit code of a function is often stored in a variable for probed later, and your system's init codes use the RETVAL variable checking technology in such a conditional test:

if [$ RETVAL -eq 0]; then

<start the daemon>

Or, as in the following example of the /etc/init.d/amd code where bash configuration features are used:

[$ RETVAL = 0] && touch / var / lock / subsys / amd

Commands after && are only executed if the test is successful, a shorter way to express if / then / if. The return code for the function is used as an exit code from the entire code, and you'll see many init codes ending with something like exit $ RETVAL.

Displaying Functions

All known functions of the current shell can be displayed using the set command without options. The functions remain after they are used unless the unset command is used to remove them.

[ahmad @ ahmadpc ~] which zless

zless is a function

zless ()

{

zcat "$ @" | "$ PAGER"

}

[ahmad @ ahmadpc ~] echo $ PAGER

less

This is an example of a function that is configured in a user's shell resource configuration files. In general, functions are more flexible than alternative commands and provide a simple and easy way to configure the user environment. Here's one for DOS users:

```
dir ()
{
    ls -F --color = auto -lF --color = always "$ @" | less -r
}
```

Recycling

You will find many programs on your system that use functions as a structured way to handle a series of commands. In some Linux systems, for example, you will find the etc / rc.d / init / functions / profile referred to as a source in all init text codes. In this way you only need to write repetitive tasks once and generally, tasks such as checking whether a task is running, turning on or off a daemon, and so on. Then if you need to write those tasks back then you need only recycle the code and not write it from scratch.

You can create your own etc / functions / file, which will contain all the functions that you use regularly on your system that may be spread over different codes. Once you have this file you can refer to it in any script you type by placing this line somewhere at the beginning of the program, and then you can rotate the functions afterwards:

. / etc / functions

Adjust the path
You may find this part you have in your / etc / profile file. Here the pathmung function is defined and used to set the path for the root user and the rest of the users:

```
pathmunge () {
        if! echo $ PATH | / bin / egrep -q "(^ |:) $ 1 ($ | :)"; then
           if ["$ 2" = "after"]; then
              PATH = $ PATH: $ 1
           else
              PATH = $ 1: $ PATH
           fi
        fi
}

# Path manipulation
if [`id -u` = 0]; then
        pathmunge / sbin
        pathmunge / usr / sbin
        pathmunge / usr / local / sbin
fi

pathmunge / usr / X11R6 / bin after
```

unset pathmunge

This function takes its first argument to be the name of the path. This path is added to the current path if it does not already exist. The second argument determines whether the path will be added before or after the current PATH variable is defined. Only usr / X11R6 / bin / is added to the normal user path, and the root user gets some additional folders containing system commands. The function is canceled after use so that it does not remain present.

Remote backup

The following example is used to back up books, and supports SSH keys to open the connection remotely. Here you specify two functions, buplinux and bupbash, each creating a tar file. It is then compressed and sent to a remote server, and the local copy is then erased. The program is set to execute the bupbash function only on Sunday.

```
# / bin / bash
```

```
LOGFILE = "/ nethome / tille / log / backupscript.log"
echo "Starting backups for` date` ">>" $ LOGFILE "

buplinux ()
{
DIR = "/ nethome / tille / xml / db / linux-basics /"
TAR = "Linux.tar"
BZIP = "$ TAR.bz2"
SERVER = "rincewind"
RDIR = "/ var / www / intra / tille / html / training /"

cd "$ DIR"
tar cf "$ TAR" src / *. xml src / images / *. png src / images / *. eps
echo "Compressing $ TAR ..." >> "$ LOGFILE"
bzip2 "$ TAR"
echo "... done." >> "$ LOGFILE"
echo "Copying to $ SERVER ..." >> "$ LOGFILE"
scp "$ BZIP" "$ SERVER: $ RDIR"> / dev / null 2> & 1
echo "... done." >> "$ LOGFILE"
echo -e "Done backing up Linux course: \ nSource files, PNG and EPS
images. \ nRubbish removed." >> "$ LOGFILE"
rm "$ BZIP"
}

bupbash ()
{
DIR = "/ nethome / tille / xml / db /"
TAR = "Bash.tar"
BZIP = "$ TAR.bz2"
FILES = "bash-programming /"
SERVER = "rincewind"
RDIR = "/ var / www / intra / tille / html / training /"

cd "$ DIR"
tar cf "$ TAR" "$ FILES"
echo "Compressing $ TAR ..." >> "$ LOGFILE"
bzip2 "$ TAR"
```

```
echo "... done." >> "$ LOGFILE"
echo "Copying to $ SERVER ..." >> "$ LOGFILE"
scp "$ BZIP" "$ SERVER: $ RDIR"> / dev / null 2> & 1
echo "... done." >> "$ LOGFILE"

echo -e "Done backing up Bash course: \ n $ FILES \ nRubbish removed." >> "$ LOGFILE"
rm "$ BZIP"
}

DAY = `date +% w`

if ["$ DAY" -lt "2"]; then
  echo "It is` date +% A`, only backing up Bash course. " >> "$ LOGFILE"
  bupbash
else
  buplinux
  bupbash
fi


echo -e "Remote backup` date` SUCCESS \ n ---------- ">>" $ LOGFILE "
```

This program runs via cron, meaning that it works without user intervention, so we redirect the standard error from the scp command to dev / null /. Some may argue that all these separate steps can be combined into something like:

tar c dir_to_backup / | bzip2 | ssh server "cat> backup.tar.bz2"

But if you want results that can be restored when the program fails, that shortcut will not work for you.

Chapter VII

Writing interactive Bash programs

In this section we will learn how to enter user feedback and how to ask the user to enter data, usually using the echo / read structure, and discuss how files can be used as inputs and outputs using file descriptors and redirection, and how this can be combined with input from the user. This section also emphasizes the importance of providing detailed messages to those who use our shell programs. It is better to give too much information than to write a brief documentation.

Here documents are a type of shell structure that allows creating lists and saving options for users, and can also be used to perform interactive tasks in the background without interference.

-View user messages in Bash

The difference between interactive and non-interactive programs in Bash, a simple explanation of the command echo.

-Get user input in Bash

An explanation of the read command, file descriptions, requesting user input, Here documents and their use cases, as well as redirection and their various states.

View user messages in Bash

Interactive or non-interactive programs?

Some shell scripts work without any user intervention and are called non-interactive programs. Some of the advantages of these programs are that they work unexpectedly every time, and can work in the background as well. However, other shell programs need user input or you need to print messages to the user while they are working. This type of program has advantages as well. Turn it on.

Do not be lazy about writing comments in your programs while writing interactive programs. , So be sure to put messages telling the user, for example, to wait for the output to appear because the program is now calculating. Whenever possible, try to provide an estimate of the length of the ongoing process or how long the user should wait.

In the case of an input request from the user, it is preferable to give clear and transparent information about the data that you would like the user to enter.

The commands you use to print your comments to the user are echo and printf, and we'll discuss some examples below, although you may already be accustomed to using echo.

Use the echo command

The echo command outputs two arguments separated by spaces and ends with the new line character, and the return state is always zero. The echo command also takes a few options:

e-: Explains smuggled characters with a slash character (see escape characters in the character quote chapter).

n-: Trailing new line will eventually stop working.

Here's an example of adding comments in which we will use two examples from the command-line arguments section, separating the advanced uses of the conditional if statement:

hsoub ~ / test> cat penguin.sh

#! / bin / bash

 # (Tux) This program allows you to serve food to the penguin

#The penguin will only be happy when a fish is given, and we've added a little fun by adding some other animals

```
if ["$ menu" == "fish"]; then
  if ["$ animal" == "penguin"]; then
    echo -e "Hmmmmmm fish ... Tux is happy! \ n"
  elif ["$ animal" == "dolphin"]; then
    echo -e "\ a \ a \ aPweetpeettreetppeterdepweet! \ a \ a \ a \ n"
  else
    echo -e "* prrrrrrrt * \ n"
  fi
else
  if ["$ animal" == "penguin"]; then
    echo -e "Tux doesn't like that. Tux wants fish! \ n"
    exit 1
  elif ["$ animal" == "dolphin"]; then
    echo -e "\ a \ a \ a \ a \ a \ a \ aPweepwishpeeterdepweet! \ a \ a \ a"
    exit 2
  else
```

```
    echo -e "Will you read this sign ?! Don't feed the" $ animal "s! \ n"
    exit 3
  fi
fi

michel ~ / test> cat feed.sh
#! / bin / bash
# penguin.sh This program behaves according to the exit status given by a program

if ["$ #"! = "2"]; then
  echo -e "Usage of the feed script: \ t $ 0 food-on-menu animal-name \ n"
  exit 1
else

  export menu = "$ 1"
  export animal = "$ 2"

  echo -e "Feeding $ menu to $ animal ... \ n"

  feed = "/ nethome / anny / testdir / penguin.sh"

  $ feed $ menu $ animal

result = "$?"

  echo -e "Done feeding. \ n"

case "$ result" in

  1)
    echo -e "Guard: \" You'd better give'm a fish, less they get violent ... \ "\ n"
    ;;
  2)
    echo -e "Guard: \" No wonder they flee our planet ... \ "\ n"
    ;;
  3)
    echo -e "Guard: \" Buy the food that the Zoo provides at the entry, you *** \ "\ n"
```

echo -e "Guard: \" You want to poison them, do you? \ "\ n"

;;

***)**

echo -e "Guard: \" Don't forget the guide! \ "\ n"

;;

esac

fi

echo "Leaving ..."

echo -e "\ a \ a \ aThanks for visiting the Zoo, hope to see you again soon! \ n"

michel ~ / test> feed.sh apple camel

Feeding apple to camel ...

Will you read this sign ?! Don't feed the camels!

Done feeding.

Guard: "Buy the food that the Zoo provides at the entry, you *"**

Guard: "You want to poison them, do you?"

Leaving ...

Thanks for visiting the Zoo, hope to see you again soon!

hsoub ~ / test> feed.sh apple

Usage of the feed script: ./feed.sh food-on-menu animal-name

See escape characters for more detailed explanations.

Get user input in Bash

Use the read command

The read command is the complementary command for the echo and printf commands, and its syntax is as follows:

read [options] NAME1 NAME2 ... NAMEN

One line is read from the standard input or from a file descriptor entered as an argument for the u- option, the first word of the line is assigned to the first name NAME1, the second to the second name NAME2 and so on, then the remaining words and their intervening seperators are assigned to The last name is NAMEN, but if the words read from the income stream are less than the names, null values are assigned to those names.

Characters are used in the value of the IFS variable to separate the input line into words or tokens. See Split words in the expansions chapter in Bash. The backslash character may be used to follow the line and to remove any special meaning for the next character. Also, if no name is entered, the read line is assigned to the REPLY variable.

The return code for a read command is zero if there is no end-of-file character, the time for the command to expire (time out), or an invalid file descriptor is entered as an argument for the u-

The following is an improved version of testleap.sh from nested if statements, separating the advanced uses of the conditional if statement in Bash:

```
hsoub ~ / test> cat leaptest.sh
#! / bin / bash
# This program will inspect this year to see if we are in a leap year.

echo "Type the year that you want to check (4 digits), followed by [ENTER]:"

read year

if ((("$ year"% 400) == "0")) || ((("$ year"% 4 == "0") && ("$ year"% 100! =
"0"))); then
  echo "$ year is a leap year."
else
  echo "This is not a leap year."
fi
```

hsoub ~ / test> leaptest.sh

Type the year that you want to check (4 digits), followed by [ENTER]:

2000

2000 is a leap year.

Request user input
The following example shows how inductors are used to explain what a user should enter:

hsoub ~ / test> cat friends.sh

```
#! / bin / bash

# This program saves your address book up to date

friends = "/ var / tmp / michel / friends"

echo "Hello," $ USER ". This script will register you in Hsoub's friends database."

echo -n "Enter your name and press [ENTER]:"
read name
echo -n "Enter your gender and press [ENTER]:"
read -n 1 gender
echo

grep -i "$ name" "$ friends"

if [$? == 0]; then
  echo "You are already registered, quitting."
  exit 1
elif ["$ gender" == "m"]; then
  echo "You are added to Hsoub's friends list."
  exit 1
else
  echo -n "How old are you?"
  read age
  if [$ age -lt 25]; then
    echo -n "Which color of hair do you have?"
    read color
    echo "$ name $ age $ color" >> "$ friends"
```

```
    echo "You are added to Hsoub's friends list. Thank you so much!"
  else
    echo "You are added to Hsoub's friends list."
    exit 1
  fi
fi
```

```
michel ~ / test> cp friends.sh / var / tmp; cd / var / tmp

michel ~ / test> touch friends; chmod a + w friends

michel ~ / test> friends.sh
Hello, hsoub. This script will register you in Hsoub's friends database.
Enter your name and press [ENTER]: michel
Enter your gender and press [ENTER]: m
You are added to Hsoub's friends list.

hsoub ~ / test> cat friends
```

Note that the output is not neglected here, the program only stores information about the people that the user is interested in, but it will always tell you that you have been added to the list, unless you are already there. Any user can now start the program:

```
[anny @ octarine tmp] $ friends.sh
Hello, anny. This script will register you in Hsoub's friends database.
Enter your name and press [ENTER]: anny
Enter your gender and press [ENTER]: f
How old are you? 22
Which color of hair do you have? black
You are added to Hsoub's friends list.
```

Your friends list will look soon after this:

```
tille 24 black
anny 22 black
katya 22 blonde
maria 21 black
--output omitted--
```

This situation is not ideal since anyone can modify the user's computer files (but cannot delete them), and you can solve this problem using the special access modes on the program file, see SUID and SGID in a guide to Linux.

Redirection and file descriptors

You may know from the simple use of shell that the input and output of an object may be redirected before execution using special characters interpreted by shell - redirect operators - and redirection can also be used to open and close files for the current shell execution environment.

Redirection may also occur in a shell script so that it can receive input from a file, for example, or send output to it. The user can review the output file later, and that output file can be used as input for another program.

The inputs and outputs of a file can be completed using integer handles that monitor all open files of a process. These numerical values are known as file descriptors. And 2 respectively, and those numbers and corresponding devices are saved. Bash shell can also take TCP and UDP ports on networked hosts as file descriptions.

The following example demonstrates how saved file attributes point to physical devices:

hsoub ~> ls -l / dev / std *

lrwxrwxrwx 1 root root 17 Oct 2 07:46 / dev / stderr -> ../proc/self/fd/2

lrwxrwxrwx 1 root root 17 Oct 2 07:46 / dev / stdin -> ../proc/self/fd/0

lrwxrwxrwx 1 root root 17 Oct 2 07:46 / dev / stdout -> ../proc/self/fd/1

hsoub ~> ls -l / proc / self / fd / [0-2]

lrwx ------ 1 hsoub hsoub 64 Jan 23 12:11 / proc / self / fd / 0 -> / dev / pts / 6

lrwx ------ 1 hsoub hsoub 64 Jan 23 12:11 / proc / self / fd / 1 -> / dev / pts / 6

lrwx ------ 1 hsoub hsoub 64 Jan 23 12:11 / proc / self / fd / 2 -> / dev / pts / 6

Note that each process has its own view of files in proc / self / since it is a symbolic link to <proc / <process_ID /, you should probably consider info MAKEDEV and info proc for more information about proc / sub folders and how your system processes file descriptors Standard for each process underway.

The following steps are followed in order when you execute an order:

If the standard output of a previous command is entered into the standard

input of the current command via a pipe, the file descriptor proc / <current_process_ID> / fd / 0 is updated to target the same unknown pipe as proc / <previous_process_ID> / fd / 1.

If the standard output of the current command is entered into the standard input of the next command via a pipe, the file descriptor proc / <current_process_ID> / fd / 1 is updated to target another unknown pipe.

The redirection of the current command is handled from left to right.

Forwarding "N> & M" or "N <& M" after a command has the same effect as creating or updating the proc / self / fd / N symbolic link with the same target as the proc / self / fd / M link.

The "N> file" and "N <file" redirects have the effect of creating or updating the proc / self / fd / N symbolic link with the target file.

Closing the file descriptor "- & <N" has the effect of deleting the symbolic proc / self / fd / N.

The current command is only executed now.

Not many changes occur when you execute a program from the command line because the sub-shell will use the same file descriptors as the parent shell. If the parent shell does not exist, as in the case of the command using the cron tool, the standard file descriptors are pipes or any temporary files unless A redirect image is used. See the following example that shows the output of a simple at program:

hsoub ~> date

Fri Jan 24 11:05:50 CET 2003

hsoub ~> at 1107

warning: commands will be executed using (in order)

a) $ SHELL b) login shell c) / bin / sh

at> ls -l / proc / self / fd /> /var/tmp/fdtest.at

at> <EOT>

job 10 at 2003-01-24 11:07

hsoub ~> cat /var/tmp/fdtest.at

total 0

lr-x ------ 1 michel michel 64 Jan 24 11:07 0 -> / var / spool / at /! 0000c010959eb (deleted)

l-wx ------ 1 michel michel 64 Jan 24 11:07 1 -> /var/tmp/fdtest.at

l-wx ------ 1 michel michel 64 Jan 24 11:07 2 -> / var / spool / at / spool / a0000c010959eb

lr-x ------ 1 michel michel 64 Jan 24 11:07 3 -> / proc / 21949 / fd

Redirect errors

It is clear from the examples above that you can add inputs and outputs to a program - see file inputs and outputs below - but some may forget to redirect errors - outputs that we may need later. Perhaps if you are lucky you will be sent errors to know the reasons for the failure of the program, but if you do not have a share in that luck errors cause the failure of your program, but you can not correct them because they were not sent to you or picked up elsewhere.

Note that the order of precedence for error routing is very important, for example, this will direct the standard output of the ls command to an unaccessible-in-spool file in var / tmp /:

ls -l * 2> / var / tmp / unaccessible-in-spool

This will direct both the standard input and output to the spoollist file:

ls -l *> / var / tmp / spoollist 2> & 1

The following command directs the standard output only to the destination file because the standard error was copied to the standard output before the output was redirected.

Errors are usually directed to dev / null / when you make sure that the user does not need them, you'll find hundreds of examples in your system's startup programs. Bash allows redirecting both the standard output and the standard error to the file whose name is the result of the FILE expansion on this image: File <&, which is equivalent to the FILE 2> & 1 <structure used in the previous examples. This is also combined with a redirect to dev / null /, for example, when you want to execute a command without looking at the errors it will produce or the output it will produce.

File input and output

Use dev / fd /

The dev / fd / folder contains entries with names of 0, 1, 2, and so on, and opening the dev / fd / N / file is equivalent to repeating the N file descriptor, even if your system has dev / stdin /, dev / stdout / and dev / / stderr, you'll see that these are rewarded dev / fd / 0, dev / fd / 1 and dev / fd / 2 respectively.

The main use of dev / fd / files is a coincidence, and this mechanism allows programs that use pathname arguments to handle both standard input and standard output in the same way as other path names. If dev / fd / is not available in the system, you should find a way to overcome the problem, for example by using a dash - to illustrate that the program should read from a tube, see the following example:

hsoub ~> filter body.txt.gz | cat header.txt - footer.txt

This text is printed at the beginning of each print job and thanks the sysadmin for setting us up such a great printing infrastructure.

Text to be filtered.

This text is printed at the end of each print job.

The Read command in shell programs

The following example shows how to switch between file and command line entries:

```
hsoub ~ / testdir> cat sysnotes.sh
#! / bin / bash

# This script makes an index of important config files, puts them together in
# a backup file and allows for adding comment for each file.

CONFIG = / var / tmp / sysconfig.out
rm "$ CONFIG" 2> / dev / null

echo "Output will be saved in $ CONFIG."

# create fd 7 with same target as fd 0 (save stdin "value")
exec 7 <& 0

# update fd 0 to target file / etc / passwd
exec </ etc / passwd

# Read the first line of / etc / passwd
read rootpasswd

echo "Saving root account info ..."
echo "Your root account info:" >> "$ CONFIG"
echo $ rootpasswd >> "$ CONFIG"

# update fd 0 to target fd 7 target (old fd 0 target); delete fd 7
exec 0 <& 7 7 <& -

echo -n "Enter comment or [ENTER] for no comment:"
read comment; echo $ comment >> "$ CONFIG"

echo "Saving hosts information ..."
```

```
# first prepare a hosts file not containing any comments
TEMP = "/ var / tmp / hosts.tmp"
cat / etc / hosts | grep -v "^ #"> "$ TEMP"

exec 7 <& 0
exec <"$ TEMP"

read ip1 name1 alias1
read ip2 name2 alias2

echo "Your local host configuration:" >> "$ CONFIG"

echo "$ ip1 $ name1 $ alias1" >> "$ CONFIG"
echo "$ ip2 $ name2 $ alias2" >> "$ CONFIG"

exec 0 <& 7 7 <& -

echo -n "Enter comment or [ENTER] for no comment:"
read comment; echo $ comment >> "$ CONFIG"
rm "$ TEMP"

hsoub ~ / testdir> sysnotes.sh
Output will be saved in /var/tmp/sysconfig.out.
Saving root account info ...
Enter comment or [ENTER] for no comment: hint for password: blue lagoon
Saving hosts information ...
Enter comment or [ENTER] for no comment: in central DNS

hsoub ~ / testdir> cat /var/tmp/sysconfig.out
Your root account info:
root: x: 0: 0: root: / root: / bin / bash
hint for password: blue lagoon
Your local host configuration:
127.0.0.1 localhost.localdomain localhost
192.168.42.1 tintagel.kingarthur.com tintagel
in central DNS
```

Close the file descriptors
It is good to get used to attaching a file descriptor when it is no longer needed, since the sub-processes inherit the open file descriptors.

exec fd <& -

In the previous example, the file descriptor 7, which was assigned to standard input, was closed each time a user needed access to the physical device of the standard input, which was often the keyboard. See the following example showing only the standard error redirected to a pipe:

hsoub ~> cat listdirs.sh

#! / bin / bash

This program prints the standard output unchanged, while the standard error is drawn

awk for processing with

INPUTDIR = "$ 1"

fd 6 targets fd 1 target (console out) in current shell

exec 6> & 1

fd 1 targets pipe, fd 2 targets fd 1 target (pipe),

fd 1 targets fd 6 target (console out), fd 6 closed, execute ls

ls "$ INPUTDIR" / * 2> & 1> & 6 6> & - \

Closes fd 6 for awk, but not for ls.

| awk 'BEGIN {FS = ":"} {print "YOU HAVE NO ACCESS TO" $ 2}' 6> &
-

fd 6 closed for current shell

exec 6> & -

Documents Here

Your program may need to address other programs that require input, and the document here provides a way to read the shell from the current source until you find the line that contains only the search text (with no blank spaces followed by it). The result is that you do not need to communicate with separate files, all you have to do is use special chars, and the program will look better than if you used a series of echo commands:

hsoub ~> cat startsurf.sh

#! / bin / bash

This program offers an easy way to choose between browsers

echo "These are the web browsers on this system:"

```
# here Start a document
cat << BROWSERS
mozilla
links
lynx
konqueror
opera
netscape
BROWSERS
# here Document finished

echo -n "Which is your favorite?"
read browser

echo "Starting $ browser, please wait ..."
$ browser &

hsoub ~> startsurf.sh
These are the web browsers on this system:
mozilla
links
lynx
konqueror
opera
netscape
Which is your favorite? opera
Starting opera, please wait ...
```

Although we are talking about "document" here, it should be in the same program file. Here's an example that automatically installs a package, although you often have to confirm the installation:

```
#! / bin / bash

# yum This program automatically installs packages using.

if [$ # -lt 1]; then
        echo "Usage: $ 0 package."
        exit 1
```

fi

yum install $ 1 << CONFIRM

y

CONFIRM

The previous example works like this, where the program answers y automatically when [Is this ok [y / N] appears:

[root @ hsoub bin] # ./install.sh tuxracer

Gathering header information file (s) from server (s)

Server: Fedora Linux 2 - i386 - core

Server: Fedora Linux 2 - i386 - freshrpms

Server: JPackage 1.5 for Fedora Core 2

Server: JPackage 1.5, generic

Server: Fedora Linux 2 - i386 - updates

Finding updated packages

Downloading needed headers

Resolving dependencies

Dependencies resolved

I will do the following:

[install: tuxracer 0.61-26.i386]

Is this ok [y / N]: EnterDownloading Packages

Running test transaction:

Test transaction complete, Success!

tuxracer 100% done 1/1

Installed: tuxracer 0.61-26.i386

Transaction (s) Complete