CDOT
SICT AR Meeting Area
People

get involved with CDOT

as a Student
as an Open Source Community Member
as a Company

courses

BTC640
BTH740
BTP300
DPI908
DPS901
DPS905
DPS909
DPS911
DPS914
DPS915
DPS924
DPS931
EAC234
ECL500
GAM531
GAM666
GAM670
GPU610
LUX Program
MAP524
OOP344
OPS235
OPS245
OPS335
OPS345
OPS435
OPS445
OPS535
OPS635
OSD600
OSD700
OSL640
OSL740
OSL840

ULI101 Week 10

Contents [hide]

Shell Startup and Configuration Files [edit]

Shell Start-up/Configuration files are settings that are applied every time a shell is created. Start-up files are sequences of shell commands (scripts). They also apply when users log in, as it creates a shell.

There is a single system-wide configuration file that belongs to the root user: `/etc/profile`

User-specific configuration files that belong to the user are hidden files found in the user's home directory:

- `.bash_profile`
- `.bashrc`
- `.bash_logout` :: Executed when you log out

`/etc/profile` [edit]

This file can only be modified by the root user. Affects the environment of all users, regardless of their default shell. Bash users can change their environment by modifying the `.bash_profile` or the `.bashrc` files. Different shells have different configuration files.

Real World Mozilla
RHT524
SBR600
SEC520
SPO600
SRT210
ULI101

course projects

Course Project List
Potential Course Projects
Contrib Opportunities

links

CDOT
Planet CDOT
FSOSS

Tools

What links here
Related changes
Upload file
Special pages
Printable version
Permanent link
Page information

Other configuration files such as `.profile` exist - read comments in your `.bash_rc` file to find out more

`.bashrc` and `.bash_profile` [edit]

Located in the user’s home directory. These files are executed every time a user logs in or creates a new shell. Things vary depending whether the shell is interactive or not. By modifying either one of these files, each user can change his individual working environment. They can be used for the following:

- Setting the prompt and screen display
- Creating local variables
- Creating temporary Linux commands (aliases)
- Mapping new keys on the keyboard

Shell History [edit]

Many shells keep a history of recently executed command lines in a file. This history is used by users to save time, when executing same or similar commands over and over. Bash uses the up/down arrow keys. Use the Ctrl+r to search by keyword. Bash stores it’s history in the `.bash_history` file

Alias [edit]

A way to create “shortcuts” or temporary commands in UNIX. Stored in memory, while the user is logged in. Usually found in the `.bash_profile`. Syntax:

`alias name=value`

For example: `alias dir=ls`

Even complex command lines can have an alias. enclose the command within double quotes

`alias clearfile=”cat /dev/null >”`

Variables [edit]

Shell variables a classified in 2 groups:

- System (shell) variables, describing the working environment
- User-created variables, associated with scripts

Variables can be read/write or read-only. Name of a variable can be any sequence of letters and numbers, but it must not start with a number.

Common Shell Variables [edit]

Shell environment variables shape the working environment whenever you are logged in. Common shell variables include:

`PS1`	primary prompt
`PWD`	present working directory
`HOME`	absolute path to user’s home

`PATH`	list of directories where executables are
`HOST`	name of the host
`USER`	name of the user logged in
`SHELL`	current shell

The `set` command will display all available variables

The `PATH` variable [edit]

- `PATH` is an environment variable present in Unix/Linux operating systems, listing directories where executable programs are located
- Multiple entries are separated by a colon (:)
- Each user can customize a default system-wide `PATH`
- The shell searches these directories whenever a command is invoked in sequence listed for a match
- In case of multiple matches use the `which` utility to determine which match has a precedence
- On some systems the present working directory may not be included in the `PATH` by default
- Use `./` prefix or modify the `PATH` as needed

Assigning a Value [edit]

Syntax: `name=value`

```
# make a variable called course and have it store the string
"ULI101"
$ course=ULI101
$

# display the contents of course variable.
$ echo $course
ULI101
$

# make a variable called college and have it store the string
"Seneca College"
$ college=“Seneca College”
$

# display the contents of college variable.
$ echo $college
Seneca College
$
```

Read-Only Variables [edit]

- Including the keyword `readonly` before the command assignment prevents you from changing the variable afterwards. For example: `readonly phone=“123-4567”`
- After a variable is set, it can be protected from changing by using the readonly command. Syntax: `readonly variable`. For example: `readonly phone`

- If no variable name is supplied a list of defined read only variables will be displayed

Removing Variables [edit]

`variable=`

```
# 2 ways to remove variables
$ course=
$
$ unset college
$
```

Read-only variables cannot be removed, you must log out for them to be cleared.

Variable Substitution [edit]

Whenever you wish to read a variable (its contents), use the variable name preceded by a dollar sign ($). This is commonly called *variable substitution*.

```
$ name=Bob
$
$ echo $name
Bob
$ name=Alice
$
$ echo $name
Alice
```

Introduction to Shell Scripting [edit]

Usefulness of Shell programming [edit]

- Scope ranges from simple day-to-day tasks to large database driven CGI applications.
- Shell-dependent - each shell script is written for a specific shell, such as bash
- First line of each script usually specifies the path to the program which executes the script - `#!` statement, for example: `#!/bin/bash`
- Use the which utility to find out what path to use there
- This must be the first line and nothing can precede it, not even a single space
- This line is not necessary if the script will be executed in the default shell of the user
- Any line other than first one starting with a `#` is treated as a comment

Positional Parameters [edit]

- Every script can have parameters supplied
- Traditionally command line parameters are referred to as `$0, $1, $2, … $9`
- Parameters > `$9` can be accessed by using the `shift` command. shift will literally shift parameters to the left by one or more positions.

- Some shells can use the `${ }` form. This enables direct access to parameters > `$9`. For example: `${10}`
- `$*` and `$@` represent all command line arguments
- `$#` represents the number of parameters (not including the script name)

`echo` command [edit]

- Displays messages to the terminal followed by a newline. Use the `-n` option to suppress the default newline.
- Output can be redirected or piped.
- Arguments are usually double quoted.

`read` command [edit]

- The `read` command allows obtaining user input and storing it in a variable. Everything is captured until the Enter key is pressed.

```
echo -n "What is your name? "
read name
echo Hello $name
```

Using Logic [edit]

The purpose of the if statement is execute a command or commands based on a condition. The condition is evaluated by a test command, represented below by a pair of square brackets

```
# syntax of if command
if [ condition ]
then
command(s)
fi

# example
read password
if [ "$password" = "P@ssw0rd!" ]
then
echo "BAD PASSWORD!"
fi
```

`test` Command [edit]

The `test` command can be used in two ways:

- As a pair of square brackets: `[ condition ]`
- The test keyword: `test condition`

The condition test can result in true (0) or false (1), unless the negation "is not" (!), is used. The test can compare numbers, strings and evaluate various file attributes:

- Use `=` and `!=` to compare strings, for example: `[ "$name" = "Bob" ]`

Use `-z` and `-n` to check string length, example: `[ ! -z “$name” ]`
- Use `-gt`, `-lt`, `-eq`, `-ne`, `-le`, `-ge` for numbers like: `[ ”$salary” -gt 100000 ]`

Common file test operations include:

`-e` file exists

`-d` file exists and is a directory

`-s` file exists and has a size greater than zero

`-w` file exists and write permission is granted

Check `man test` for more details

Using Loops [edit]

A for loop is a very effective way to repeat the same command(s) for several arguments such as file names. The syntax and example of for command is shown below:

```
# syntax of for command
# Variable “item” will hold one item from the list every time
the loop iterates
for item in list  # "list" can be typed in explicitly or
supplied by a command
do
command(s)         # commands to be repeated
done

# example showing the making of 1000 directories named:
student_0 student_1
for id in $(seq 1 1000)
do
mkdir student_$id
done

# example showing the mailing everyone in ~/address the file
~/spam/newsletter.txt
for addr in $(cat ~/addresses)
do
mail -s ”Newsletter” $addr < ~/spam/newsletter.txt
done

# example showing counting down before printing "BLAST OFF!!!"
for count in 3 2 1 'BLAST OFF!!!'
do
sleep 1
echo $count
done
```

Categories: Pages with syntax highlighting errors | ULI101 | ULI101-2018