

A shell is simply a program that supplies certain services to users.

As such, a shell may take parameters whose values modify or define certain behaviors.

These parameters (or *shell variables* or global *environment variables*) typically have values that are set in certain *configuration files*.

When you install Linux, or use your rlogin account, many of these parameters will have default values determined by the system administrator or by Linux installer.

You may generally modify those default values and even define new parameters by editing configuration files within your home directory.

Open a bash shell and enter the command **\$HOME**... this will show the current value of the environment variable **HOME**.

The environment variable that is most often encountered is the **PATH** variable, which determines which directories the shell will search (and in what order) when the shell attempts to locate programs you are attempting to execute.



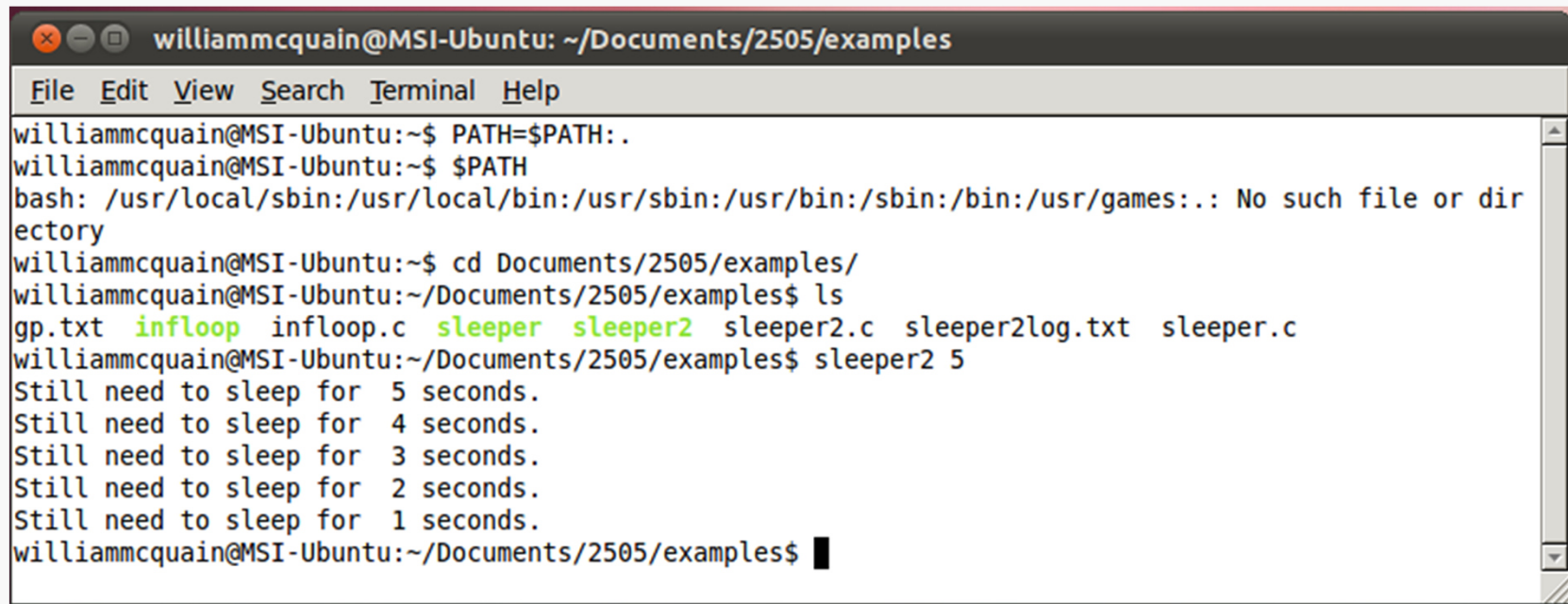
```
williammcquain@MSI-Ubuntu: ~  
File Edit View Search Terminal Help  
williammcquain@MSI-Ubuntu:~$ muddle  
muddle: command not found  
williammcquain@MSI-Ubuntu:~$ $PATH  
bash: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games: No such file or directory  
williammcquain@MSI-Ubuntu:~$
```

We see that the default **PATH** for this Ubuntu installation contains the directories:

- /usr/local/sbin**
- /usr/local/bin**
- /usr/sbin**
- /usr/bin**
- /sbin**
- /bin**
- /usr/games** (which apparently does not exist!)

You can change the value of a shell variable from the command line.

Let's add the current directory to the **PATH**:



```
williammcquain@MSI-Ubuntu: ~/Documents/2505/examples
File Edit View Search Terminal Help
williammcquain@MSI-Ubuntu:~$ PATH=$PATH:.
williammcquain@MSI-Ubuntu:~$ $PATH
bash: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:.: No such file or dir
ectory
williammcquain@MSI-Ubuntu:~$ cd Documents/2505/examples/
williammcquain@MSI-Ubuntu:~/Documents/2505/examples$ ls
gp.txt infloop infloop.c sleeper sleeper2 sleeper2.c sleeper2log.txt sleeper.c
williammcquain@MSI-Ubuntu:~/Documents/2505/examples$ sleeper2 5
Still need to sleep for 5 seconds.
Still need to sleep for 4 seconds.
Still need to sleep for 3 seconds.
Still need to sleep for 2 seconds.
Still need to sleep for 1 seconds.
williammcquain@MSI-Ubuntu:~/Documents/2505/examples$
```

Note that we can now run the user program **sleeper2** without specifying the path.

But... this only resets **PATH** for the current shell session.

When a bash shell is started, it automatically executes commands stored in certain files.

There are three kinds of shells:

(interactive) login shells

/etc/profile

~/.bash_profile

~/.bash_login

~/.profile

(sets values for various shell variables)

a system file that only the **root** user can modify
files in your **HOME** directory that you can change

interactive non-login shells

/etc/bashrc

~/.bashrc

(inherits login shell variables from files above)

another system file

another file in your **HOME** directory

non-interactive shells

files named by the environment variable **BASH_ENV**

(inherits login shell variables from files above)

If you try the **ls** command in your home directory, you will (probably) notice that the file **.bash_profile** is not listed.

Filenames that begin with a period are hidden by default.

You can use the **ls -a** command will show hidden files as well as non-hidden files.

When you open an interactive terminal session in Linux, the sequence described on the preceding slide is probably NOT followed by default.

In particular, `~/.bash_profile` is not executed automatically, and therefore changes you make to it will not be effective.

There is a simple fix for the issue:

- open a terminal session and go to Edit/Profile Preferences
- select the Title and Command tab
- check the box for “Run command as a login shell”

In fact, in my rlogin installation, `~/.bash_profile` did not exist initially; I had to create it with a text editor.

You should use **~/.bash_profile** to set changes to the PATH variable because **~/.bash_profile** is only executed once.

Here is a sample **.bash_profile** taken from Sobell:

```
if [ -f ~/.bashrc ]; then      # if .bashrc exists in the home directory
    source ~/.bashrc          # run it
fi

PATH=$PATH:.                  # add working directory to the path

export PS1='\h \W \!]\$ '     # configure the shell prompt
```

Normally, **~/.bashrc** is invoked from another configuration file, as shown here.

See the note in Sobell regarding adding the working directory to the path; **NEVER** add it at the beginning of the path!

Sobell has a good discussion of the various options for the appearance of the prompt.

Here is a sample **~/.bashrc** adapted from Sobell:

```
if [ -f /etc/bashrc ]; then      # if global bashrc exists , run it
    source /etc/bashrc          # note: no period in file name
fi
if [ -d "$HOME/bin" ] ; then    # add user's bin directory to path
    PATH="$HOME/bin:$PATH"
fi
set -o noclobber                # prevent silent overwriting of files
                                # (by redirection)
unset MAILCHECK                 # disable "you have mail" notice

alias rm='rm -i'                # always use interactive rm cmd
alias cp='cp -i'                # and interactive cp cmd
alias h='history | tail'
alias ll='ls -aF'
```

alias commands are a convenient way to create mnemonics for specialized execution of system commands.

You can define functions (think shell scripting) and run them from your shell.

For example, I might add the following to my `.bashrc` file:

```
...
# User specific functions
numUsers() {

    whoson=`who -q`      # save output from invoking who with -q
    echo ${whoson##*=}  # parse to isolate part we want and write it
}
...
```

Then, I can run this function, as a command, from my shell prompt:

```
wdm@VMCentos64:~> who -q
wdm wdm
# users=2

wdm@VMCentos64:~> numUsers
2
```

If I modify my prompt definition (in `.bash_profile`):

```
...  
PS1='Cmd \! `numUsers` \W> '  
...
```

My prompt now includes the output (note the backticks above) from the function:

```
Cmd 1000 14 ~>
```

I can easily turn a shell script into a shell function. Consider the backup script discussed earlier.

```
#!/bin/bash
# This script makes a backup of a directory to another server.
# Invocation:  ./backup3.sh DIRNAME
##### backup support fns
show_usage() {
    echo "Invocation:  ./backup3.sh DIRNAME"
}
. . .
##### body of script
if [[ $# -ne 1 ]]; then    # check for a parameter
    show_usage
    exit 1
fi
. . .
# Create a timestamp in the logfile to record the backup operation.
log_backup $BACKUPDIR $LOGFILE

exit 0                    # return 0 on success
```

Modify the script body to make it a function and embed into `.bashrc`:

```
. . .
##### backup support fns
show_usage() {
    echo "Invocation:  backup DIRNAME"
}
. . .
##### body of script
backup() {
    if [[ $# -ne 1 ]]; then    # check for a parameter
        show_usage
        exit 1
    fi
    . . .
    # Create a timestamp in the logfile to record the backup operation.
    log_backup $BACKUPDIR $LOGFILE

    exit 0                    # return 0 on success
}
. . .
```

I can now invoke this directly from the command-line (no path information is needed).