How to Set and Unset Local, User and System Wide Environment Variables in Linux

Environment Variables are some special variables that are defined in shell and are needed by programs while execution. They can be **system** defined or **user** defined. System defined variables are those which are set by system and are used by system level programs.

For e.g. PWD command is a very common system variable which is used to store the present working directory. User defined variables are typically set by user, either temporarily for the current shell or permanently. The whole concept of setting and un-setting environment variables revolves around some set of files and few commands and different shells.

In Broader terms, an environment variable can be in three types:

1. Local Environment Variable

One defined for the current session. These environment variables last only till the current session, be it remote login session, or local terminal session. These variables are not specified in any configuration files and are created, and removed by using a special set of commands.

2. User Environment Variable

These are the variables which are defined for a particular user and are loaded every time a user logs in using a local terminal session or that user is logged in using remote login session. These variables are typically set in and loaded from following configuration files: `.bashrc`, `.bash_profile`, `.bash_login`, `.profile` files which are present in user's home directory.

3. System wide Environment Variables

These are the environment variables which are available system-wide, i.e. for all the users present on that system. These variables are present in system-wide configuration files present in following directories and files: `/etc/environment`, `/etc/profile`, `/etc/profile.d/`, `/etc/bash.bashrc`. These variables are loaded every time system is powered on and logged in either locally or remotely by any user.

Understanding User-Wide and System-wide Configuration files

Here, we briefly describe various configuration files listed above that hold Environment Variables, either system wide or user specific.

.bashrc

This file is user specific file that gets loaded each time user creates a new **local session** i.e. in simple words, opens a new terminal. All environment variables created in this file would take effect every time a new local session is started.

.bash_profile

This file is user specific **remote login file**. Environment variables listed in this file are invoked every time the user is logged in remotely i.e. using ssh session. If this file is not present, system looks for either `.bash_login` or `.profile` files.

/etc/environment

This file is system wide file for creating, editing or removing any environment variables. Environment variables created in this file are accessible all throughout the system, by each and every user, both locally and remotely.

/etc/bash.bashrc

System wide `bashrc` file. This file is loaded once for every user, each time that user opens a local terminal session. Environment variables created in this file are accessible for all users but only through local terminal session. When any user on that machine is accessed remotely via a remote login session, these variables would not be visible.

/etc/profile

System wide profile file. All the variables created in this file are accessible by every user on the system, but only if that user's session is invoked remotely, i.e. via remote login. Any variable in this file will not be accessible for local login session i.e. when user opens a new terminal on his local system.

Note: Environment variables created using system-wide or user-wide configuration files can be removed by removing them from these files only. Just that after each change in these files, either log out and log in again or just type following command on the terminal for changes to take effect:

```
linsrv1:~ # su - oracle
oracle@linsrv1:~> source .profile
```

Set or Unset Local or Session-wide Environment Variables in Linux

Local Environment Variables can be created using following commands:

```
oracle@linsrv1:~> vartest="Hello World"
oracle@linsrv1:~> export vartest="Hello World"
```

These variables are session wide and are valid only for current terminal session. To Clear these session-wide environment variables following commands can be used:

1. Using env

By default, `"env"` command lists all the current environment variables. But, if used with `'-i'` switch, it temporarily clears out all the environment variables and lets user execute a command in current session in absence of all the environment variables.

```
$ env –i [Var=Value]… command args…
```

Here, `var=value` corresponds to any local environment variable that you want to use with this command only.

```
$ env –i bash
```

Will give bash shell which temporarily would not have any of the environment variable. But, as you exit from the shell, all the variables would be restored.

2. Using unset

Another way to clear local environment variable is by using unset command. To unset any local environment variable temporarily,

```
$ unset <var-name>
```

Where, `var-name` is the name of local variable which you want to un-set or clear.

3. Set the variable name to ”

Another less common way would be to set the name of the variable which you want to clear, to `''` (Empty). This would clear the value of the local variable for current session for which it is active.

NOTE – YOU CAN EVEN PLAY WITH AND CHANGE THE VALUES OF SYSTEM OR USER ENVIRONMENT VARIABLES, BUT CHANGES WOULD REFLECT IN CURRENT TERMINAL SESSION ONLY AND WOULD NOT BE PERMANENT.

Learn How to Create, User-Wide and System-Wide Environment Variables in Linux

In section, we will going to learn how to set or unset local, user and system wide environment variables in Linux with below examples:

1. Set and Unset Local Variables in Linux

a.) Here, we create a local variable `VAR1` and set it to any value. Then, we use unset to remove that local variable, and at the end that variable is removed.

```
$ VAR1='TecMint is best Site for Linux Articles'
$ echo $VAR1
$ unset VAR1
$ echo $VAR1
```

b.) Another way of creating a local variable is by using `export` command. The local variable created will be available for current session. To unset the variable simply set the value of variable to `''`.

```
$ export VAR='TecMint is best Site for Linux Articles'
$ echo $VAR
$ VAR=
$ echo $VAR
```

c.) Here, we created a local variable `VAR2` and set it to a value. Then in-order to run a command temporarily clearing out all local and other environment variables, we executed `'env -i'` command. This command here executed bash shell by clearing out all other environment variables. After entering `'exit'` on the invoked bash shell, all variables would be restored.

```
$ VAR2='TecMint is best Site for Linux Articles'
$ echo $VAR2
$ env -i bash
$ echo $VAR2
```

2. Set and Unset User-Wide Environment Variables in Linux

a.) Modify `.bashrc` file in your home directory to export or set the environment variable you need to add. After that source the file, to make the changes take effect. Then you would see the variable (`'CD'` in my case), taking effect. This variable will be available every time you open a new terminal for this user, but not for remote login sessions.

```
$ vi .bashrc
```

Add the following line to `.bashrc` file at the bottom.

```
export CD='This is TecMint Home'
```

Now run the following command to take new changes and test it.

```
$ source .bashrc

$ echo $CD
```

To remove this variable, just remove the following line in .bashrc file and re-source it:

b.) To add a variable which will be available for remote login sessions (i.e. when you ssh to the user from remote system), modify `.bash_profile` file.

```
$ vi .bash_profile
```

Add the following line to `.bash_profile` file at the bottom.

```
export VAR2='This is TecMint Home'
```

When on sourcing this file, the variable will be available when you ssh to this user, but not on opening any new local terminal.

```
$ source .bash_profile

$ echo $VAR2
```

Here, `VAR2` is not initially available but, on doing ssh to user on localhost, the variable becomes available.

```
$ ssh tecmint@localhost

$ echo $VAR2
```

To remove this variable, just remove the line in `.bash_profile` file which you added, and re-source the file.

NOTE: These variables will be available every time you are logged in to current user but not for other users.

3. Set and Unset System-Wide Environment Variables in Linux

a.) To add system wide no-login variable (i.e. one which is available for all users when any of them opens new terminal but not when any user of machine is remotely accessed) add the variable to `/etc/bash.bashrc` file.

```
export VAR='This is system-wide variable'
```

After that, source the file.

```
$ source /etc/bash.bashrc
```

Now this variable will be available for every user when he opens any new terminal.

```
$ echo $VAR

$ sudo su

$ echo $VAR

$ su -

$ echo $VAR
```

Here, same variable is available for root user as well as normal user. You can verify this by logging in to other user.

b.) If you want any environment variable to be available when any of the user on your machine is remotely logged in, but not on opening any new terminal on local machine, then you need to edit the file – `'/etc/profile'`.

```
export VAR1='This is system-wide variable for only remote sessions'
```

After adding the variable, just re-source the file. Then the variable would be available.

```
$ source /etc/profile

$ echo $VAR1
```

To remove this variable, remove the line from `/etc/profile` file and re-source it.

c.) However, if you want to add any environment which you want to be available all throughout the system, on both remote login sessions as well as local sessions(i.e. opening a new terminal window) for all users, just export the variable in /etc/environment file.

```
export VAR12='I am available everywhere'
```

After that just source the file and the changes would take effect.

```
$ source /etc/environment

$ echo $VAR12

$ sudo su

$ echo $VAR12

$ exit

$ ssh localhost

$ echo $VAR12
```

Here, as we see the environment variable is available for normal user, root user, as well as on remote login session (here, to localhost).

To clear out this variable, just remove the entry in the /etc/environment file and re-source it or login again.

NOTE: Changes take effect when you source the file. But, if not then you might need to log out and log in again.

Conclusion

Thus, these are few ways we can modify the environment variables. If you find any new and interesting tricks for the same do mention in your comments.

```
linsrv1:~ # su - oracle
oracle@linsrv1:~> ls -l ~/.bash* ~/.profile /etc/bash* /etc/profile
-rw-r--r-- 1 root   root     9159 Aug  4 16:59 /etc/bash.bashrc
-rw-r--r-- 1 root   root     1336 Sep 22  2014 /etc/bash_command_not_found
-rw-r--r-- 1 root   root     9168 Aug  4 16:59 /etc/profile
-rw------- 1 oracle oinstall 4643 Sep 24 23:30 /home/oracle/.bash_history
-rw-r--r-- 1 oracle oinstall 1177 Sep 18 01:18 /home/oracle/.bashrc
-rw-r--r-- 1 oracle oinstall 1157 Sep 19 12:31 /home/oracle/.profile

/etc/bash_completion.d:
total 56
-rwxr-xr-x 1 root root   561 Sep 22  2014 dbus-bash-completion.sh
-rw-r--r-- 1 root root  3546 Sep  1 16:17 dracut
-rw-r--r-- 1 root root 11158 Aug  2 12:22 grub
-rw-r--r-- 1 root root  2095 Sep  1 16:17 lsinitrd
-rw-r--r-- 1 root root 15593 Sep 23  2014 pulseaudio-bash-completion.sh
-rw-r--r-- 1 root root  3086 Sep 22  2014 scout.sh
-rw-r--r-- 1 root root  3532 Feb  4  2016 yast2-completion.sh
-rw-r--r-- 1 root root  4786 Mar  7  2016 zypper.sh
oracle@linsrv1:~>
```

You can **start a bash shell** in three ways:

- As a default login shell at login time
- As an interactive shell that is started by spawning a subshell
- As a non-interactive shell to run a script

Bash behaviour can be altered depending on how it is invoked. Some descriptions of different modes follow.
If Bash is spawned by `login` in a TTY, by an SSH daemon, or similar means, it is considered a **login shell**.
This mode can also be engaged using the `-l`/`--login` command line option.

Bash is considered an **interactive shell** when its standard input and error are connected to a terminal (for example, when run in a terminal emulator), and it is not started with the `-c` option or non-option arguments (for example, `bash` **`script`**). All interactive shells source `/etc/bash.bashrc` and `~/.bashrc`, while interactive *login* shells also source `/etc/profile` and `~/.bash_profile`.

Startup methods

1. Logging In

When you log in to the Linux system, the bash shell starts as a login shell. The login shell typically looks for five different startup files to process commands from:

- /etc/profile
- $HOME/.bash_profile
- $HOME/.bashrc
- $HOME/.bash_login
- $HOME/.profile
- $HOME/.bash_logout (when exiting)

The /etc/profile file is the main default startup file for the bash shell on the system.

All users on the system execute this startup file when they log in.

```
linsrv1:~ # cat /etc/profile
# /etc/profile for SUSE Linux
#
# PLEASE DO NOT CHANGE /etc/profile. There are chances that your changes
# will be lost during system upgrades. Instead use /etc/profile.local for
# your local settings, favourite global aliases, VISUAL and EDITOR
# variables, etc ...
```

Viewing the $HOME startup files

The remaining startup files are all used for the same function to provide a user-specific startup file for defining user-specific environment variables. Most Linux distributions use only one or two of these four startup files :

- $HOME/.bash_profile
- $HOME/.bashrc
- $HOME/.bash_login
- $HOME/.profile

Notice that all four files start with a dot, making them hidden files (They don't appear in a normal ls command listing).

Because they are in the user's HOME directory, each user can edit the files and add his or her own environment variables that are active for every bash shell session they start.

```
yves@linsrv1:~> cat .profile
# Sample .profile for SuSE Linux
# rewritten by Christian Steinruecken <cstein@suse.de>
#
# This file is read each time a login shell is started.
# All other interactive shells will only read .bashrc; this is particularly
# important for language settings, see below.
```

2. Understanding the interactive shell process

If you start a bash shell without logging into a system (If you type bash at a CLI prompt, for example), you start what's called an interactive shell.

Interactive login shell (ssh from somewhere else)

```
yves@linsrv1:~> cat .bashrc
# Sample .bashrc for SuSE Linux
# Copyright (c) SuSE GmbH Nuernberg

# There are 3 different types of shells in bash: the login shell, normal shell
# and interactive shell. Login shells read ~/.profile and interactive shells
# read ~/.bashrc; in our setup, /etc/profile sources ~/.bashrc - thus all
# settings made here will also take effect in a login shell.
#
# NOTE: It is recommended to make language settings in ~/.profile rather than
# here, since multilingual X sessions would not work properly if LANG is over-
# ridden in every subshell.
```

The interactive shell doesn't act like the login shell, but it still provides a CLI prompt for you to enter commands.

If bash is started as an interactive shell, it doesn't process the /etc/profile file. Instead, it only checks for the .bashrc file in the user's HOME directory.

```
linsrv1:~ # su - yves
yves@linsrv1:~> bash
yves@linsrv1:~> ps -ef | grep bash
root     13809 13804  0 12:44 pts/0    00:00:00 -bash
yves     14113 14112  0 13:31 pts/0    00:00:00 -bash
yves     14153 14113  0 13:32 pts/0    00:00:00 bash
yves     14174 14153  0 13:32 pts/0    00:00:00 grep --color=auto bash
```

The startup files are sourced, not executed

Source and .

Source executes a file in the current shell and preserves changes to the environment

. is the same as source

Interactive non-login shell (calling bash from the commandline)

Retains environment from login shell

~/.bashrc

Shell levels seen with $SHLVL

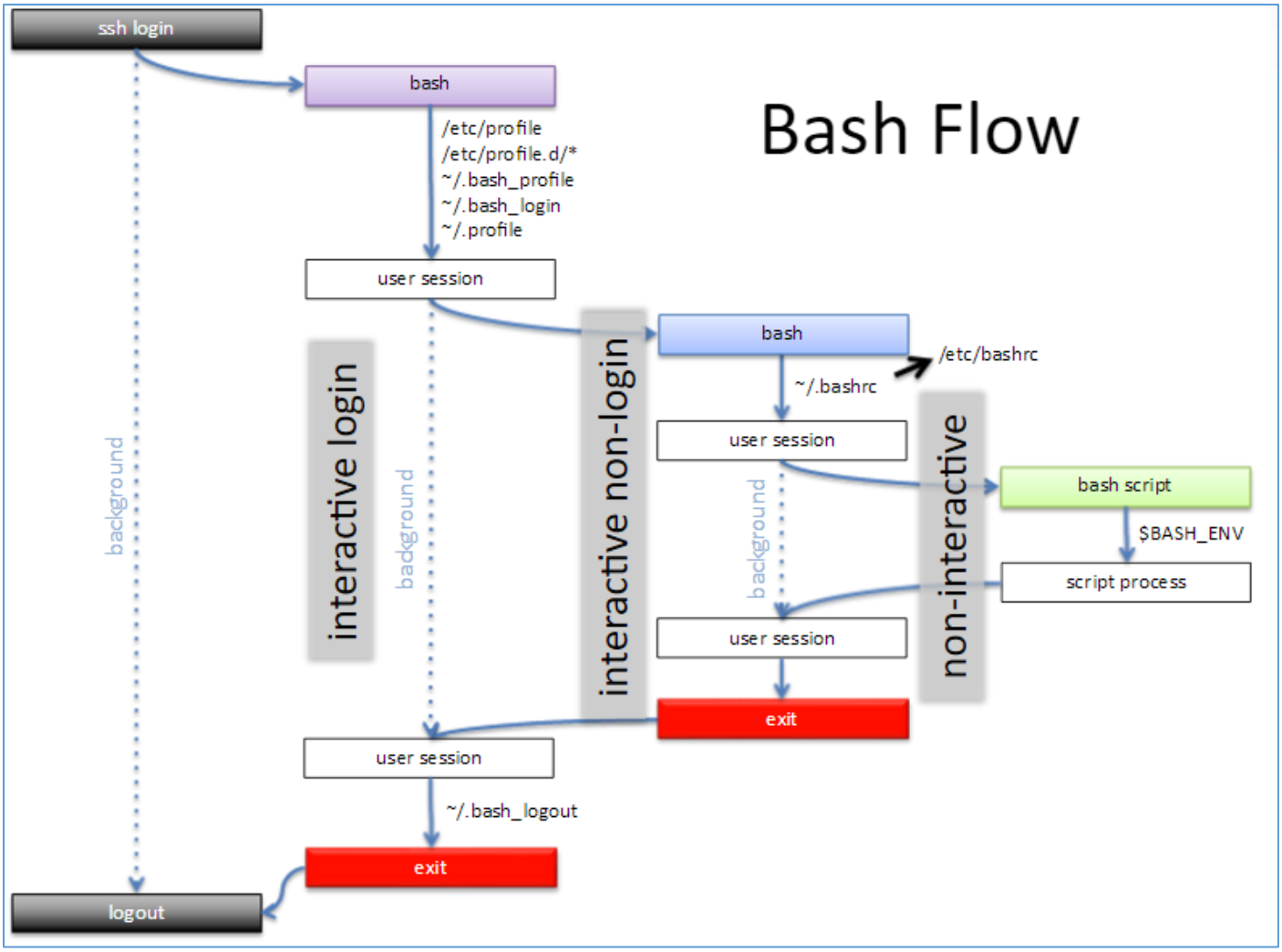
```
linsrv1:/etc # su - oracle
oracle@linsrv1:~> echo $SHLVL
1
oracle@linsrv1:~> bash
oracle@linsrv1:~> echo $SHLVL
2
oracle@linsrv1:~> bash
oracle@linsrv1:~> echo $SHLVL
3
```

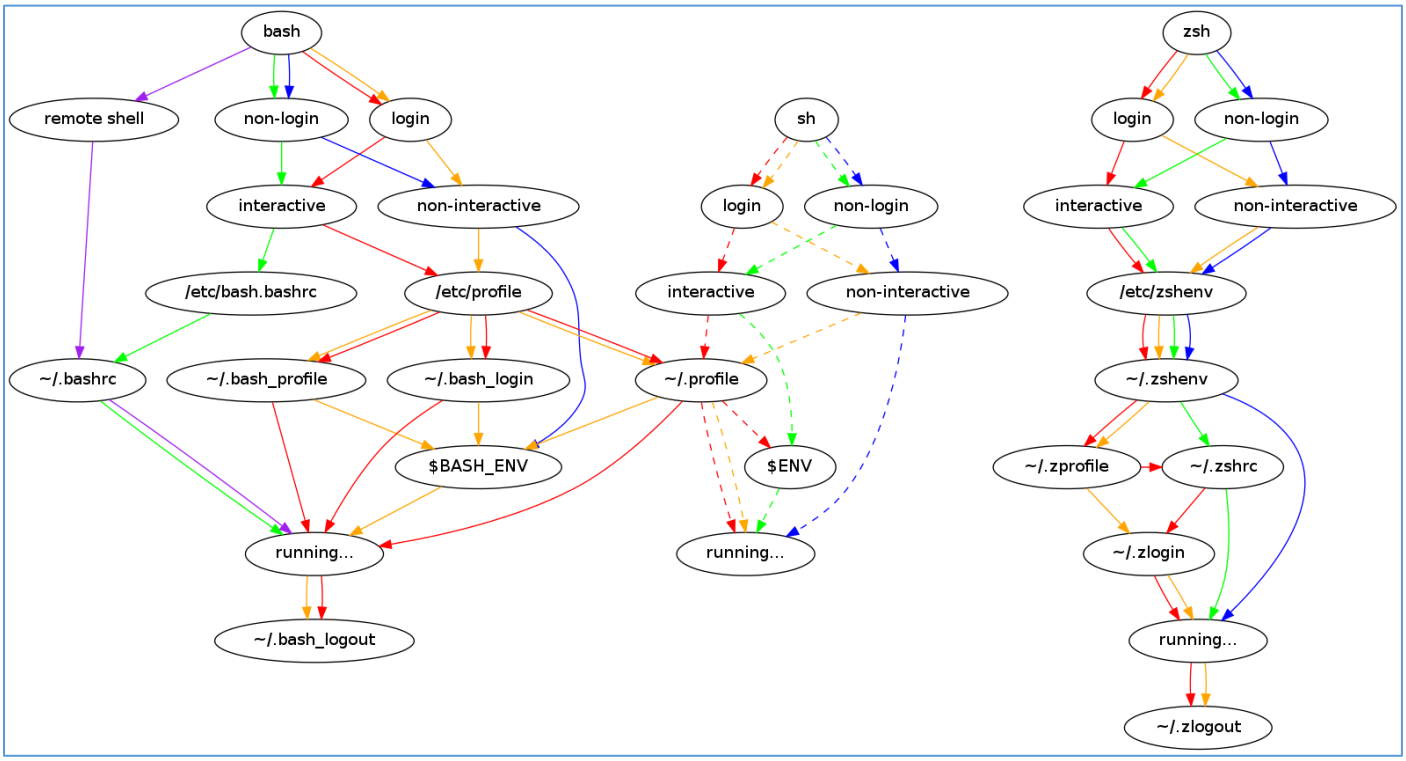
3. Understanding the non-interactive shell process

The last type of shell is a non-interactive subshell. This is the shell where the system can start to execute a shell script. This is different in that there isn't a CLI prompt to worry about. However, you may want to run specific startup commands each time you start a script on your system.

To accommodate that situation, the bash shell provides the BASH_ENV environment variable. When the shell starts a non-interactive subshell process, it checks this environment variable for the startup fi le name to execute. If one is present, the shell executes the fi le's commands, which typically include variables set for the shell scripts.

Retains environment from login shell $BASH_ENV (if set) Set to a file like ~/.bashrc


Bash Flow
ssh login
bash
/etc/profile
/etc/profile.d/*
~/.bash_profile
~/.bash_login
~/.profile
user session
background
interactive login
background
interactive non-login
bash
/etc/bashrc
~/.bashrc
user session
background
non-interactive
bash script
$BASH_ENV
script process
user session
exit
user session
~/.bash_logout
exit
logout


bash
remote shell
non-login
login
interactive
non-interactive
/etc/bash.bashrc
/etc/profile
~/.bashrc
~/.bash_profile
~/.bash_login
$BASH_ENV
running...
~/.bash_logout
sh
login
non-login
interactive
non-interactive
~/.profile
$ENV
running...
zsh
login
non-login
interactive
non-interactive
/etc/zshenv
~/.zshenv
~/.zprofile
~/.zshrc
~/.zlogin
running...
~/.zlogout

File		Login shells	Interactive, non-login shells	Intended Use
/etc/profile	Sources application settings in /etc/profile.d/*.sh and /etc/bash.bashrc	Yes	No	Global Useful Environment Variables System wide initialization file
~/.bash_profile	Per-user, after /etc/profile. If this file does not exist, ~/.bash_login and ~/.profile are checked in that order	Yes	No	User Specific Environment Variables
~/.bash_logout	After exit of a login shell	Yes	No	
/etc/bash.bashrc	Depends on the -DSYS_BASHRC="/etc/bash.bashrc" compilation flag. Sources /usr/share/bash-completion/bash_completion	No	Yes	System wide per-interactive-shell startup file This is a non-standard file which may not exist on your distribution. Even if it exists, it will not be sourced unless it is done explicitely in another start-up file
~/.bashrc	Per-user, after /etc/bash.bashrc	No	Yes	User specific aliases, shell functions, and shell options

Login, Interactive shell

The shell reads these files in sequence: (system wide) /etc/profile; (per user) ~/.bash_profile; ~/.bash_login; ~/.profile; (when logout) ~/.bash_logout.

Login, Non-Interactive shell

Login shells can be non-interactive when called with the `--login` argument.
The shell reads these files in sequence: (system wide) /etc/profile; (per user) ~/.profile; (when logout) ~/.bash_logout.

Non Login, Interactive shell

The shell reads these files in sequence: (per user) ~/.bashrc; (when exit) ~/.bash_logout.

Non Login, Non Interactive shell

The shell starts by executing $BASH_ENV and exits with reading ~/.bash_logout.
While interactive, non-login shells do **not** source `~/.bash_profile`, they still inherit the environment from their parent process (which may be a login shell).

Configuring your login sessions with dot files

The way your system behaves with respect to the reading of "dot files" at login time, setting up aliases, setting up environment variables, and so on is all highly dependent on how you actually log in.

Console logins

Let's start with the simplest configuration: a local login on the Linux text console, without any graphical environment at all. In this case, when you boot the computer, you eventually see a "login:" prompt. This prompt is produced by a program called `getty(8)` which runs on the tty (terminal) device.
When you type a username, getty reads it and passes it to the program called `login(1)`.
`login` reads the password database and decides whether it needs to ask you for a password. Once you've provided your password, login `exec(2)`s your login shell, bash.

Now, since bash is being invoked as a login shell (with name "-bash", a special ancient hack), it reads `/etc/profile` first. On Linux systems, this will typically also source every file in `/etc/profile.d` as required by the Linux standard base. (Generally `/etc/profile` should include code for this.)
Then it looks in your home directory for `.bash_profile`, and if it finds it, it reads that. If it doesn't find `.bash_profile`, it looks for `.bash_login`, and if it doesn't find that, it looks for `.profile` (the standard Bourne/Korn shell configuration file).Otherwise, it stops looking for dot files, and gives you a prompt.

Many Linux systems also have another layer called PAM which is relevant here. Before "execing" bash, login will read the `/etc/pam.d/login` file (or its equivalent on your system), which may tell it to read various other files such as `/etc/environment`. Other systems such as OpenBSD have an `/etc/login.conf` file which controls resource limits for various classes of user accounts. So you may have some extra enviroment variables, process limits, and so on, before your shell reads `/etc/profile`.

You may have noted that `.bashrc` is not being read in this situation. You should therefore **always** have `source ~/.bashrc` at the end of your `.bash_profile` in order to force it to be read by a login shell. If you use `.profile` instead of `.bash_profile`, you additionally need to test if the shell is bash first:

```
# .profile
if [ -n "$BASH" ] && [ -r ~/.bashrc ]; then
   . ~/.bashrc
fi
```

Why is `.bashrc` a separate file from `.bash_profile`, then? There are a couple reasons. The first is performance -- when machines were extremely slow compared to today's workstations, processing the commands in `.profile` or `.bash_profile` could take quite a long time, especially on machines where a lot of the work had to be done by external commands (before Korn/Bash shells). So the difficult initial set-up commands, which create environment variables that can be passed down to child processes, are put in `.bash_profile`. The transient settings and aliases/functions which are not inherited are put in `.bashrc` so that they can be re-read by every new interactive shell.
The second reason why `.bashrc` is separate is due to work habits. If you work in an office setting with a terminal on your desk, you probably login one time at the start of each day, and logout at the end of the day. You may put various special commands in your `.bash_profile` that you want to run at the start of each day, when you login -- checking for announcements from management, etc. You wouldn't want those to be done every time you launch a new shell. So, having this separation gives you some flexibility.

Let's take a moment to review. A system administrator has set up a Debian system (which is Linux-based and uses PAM) and has a locale setting of `LANG=en_US` in `/etc/environment`. A local user named pierre prefers to use the `fr_CA` locale instead, so he puts `export LANG=fr_CA` in his `.bash_profile`. He also puts `source ~/.bashrc` in that same file, and then puts `set +o histexpand` in his `.bashrc`. Now he logs in to the Debian system by sitting at the console. `login(1)` (via PAM) reads `/etc/environment` and puts `LANG=en_US` in the environment.

Then login "execs" bash, which reads `/etc/profile` and `.bash_profile`. The `export` command in `.bash_profile` causes the environment variable `LANG` to be changed from `en_US` to `fr_CA`. Finally, the `source` command causes bash to read `.bashrc`, so that the `set +o histexpand` command is executed for this shell. After all this, pierre gets his prompt and is ready to type commands interactively.

Remote shell logins

Now let's take the second-simplest example: an ssh login. This is extremely similar to the text console login, except that instead of using getty and login to handle the initial greeting and password authentication, `sshd(8)` handles it. `sshd` in Debian is also linked with PAM, and it will read the `/etc/pam.d/ssh` file (instead of `/etc/pam.d/login`). Otherwise, the handling is the same. Once sshd has run through the PAM steps (if applicable to your system), it "execs" bash as a login shell, which causes it to read `/etc/profile` and then one of `.bash_profile` or `.bash_login` or `.profile`.
The major difference when using a remote shell login instead of a local console login is that there is a client `ssh` process running on your local system (or wherever you `ssh`-ed from) which already has its own environment variables -- and some of those may be sent to the `sshd` on the system you're logging into. In particular, it is desirable for the `LANG` and `LC_*` variables to be preserved by the remote shell. Unfortunately, the configuration files on the server may override them. Getting this set up to work correctly in all cases is tricky. (Here's an example procedure for Debian.)

Remote non login non interactive shells

Bash has a special compile time option that will cause it to source the `.bashrc` file on non-login, non-interactive ssh sessions. This feature is only enabled by certain OS vendors (mostly Linux distributions). It is not enabled in a default upstream Bash build, and (empirically) not on OpenBSD either.
If this feature is enabled on your system, Bash detects that SSH_CLIENT or SSH_CLIENT2 is in the environment and in this case source .bashrc. eg suppose you have var=foo in your remote .bashrc and you do
`ssh remotehost echo \$var` it will print foo.
This shell is non-interactive so you can test $- or $PS1, if you don't want things to be executed this way in your `.bashrc`.
Without this option bash will test if stdin is connected to a socket and will also source `.bashrc` in this case **BUT** this test fails if you use a recent openssh server (>5.0) which means that you will probably only see this on older systems.
Note that a test on SHLVL is also done, so if you do: ssh remotehost bash -c echo then the first bash will source `.bashrc` but not the second one (the explicit bash on the command line that runs echo).
The behaviour of the bash patched to source a system level bashrc by some vendors is left as an exercise.

X sessions

Let's suppose pierre (our console user) decides he wants to run X for a while. He types `startx`, which invokes whichever set of X clients he prefers. `startx` is a wrapper around `xinit`, which runs the X server ("X"), and then runs through pierre's `.xinitrc` or `.xsession` file (if either one exists), or the system-wide default Xsession otherwise. Let's suppose pierre has the command `exec fluxbox` (and nothing else) in his `.xsession` file. When the smoke clears, he'll have an X server process running (as root), and a fluxbox process running (as pierre). fluxbox was created as a child of xinit, which was a child of startx, which was a child of bash, so it inherited pierre's `LANG` and other environment variables. When pierre launches something from the window manager's menu, that new command will be a child of fluxbox, so it inherits `LANG`, `PATH`, `MAIL`, etc. as well. In addition to all of that, fluxbox inherits the `DISPLAY` environment variable which tells it what X server to contact (in this case, probably `:0`).
So what happens when pierre runs an xterm? fluxbox "forks" and "execs" an xterm process, which inherits `DISPLAY`, and so on. xterm contacts the X server, authenticates if necessary, and then draws itself on the display. In addition to `DISPLAY`, it inherited pierre's `SHELL` variable, which probably contains `/bin/bash`, so it sets up a pseudo-terminal, then spawns a `/bin/bash` process to run in it. Since `/bin/bash` doesn't start with a `-`, this one will not be a login shell. It will be a normal shell, which doesn't read `/etc/profile` or `.bash_profile` or `.profile`. Instead, it reads `.bashrc` which in our example contains the line `set +o histexpand`. So his new xterm is running a bash

shell, with all of his environment variables set (remember, they were inherited from his initial text console login shell), and his shell option of choice has been enabled (from `.bashrc`).
What we've seen so far is the normal Unix setup. Many people choose to run their systems this way, and it works well. It's also fairly simple to understand once you've been exposed to the basic concepts. If you want to change something, you know precisely what file to edit to make it happen -- aliases and transient shell options go in `.bashrc`, and environment variables, process limits, and so on go in `.bash_profile`. If you want to run various X client commands before your window manager or desktop environment is invoked (for example, `xterm &` or `xmodmap -e 'keysym Super_R = Multi_key'`), you can put them in `.xsession` before the `exec YourWindowManager` line.
However, some people like to have a graphical login (display manager), and that changes pretty much everything we've seen so far. Instead of getty and login, there's an xdm (or gdm or kdm or wdm or ...) process handling the authentication. And the biggest difference of all is that when our *dm process finishes authenticating the user, it doesn't "exec" a login shell. Instead, it "execs" an X session directly. Therefore, none of the "normal" user configuration files are read in at all -- no `/etc/profile`, no `.bash_profile` and no `.profile`. (But `/etc/environment` is still read in by PAM, assuming `/etc/pam.d/*dm` is configured to use pam_limits, as is the case on Debian.)
Let's take xdm as an example. Pierre comes back from vacation one day and discovers that his system administrator has installed xdm on the Debian system. He logs in just fine, and xdm reads his `.xsession` file and runs fluxbox. Everything seems to be OK until he gets an error message in the wrong locale! Since he overrides the `LANG` variable in his `.bash_profile`, and since xdm never reads `.bash_profile`, his `LANG` variable is now set to `en_US` instead of `fr_CA`.
Now, the naive solution to this problem is that instead of launching `xterm`, he could configure his window manager to launch `xterm -ls`. This flag tells xterm that instead of launching a normal shell, it should launch a login shell. Under this setup, xterm spawns `/bin/bash` but it puts `-/bin/bash` (or maybe `-bash`) in the argument vector, so bash acts like a login shell. This means that every time he opens up a new xterm, it will read `/etc/profile` and `.bash_profile` (built-in bash behavior), and then `.bashrc` (because his `.bash_profile` says to do that). This may seem to work fine at first -- his dot files aren't heavy, so he doesn't even notice the delay -- but there's a more subtle problem. He also launches a web browser directly from his fluxbox menu, and the web browser inherits the `LANG` variable from fluxbox, which is still set to the wrong locale. So while his xterms may be fine, and anything launched from his xterms may be fine, his web browser is still giving him pages in the wrong locale.
So, what's the best solution to this problem? There really isn't a universal one. One approach is to modify the `.xsession` file to look something like this:

```
[ -r /etc/profile ] && source /etc/profile
[ -r ~/.bash_profile ] && source ~/.bash_profile
xmodmap -e 'keysym Super_R = Multi_key'
xterm &
exec fluxbox
```

This causes the shell that's interpreting the `.xsession` script to read in `/etc/profile` and `.bash_profile` if they exist and are readable, before running xmodmap or xterm or "execing" the window manager. However, there's one potential drawback to this approach: under xdm, the shell that reads `.xsession` runs without a controlling terminal. If either `/etc/profile` or `.bash_profile` uses any commands that assume the presence of a terminal (such as `fortune` or `stty`), those commands may fail. This is the primary reason why xdm doesn't read those files by default. If you're going to use this approach, you must make sure that all of the commands in your "dot files" are safe to run when there's no terminal.
One way to do that, but still retain those commands for use when you login with `ssh`, is to protect the relevant block of code with an `if` statement. For example:
Afficher/masquer les numéros de lignes

```
## Sample .bash_profile
export PATH=$HOME/bin:$PATH
export MAIL=$HOME/Maildir/
export LESS=-X
export EDITOR=vim VISUAL=vim
export LANG=fr_CA
# Begin protected block
if [ -t 0 ]; then       # check for a terminal
  [ x"$TERM" = x"wy30" ] && stty erase ^h       # sample legacy environment
```

```
  echo "Welcome to Debian, $LOGNAME"
  /usr/games/fortune
fi
# End protected block
[ -r ~/.bashrc ] && source ~/.bashrc
```

Unfortunately, the other display manager programs (kdm, gdm, etc.) do not all use the same configuration files that xdm uses. So this approach may not work for them. You may need to consult the documentation for your display manager program to find out which file(s) you should use for controlling your sessions.