

BASH - the shell - kickstart

Nicholas Mc Guire

Distributed & Embedded System Lab, School of Information
Science & Engineering, Lanzhou University, China

October 8, 2005

Contents

1. Introduction	1
1.1. execution environment	1
1.2. builtin commands	2
1.3. history management	3
1.4. job-control	3
1.5. shell customization	3
2. Anatomy of a shell	4
3. Invoking bash	7
4. Configuration files	8
4.1. default	9
4.2. login shell	9
4.3. non-interactive mode	10
5. history control	10
5.1. history command details	12
5.2. searching history	13
5.3. modifying entries using regex	14
6. Command line editing	14
6.1. Cursor movement	14
7. VI mode	15
7.1. Completion	15
7.2. moving in the history	17
7.3. Command line editing	17
8. Job control	18
8.1. simple example	19
8.1.1. Managing background jobs	20
9. customizing bash	21
9.1. hot-key binding	21
9.2. binding your favorite	22
9.3. aliasing	23
9.4. Configuring the environment	24
9.5. checking and setting options	25

Contents

Version	Author	Date	Comment
1.0	Nicholas Guire	Mc 19 Jan 2005	First shot
1.1	Nicholas Guire	Mc 11 Feb 2005	cleanup and ex- tension
1.2	Nicholas Guire	Mc Sep 30 2005	shell background infos added

This document is (C) OpenTech EDV Research GmbH and is provided under FDL V1.2 as published by the Free Software Foundation Inc. for details refer to <http://www.gnu.org/copyleft/fdl.html>.

1. Introduction

The shell is one of the key components of the UNIX operating system, one can see the prominence of it if one looks at the invocation path from the booting system. The terminal shells are typically directly managed by the init process via entries in the `/etc/inittab` and are the primary interface that UNIX provides. Even the X-environment is spawned via an initial shell, typically the xdm startup sequence utilizes non-interactive shell-scripts to launch the graphics framework. In this intro to bash we will focus on the bash (Born Again Shell) but will also build the general context of what shells are actually about.

1.1. execution environment

A UNIX executable consists of the executable file it self and the associated libraries. Some application, notably daemons and larger applications like debuggers, may have there private configuration files that are read at startup, for simpler applications it would not only be to complicated to manage a application specific config file but it also would be too expensive at runtime. imagen `ls` would need to open, read and parse a config file every time you invoke it! For these commands the shell solves the configuration problem by setting up the necessary configuration information in the environment from which we then spawn the `ls` process. So one of the primary roles of the shell is to provide this execution environment to allow customizing the application behavior with a reasonable overhead. This configuration is provided by a set of config files, system wide and user-specific.

The environment can thus be seen as providing the following key functionality:

- configuration scratch-pad, allowing to provide configuration file mechanisms to simple applications that would not justify the overhead of an application specific configuration file.
- software caching mechanism, preventing the expensive open,read,parse,closecycle that would be needed if configuration files were the only resource of application customization information.
- provide a dynamic configuration space that allows an application to undertake runtime modifications to the environment that should not be permanent, i.e. adjust the terminal type or settings to be used during execution of a particular application.

1. Introduction

- allow for a trivial "IPC" mechanism for parameter passing by providing volatile variables that can be utilized by later spawned applications, i.e. setting of the CFLAGS in the environment variable for a later call to make.

The environment is may be the most complex aspect of a shell, it is the arrea of the geatest problems with portability and the area that can be the most confusing. how deaply the environment is built into the UNIX system can be see by looking at the mechanism of process creation, the exec system call, which has three main parameters in its general form.

- the full name of the executable (including the path) that will replace the current executables image
- the argument vector that will be accessible via argv,argc in the main routine of the spaned process.
- the environment, if not explicidly passed the environment of the parant process is inherited by default.

man 3 exec, for details.

1.2. builtin commands

The typical UNIX command will provide a well defined mechanism to do some specific job, for those where this mechanism consists only of querying some data from the operationg system and reporting it to the user, i.e. `pwd` (Print Working Directory) it would be quite an expensive task to invoke a seperate exectuable to query the operating system diectly as this information is available in the shells environemtn. For such cases typical UNIX shells provide built-in commands that mimic standard commands in syntax and response but only query shell internal data objects, which is off course much faster. As an example of a built in command one can concider the `exit` "command" shown in the next section on the anatomy of a shell.

The primary intentions of built-in comands:

- optimize by educint the number of fork/exec cycles for trivial queries
- simplify synchronisation and cleanup that is shell specific (imagin exit would be a external command to a shell, cleanup of background processes etc. would be quite complicated!

- allow for non-standard extensions which only make sense within the particular shell to be encapsulated, simplifying them considerably as the runtime environment is well defined.

Built-in commands in most cases, as noted above, mimic standard commands. In cases where they do this it is important that they show the same behavior as the standard commands. A good example of what NOT TO DO, is the `time` command in `bash`, invoked as `time CMD` it reports the execution times in system and user mode, but the other options documented in the `time` info page (`info time`) are not supported, to get the `time` command to do what the info page documents, you need to execute it as `/usr/bin/time` - this can be considered a bug in `bash`.

1.3. history management

The next thing that a shell provides us with is facilities for history management, often we need to retype commands, or re-execute them with modifications, or simply store them somewhere so that one has a minimum documentation of how one solved some problem so one can re-do some task at a later time. `Bash` has very extensive history management facilities - we will cover the basics here.

For simple commands there is no need to use elaborate command line editing, but if one wants to utilize the power of UNIX to its full extent, command lines can become quite lengthy - for these lines it is helpful if one does not have to retype everything from scratch every time but reuse parts of previous commands or modify/correcting them using command line editing facilities.

1.4. job-control

As the shell is our primary interface to the operating system, it also is there to provide some job control - although somewhat limited in scope it is useful - so after covering command line editing we will cover basic job control facilities.

1.5. shell customization

Finally we will look into customizing `bash` a bit more. One of the problems with describing the `bash` is that you can't really say how it will behave as almost anything can be configured. The description here is based on the default `bash` settings in Slackware 9.1 and may change over time - although the basics generally stay the same. Never the less it can be handy, and fun, to modify this default behavior. So the focus here on

2. Anatomy of a shell

```
/* trivial shell, (C) der.herr@hofr.at,
 * License GPL V2 (http://www.gnu.org/copyleft/gpl.html)
 * compile: gcc -O2 shell.c -o shell
 * run: ./shell
 * Note: this shell will only execute commands found in /usr/bin, and will
 *       not pass any arguments to the call.
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char ** argv){
char ** args=NULL; /* no arguments for now */
char line[100];
char cmd[128];
pid_t pid;

/* the infinite loop */
do{
printf("prompt # ");
scanf("%s",&line);

/* exit our shell */
if(!strncmp("exit",line,4)){
break;
}

/* a builtin querying our username from the environment */
if(!strncmp("whoami",line,6)){
printf("%s\n",getenv("USER"));
continue;
}
strcpy(cmd,"/bin/");
strcat(cmd,line);
pid = fork();
if(pid==0){
/* inherit current environment, no args passed */
if(execv(cmd,args)) {
/* if exec fails - we must at least exit ! */
printf("%s not found\n",cmd);

```

2. Anatomy of a shell

```
exit(-1);
}
} else {
/* wait for the child */
waitpid(pid);
usleep(1); /* our shell is too fast - ugly sync */
}
}while(1);

/* this is run when we exit */
printf("logout\n");
return 0;
}
```

To execute this shell, compile it as shown in the header of the file and execute it as `./shell`. Following is a session snapshot - note that your commandset found in `/bin` may be different than shown here

```
root@rtl14:/tmp# ./shell
prompt # pwd
/tmp
prompt # arch
i686
prompt # user
root
prompt # cal
cal not found
prompt # exit
logout
root@rtl14:/tmp# exit
```

Note that you can only execute command that are located in `/bin`, so don't be surprised that you can't do very much with this shell.

Starting at the above, do the following simple exercises, read the man page of EVERY fundiont call used above FIRST, then start cleaning up.

Simple Exercises:

- fix overflow problems

3. Invoking bash

- fix argument passing
- add explicit environment passing

...if this was no challenge to you - great ! - go for the next set of exercises below.

Advanced Exercises:

- use PATH environment instead of a single hardcoded path
- add a setenv builtin
- add an echo builtin to echo any environment variable
- provide clean filedescriptors to the commands
- add output redirection
- add background processes

challenges:

- debug why `ls` is not generating output !

3. Invoking bash

The bash shell is not only intended for interactive use but also can serve as a scripting language. Shells usually are controlled by the user, via options and configuration files, but as shells are the key interface to the system resources it is also necessary in some cases to control the overall behavior of the shell at invocation, allowing the system administrator to configure a particular behavior for a particular user.

- bash `-i` interactive mode:
This is the mode that bash comes up in by default, its a fully user-configurable interactive shell
- bash `-r` restricted mode:
Especially for embedded systems the ability to restrict the actions users may take is important, bash has this very restricted mode of operation that locks the user into one directory - the login directory. Executables that are not in the path of the user can't be executed.

- `bash -c cmd1 arg1 arg2...` - exec mode Normally this mode will be used when calls are made from other scripting languages, like a Makefile or the like, in some cases it also can make sense for embedded systems to spawn a particular application under bash control on a terminal . As an example consider the following entry in `/etc/inittab`:

```
c3:1235:respawn:/bin/bash -c top tty3
```

which would hardwire the top application to `tty3`, the difference from simply calling top directly is that bash provides a reasonable execution environment.

- `bash -P`:
Normally paths are presented the way you entered them, so if you `cd` to `/usr/src/linux` the path will appear as `/usr/src/linux` even if this is a link to `/usr/src/linux-2.4.26` - with the `-P` option physical paths are presented in links which can help prevent confusion.
- `bash -v` verbose:
Quite a noisy mode - but helpful for debugging and learning the bash.

There are more options to bash - but we feel that these few will suffice for most purposes, refer to the bash info pages or the bash man page for details.

4. Configuration files

There is a certain variability of config files when it comes to bash, the typical default is to have a `.bashrc` and a `.profile` in the users home directory and a system wide profile in `/etc/profile`. On invocation of a bash shell (that is without any options to suppress specific config files) the order in which files are read depends on the invocation conditions: bash distinguishes between

- login shells
- interactive shells
- non interactive shells

These modes have slightly different startup behavior, and this difference is a common source of problems when setting up software that depends on a particular shell environment or when testing scripts that then execute under different conditions.

4.1. default

The default config files that will be searched for are

- /etc/profile:
System wide settings. This file is under control of the administrator and allows setting system resource limitations. Changes in here affect all users.
- .bashrc:
The users settings - a good place for messing up your system, in many cases where an applications works for one user and does not work for another - this is the place to look...

For many system these are maintained by the super user, not the actual user. Generally you want to make sure that the settings are consistent on a given system or that inconsistencies are documented in .bashrc.

4.2. login shell

Login shells are a bit different than shells spawned from within a shell, i.e. windows in a graphical user environment or from within an editor window. Some settings need time and doing this at the initial login is ok, redoing it every time is irritating. Furthermore rereading all files every time you start up a new shell can cause awkward behavior if the system settings are changed in the mean time as these are not under the users control such a change would be transparent and would be hard to detect. For a login shell - the first instance of bash, which is visible by the SHLVL environment variable being equal to 1 - reads some special files.

In case the .bash_login and .bash_logout exist they are invoked together with the system wide /etc/profile.

Note that you can force such behavior by telling bash to treat a spawned shell as a login shell by passing the `--login` flag to bash.

```
piglet:~ # bash --login
\begin{verbatim}
1) /etc/profile
2) .bash_login
piglet:~ # exit
logout
1) .bash_logout
piglet:~ #
```

In case they do not exist (which is the default on most Linux distributions) then the files invoked change - which is a common source of confusion:

```
piglet:~ # bash --login
1) /etc/profile
2) .profile
3) .bashrc
piglet:~ #
```

Note that by default no script file is called on user exit.

The issue to keep in mind here is that adding the login/logout specific files your `.bashrc` and `.profile` are no longer read - so the default in the `.bash.login` should always be to reference these files and then do anything else necessary.

4.3. non-interactive mode

Non-interactive mode, referred to as shell-scripts, does not call any of the startup files, note that this also applies to invocation of bash with the `-c` flag.

5. history control

The history of the bash shell is controlled by two environment variables:

- **HISTSIZE:**
Size of the history that will be recorded in the current bash session, this can be set at runtime of the shell, taking effect immediately, i.e.

```
root@rtl29: # export HISTSIZE=2
root@rtl29: # history
 512 export HISTSIZE=2
 513 history
root@rtl29: # export HISTSIZE=2000
```

This is a simple way of truncating the old history without removing the current history entries.

- **HISTFILESIZE:**
The size of the history that is actually stored in the

```
~/.bash_history
```

, is dependant on the HISTSIZE setting in the first place, but will be truncated to the HISTFILESIZE when exiting a shell, which actually commits the history to the

```
~/.bash_history
```

file.

- HISTFILE:

One can redefine the filename for storing the history, this is generally not needed, but if one wants to ensure that a session log is not mixed with other shell sessions it can be useful to place the history of a particular session into a specific file. To reload that particular history file you can do:

```
root@rtl29: # export HISTFILE=debug_session
root@rtl29: # gdb
root@rtl29: # exit
```

And to reload the sessions history you would do:

```
root@rtl29: # export HISTFILE=debug_session
root@rtl29: # bash
```

making the history available in the current bash session.

As writing real documentation during work is typically a problem, this history file can help by at least providing a minimum procedural listing how you accomplished a specific task.

The current history can be listed using the bash builtin history command, and can be cleared using history -c

```
root@rtl14: # history
514 ls
515 history
root@rtl14: # history -c
root@rtl14:
```

Again a minimum log of a setup task could be achieved by simply doing:

```
root@rtl29: # history > tftpsetup.log
root@rtl29: # history -c
```

This will not replace documentation - but it often is enough to prevent the "I know I had this solved a few months ago..." effect.

To query the current history number there are few variants possible, bash defines an environment variable `$HISTCMD` which provides the current history number of the current command to be executed.

With the history command you get a enumerated list of previous commands, to reinvoke one of the commands listed you can use bashs event operator `!`, and pass it the number of the command to recall.

```
root@rtl14: # history
 514 ls
 515 history
root@rtl14: # !514
ls
  output from the ls command
root@rtl14:
```

This will list the command being executed first followed by the regular output of `ls`.

5.1. history command details

One problem with the above history record can be that there can be more than one instance of a command installed on the system, typically this is the case for development systems with cross compilers or the like. This can then lead to subtle confusion if one team member is using `/usr/bin/gcc` and the other is using `/opt/dev_kit/bin/gcc`. To make sure you have the real record of the commands you used you can use the command `hash`. Without any arguments it prints out a hash table of the commands you have been using during the session, the point is that it contains the full path. So:

```
root@kanga:~/make # hash
hits    command
  1     /usr/bin/gcc
  3     /usr/bin/info
  1     /bin/ls
  6     /usr/bin/vi
```

```
5  /usr/bin/ssh
4  /usr/bin/make
3  /usr/bin/lynx
1  /usr/bin/man
```

This added to the history file should be enough to collect the most important procedural infos together at a later point in time !

5.2. searching history

To search through the history in bash's default emacs mode you can use the `<CNTRL>-r` hot-key - note though that this is only the default bash setting and can be changed with the `bind` command (see below).

`<C>-r` reverse index search

The reverse index search will prompt you for a string to identify the command to search for, while typing in the first command matching will be presented, one can continue entering further characters to specify the command more precisely or use `<C>-r` again to search up through the history prompting only those history entries that match the line entered up to this point.

```
root@rtl14: # history
522 ls
523 pwd
524 history
root@rtl14: # <CNTRL>-r
(reverse-i-search)''
```

typing in `s` would yield:

```
(reverse-i-search)'s' history
```

hitting `<CNTRL>-r` again would move to up the history to the next command containing an `s` in it and find `ls`.

A somewhat reduced variant of this reverse index search is to specify a string on the command line using the event operator `!` of bash and specifying the string to search for between question marks, like:

```
piglet:~ # !?ri?
```

to recall last command containing a `ri`

5.3. modifying entries using regex

The next step in history management is to use regular expression to modify specific history entries, this makes no sense for a command like `ls` in general, but once you end up with reasonably complex command lines it is a useful option. The event operator is used with an index in the example, so we are recalling the last command we typed in when calling `!-1`.

```
piglet:~ # find . -name xilinr_*.c -exec ls -l {} \;  
piglet:~ # !-1:s/nr/nx/
```

recall last command replacing `nr` by `nx`. Note that the last command can also be abbreviated by `!!`.

6. Command line editing

Command line editing is something that takes time to get used to - once one has it in the fingers it speeds up things quite a bit - in general when developers move to the command line they are doing too much typing at first. Aside from shell expansion and history usage, command line editing is the next most useful speedup. Bash comes up in emacs mode by default, so we will cover the emacs movements first - the mode is called emacs mode because the key bindings are more or less the same as found in the emacs editor. Basically if you decided to use emacs as your default editor then its a good idea to use emacs bindings in the shell, if you are using vi then using vi bindings makes more sense (covered in the next section) - if you are using some other editor then its really up to you what key bindings to use (but there is no need to use anything else but vi any way :)

6.1. Cursor movement

bash by default is in so called emacs mode - in this mode you can use standard emacs command sequences to move and edit the command line history.

```
<C-f> forward one  
<M-f> forward one word  
<C-b> back one  
<M-b> back one word  
<C-d> delete current cursor position
```

<C-_> Undo an edit command
<C-a> move to start of line
<C-e> move to end of line

To clear the screen and command buffer you can use the following key strokes:

<C-l> clear screen and reprint command
<C-c> clear the command buffer

Notice how `␣C-f␣` moves forward a character, while `␣M-f␣` moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

”Killing” text means to delete the text from the line, but to save it away for later use, usually by ”yanking” (re-inserting) it back into the line. Kill commands:

<C-k> kill from current to end of line
<C-w> kill to beginning of word
<M-INS> insert pattern buffer

7. VI mode

If you switch bash into vi mode then you are editing the history as if the commands were all in a file, one on each line. So you have the full set of editing capabilities and cursor movement capabilities that you find in vi to edit the command line. Note that this does not apply to the vim extensions though.

To set bash into vi mode use the set command:

```
set -o vi
```

7.1. Completion

Completion in vi mode is available for a number of different objects the shell is managing:

- <TAB> : default command expansion if its the first string on the command line, for all other strings file expansion is assumed.

7. VI mode

- `$`: to expand environment variables type in a `$` followed by the first few characters of the environment variable of interest then hit the tabulator to expand - if not unique - hitting the tabulator again will list all possibilities
- `@`: Rarely used host name expansion - this expansion is based on the host name that are known to the system without using DNS - basically this means using `/etc/hosts` - although bash does allow to override this by providing a private host file for frequently used host by doing:

```
root@rtl14: # export HOSTFILE=/home/hofrat/myhosts
```

The format of myhosts is just the same as for `/etc/hosts`:

```
192.168.1.42    rtl22    rtl22.hofr.at
```

So now typing in

```
root@rtl14: # ssh -l root@rt
<TAB>--<TAB>
@kanga    @kanga.hofr.at    @rtl22    @rtl22.hofr.at
root@rtl14: # ssh -l root@rtl22.
<TAB>
root@rtl14: # ssh -l root@rtl22.hofr.at
```

- `~`: User name expansion - the user names are based on entries in `/etc/passwd`
- `cmd`: a command can be expanded by typing in the first few characters of the command and then hitting the tabulator - if the string is not yet unique then hitting the tabulator again will give a list of possibilities. hitting the tabulator twice on an empty command line will give you all the possibilities known for that users environment (note that the available commands are only those that are found in your default path settings - configured in the `PATH` environment variable.
- `file` : file expansion will be done if the sting is not the first object on the command line - by hitting the tabulator a filename that matches will be completed, or if multiple possibilities exist hitting tab a second time will list them.

Note that this is the default behavior and again this can be overridden in bash- but this is rarely useful. Completions can also be listed by using the hot-keys bash provides - see `bind -p | grep comple`.

7.2. moving in the history

- jk - move up down
- hl - move left right
- e - move to end of next word
- b - move to beginning of word
- w - move to next word
- \$ - move to end of line
- ^ - move to beginning of line

Note that you also can use the upper case word movement characters to move through i.e. paths that have no blank. So using upper case B,W and E would jump to the end of `/usr/src/linux]` were as b,w and e would treat `usr`, `src` and `linux` as separate words.

7.3. Command line editing

- a - append
- i - insert
- r - replace current character
- A - append at end of line
- I - insert at beginning of line
- R - switch to replacement mode

Again this is the same as in vi.

- dd - delete entire line
- db - delete one word back
- dw - delete one word forward

M-? - list possible completion
<TAB><TAB> - list possible completion

Brace expansion

ls {a,b}.*

8. Job control

Job control is limited, and as Erick Raymond put it in "The Art of UNIX programming" its botched. Well it's botched but it's useful any way. It's simplicity is also a result of the resource constraints of systems at the time UNIX was designed, for such systems elaborate background job control would have been only of limited utility. A further reason for job control being limited is simply that users don't have much demand for background processing. It takes some time to really utilize the capabilities of background jobs and not simply stand up every time one has a large task to run and go for a cup of coffee...

- start with &
- push to background with <CNTRL>-Z
- fg to pull to foreground
- bg to push
- jobs to list jobs

- fg %# to pull job number
- fg PID to pull by pid
- fg %string to pull by string

Especially in Linux much of the functionality of running background jobs has been replaced by simply having many consoles or terminals under X available, thus the demand for background processing is limited, nevrerr the less it pays off to have some basic background processing know how, especially for rremote sessions where multiple temrinals are more complicated than utilizing background jobs.

8.1. simple example

Starting and stopping background jobs

```
piglet:~ # ls -lR / > /dev/null 2>&1
<CNTRL>-Z
[1]+  Stopped                  ls $LS_OPTIONS -lR / >/dev/null 2>&1
piglet:~ # bg
[1]+  ls $LS_OPTIONS -lR / >/dev/null 2>&1 &
piglet:~ # fg
ls $LS_OPTIONS -lR / >/dev/null 2>&1
<CNTRL>-C
piglet:~ #
```

We launched a process (`ls -lR`) redirecting standard out and standard error to `/dev/null`. By hitting `<CNTRL>-Z` the process is stopped and we get back the shell prompt. At this point the process is technically a background process, but it is inactive. By now pushing it into the background from the shells perspective with the `bg` (BackGround) built-in command, it is continued (by sending it a signal `SIGCONT`) and is now a real background process running concurrently to our foreground shell interpreter. To now get it back into the foreground we can use the `fg` (ForGround) command and then terminate it.

Things to note:

- The command was launched with `ls -lR...` but is reported as `ls $LS_OPTIONS -lR...`, we can see that the shell added some `ls` specific environment variables before actually forking the requested `ls` process.
- The command reported stopped after hitting `<CNTRL>-Z` is reported without the trailing `&` sign which would indicate that it is a background process for the shell. So at this point we can see that although this stopped process is in the background the shell does not see it as a background process yet.
- after pushing it to the background with `bg` it is extended with the `&` to indicate it is now a shell background process.

8.1.1. Managing background jobs

In this paragraph we describe managing multiple background processes. First we start a few processes, this time directly in the background by appending the & to the command line. Note that a background command invocation responds with a number in square brackets followed by the PID of the process. The number of the background process is only valid within this specific shell session, while the PID is of course valid on a system scope.

```
piglet:~ # ls -lR / > /dev/null 2>&1 &
[1] 7775
piglet:~ # ls -alR / > /dev/null 2>&1 &
[2] 7776
piglet:~ # jobs
[1]-  Running                  ls $LS_OPTIONS -lR / >/dev/null 2>&1 &
[2]+  Running                  ls $LS_OPTIONS -alR / >/dev/null 2>&1 &
piglet:~ # fg 1
ls $LS_OPTIONS -lR / >/dev/null 2>&1
<CNTRL>-Z
[1]+  Stopped                  ls $LS_OPTIONS -lR / >/dev/null 2>&1
piglet:~ # bg
[1]+  ls $LS_OPTIONS -lR / >/dev/null 2>&1 &
piglet:~ #
```

Note that the numbers of background jobs are kept but the current task indicator is reassigned to the first background job (+ indicating more recent). Also we can see that although we called `ls -lR /` we are presented with the "true" background command running which is `ls $LS_OPTIONS -lR /` again.

As noted above the usability of background jobs and job control is somewhat limited and has been replaced by simply running multiple terminals for interactive sessions. For shell scripting background jobs offer some interesting capabilities of shells for building protocols and error handling routines, we will see that in the later shell scripting intro when building shell-script based network applications.

9. customizing bash

We will not cover the full extent of customization here - but only those parts that we found useful for developers - there is more - see ??. The message here should be - learn your shell if you want the full power of UNIX unleashed on your PC !

9.1. hot-key binding

We noted that there are some hot keys like <CNTRL>-r - these are not hard wired in bash but they are the interface to the readline library functions mapped with the bind bash builtin command.

```
root@piglet: ~# pind -p
...
"\M-b": backwards-word
...
# vi-yank-to (not bound)
...
"\C-y": yank
"\e[2~": yank
"\e.": yank-last-arg
"\e_": yank-last-arg
"\e\C-y": yank-nth-arg
"\ey": yank-pop
```

The key binding are described by:

- Control key :

\C

and they key to be pressed while holding the control key - i.e.

"\C-y"

stands for <CNTRL>-y .

- Escape key:

`\e`

standing for `<ESC>` - so `<ESK>`. will yank the last argument and put it into the edit buffer.

- Meta key:

`"\M-b"`

- Function key:

`"\e[2~"`

- is equivalent to `<ESC>-<F2>` . Note that you can't verbatim type in the character sequence consisting of open square braken followed by 2 followed by tilde, but you simply hit `<ESC>-<F2>` when you want to assign a command to that key sequence - don't worry if the console beeps at you!

- numeric key values: although not really that useful in every day work, `bind -p` will also list some binding as

`"\235"`

, which is the binding for this specific key value.

So if some command listed in here in the command-line editing section does not work on your system then `bind -p` will tell you why - and what command to actually use !

9.2. binding your favorite

`bind` is not only there to list bindings but also to set bindings and that can be quite useful when you are doing jobs that require typing in the same thing over and over again. To now bind a command to a hot key you use the `bind -x` command.

```
root@rtl14: # bind -x '"\C-v":ls'
<CNTRL>-v
docs/          profile/
```

More complex bindings are possible - once can map key sequences like

```
root@rtl14: # bind -x '"\C-v\C-v":ls -l'
<CNTRL>-v-v
total 1109
-rw-r--r--  1 hofrat  users          133 Feb 11 10:26 1
drwxr-xr-x  2 hofrat  users          104 Feb  6 15:12 CVS
...
```

Note though that although you code it as <CNTRL>-v-<CNTRL>-v you only hold down the <CNTRL> key and hit v two times in a row - for keys bound to <CNTRL>-<ALT> you would use:

```
root@rtl14: # bind -x '"\C-\M-v":ls -al'
<CNTRL>-<ALT>-v
total 1109
drwxr--r--  1 hofrat  users          133 Feb 11 10:26 .
drwxr-xr-x  1 hofrat  users          133 Feb 11 10:26 ..
...
```

One can naturally over due shell configuration - one important thing to keep in mind is that shell configuration should not have any side effects on applications that are used in a team of developers - but hot keys for convenience should be ok.

```
root@rtl14: # bind -x '"\x23":ls'
#
doc/          uml/
```

would re-bind the # key to execute ls - effectively taking it off the keyboard for the shell.

9.3. aliasing

Bash allows to alias commands if they are too long or a option is used all the time. Typically one would alias commands for testing and if the work as needed ad the alias to your .bashrc so that it is present the next time you login to the system.

```
root@rtl14: # alias ls='ls -a'
root@rtl14: # ls
.                .uml             .xfce4           kermrc
...
```

To view what aliases are set on your shell - and normally there are a few set all ready - use the alias bash builtin command with no argument:

```
root@rtl14: # alias
alias += 'pushd .'
alias -= 'popd'
alias ..='cd ..'
alias ls='ls -a'
alias beep='echo -en "\x07"'
```

Note again that aliases - just like hot keys - should not have side effects that will affect the collaboration within a team - so setting :

```
root@rtl14: # alias gcc='gcc -g -O2'
```

Could be a real problem during development as you would be typing in gcc as your colleagues due but the effect is obviously not the same.

9.4. Configuring the environment

The default environment that bash came up with when you logged in was configured using the configuration files noted in section 2. Aside from using configuration files you also can use the export command to make a environment variable available or to modify an existing variable.

```
root@rtl14: # echo $PATH
/bin:/usr/bin:/usr/local/bin
root@rtl14: # export PATH=/bin
root@rtl14: # echo $PATH
/bin
root@rtl14: export PATH=/sbin:$PATH
root@rtl14: # echo $PATH
/sbin:/bin
```

Note that this can be a critical issue for a development team again - environment variables may not fall apart within a team - if one team member has a different path setting than the other then they might both call gcc but one is using a different version depending on which is found first in the PATH variable. This naturally is true for many other environment variables available by default.

Aside from setting/resetting/modifying existing variables, you can create new variables to be used then by applications or by the shell itself:

```
root@rtl14: # export TESTFLAGS=-g3
root@rtl14: # gcc $TESTFLAGS hello.c -o hello
```

If though a variable should only be active for one command then one would not want to export it to the shells environment, where it is effective for all subsequent commands, but one would pass it on the command line to the particular command:

```
root@rtl14: # CFLAGS=-g3 make hello
cc -g3 hello.c -o hello
```

As make will use the variable CFLAGS internally we can pass it via the environment to this invocation of make without it taking effect for later commands that might also be using this variable. In this case CFLAGS will be passed in the environment of make - via the envp to the exec call - but it only has an effect if make actually reads the variable, which is the case for CFLAGS (note that for the above example you don't need to have a Makefile in place only a hello.c file is needed).

9.5. checking and setting options

In the section above we set environment variables - these are able to influence the behavior of the shell - but beyond that they influence the behavior of any application that reads them. Bash also has some internal configuration parameters that can be set in a private manner - not making them take effect in other applications. We have two bash builtins to manage these bash specific settings.

shopt

shopt allows you to set up the shells options.

- `shopt`:
List all settings of the current bash session. If a `optname` is passed as argument then the state of this one option is listed.
- `shopt -s optname`:
set this option to on:


```
root@rtl14: # shopt mailwarn
mailwarn      off
root@rtl14: # shopt -s mailwarn
root@rtl14: # shopt mailwarn
mailwarn      on
```
- `shopt -u optname`:
Unset the option listed (set it to off)

`set`

Bash has some arguments that can be passed at invocation - `set` allows to set some of these at runtime.

```
root@rtl14: # set -x
root@rtl14: # ls
+ /usr/bin/ls --color=auto -F -b -T 0
852/          fs_opt/          hello.c
...
root@rtl14: #
```

so this `set` has the same effect as invoking `bash -x` on the commandline. For a full list of `set` options see `man(1) bash`.

References

- [GNU] GNU not UNIX,
<http://www.gnu.org/>, <ftp://ftp.gnu.org/>
- [BASH] Cameron Newham, Bill Rosenblatt, learning the bash
O'Reilly & Associates. ISBN 1-56592-347-2. 1998