3 File system

The material of this section is taken from E. Nemeth, G. Snyder, S. Seebass, T. R. Hein, *UNIX System Administration Handbook* (2nd edition, Chapter 4, or 3rd edition, Chapter 5). Prentice-Hall.

4 Programs and processes

Most of the material of this section is taken from E. Nemeth, G. Snyder, S. Seebass, T. R. Hein, *UNIX System Administration Handbook* (2nd edition, Chapter 5, or 3rd edition, Chapter 4). Prentice-Hall.

Environment

Every Unix process has an *environment*:

array of strings of the form `name=value`.

in theory, `name` may contain all characters except `\0` and `=`,

in practice one should restrict to `[A-Za-z][A-Za-z0-9]*`.

Child processes inherit the environment of the parent process during a `fork()`.

Some of the variants of the `exec()` system call have a new environment as the last argument.

The environment can also be manipulated during the life time of a process, e. g., using the C library functions `setenv` or `putenv`.

In a shell, the command `env` can be used to display the current environment.

Common examples of environment variables:

`USER`, `LOGNAME`: name of the logged-in user.

`HOME`: login directory of the user.

`PATH`: a list of directories separated by "`:`". in which the shell looks for executable files (see below).

`PWD`: the current working directory.

`TERM`: the type of the user's terminal or terminal emulator, such as `xterm`; used by the (`n`)`curses` library to translate symbolic cursor control commands into concrete escape sequences.

PAGER: the user's preferred utility, e. g., `less` or `more`, to display text files.

EDITOR, VISUAL: the user's preferred text editor.

$PATH

The value of the environment variable PATH is used by the shell and other programs to find executable files.

What happens when a user enters a commmand into the shell?

If the commmand is a built-in commmand of the shell, it is executed directly by the shell without forking a new process.

If the commmand starts with a slash "/", then the shell takes the command as a absolute pathname and tries to execute the corresponding file.

Otherwise, if the commmand contains a slash "/", then the shell takes the command as a pathname relative to the current directory and tries to execute the corresponding file.

Otherwise, the shell searches in all directories of $PATH for an executable file with the given name; the first one it encounters is executed.

The current directory is not automatically included in the PATH. In fact, putting "." into the PATH, in particular at the front, is a security hole: The functionality of the `ls` program in the current directory can be very different from the functionality of `/bin/ls`, which the user intended to execute.

Note: An empty path element, e. g., before the first colon in "`:/usr/bin:bin`" may be considered as equivalent to ".".

exec Revisited

The `exec` system call is used to overwrite the code of the current process with the contents of a specified file (and to reset the data and stack segments). When `exec` loads the file, it first looks at its first few bytes, the *magic number*. The magic number determines the type of the file: whether it is executable at all on the given machine, whether it is statically or dynamically linked, etc.

A special case of a magic number is the character sequence "`#!`" ("shebang") followed by the name of an interpreter, say `/bin/sh`. If the file `/home/tom/bin/myprog` starts with `#!/bin/sh`, then the kernel translates

```
/home/tom/bin/myprog 123
```

into

```
/bin/sh /home/tom/bin/myprog 123
```

Note that `/bin/sh` ignores the first line of `/home/tom/bin/myprog`, since "`#`" is a comment character for the shell.

This mechanism explains an important difference between scripts and proper executable files. Since the interpreter must read the script, the read-bit of the script must be set in addition to the execute-bit: File permissions "`---x--x--x`" are sufficient for proper executable files, but not for scripts.

What happens if an interpreted script has the suid-bit set? It depends. On some systems, the suid-bit for scripts is simply ignored. Some systems propagate the suid-bit to the call of the interpreter. This creates a giant security hole, however: Suppose that a malicious user has write-permissions in some directory that is located on the same file system as the suid-script `/usr/bin/rootscript`, e. g., in `/tmp`. Then this user can create a hard link from `/usr/bin/rootscript` to `/tmp/myscript` (which has now the same permission bits as `/usr/bin/rootscript`). The kernel will translate the command

```
/tmp/myscript
```

into

```
/bin/sh /tmp/myscript
```

and call `/bin/sh` with suid privileges. But since this action is not atomic, the user has a chance to execute

```
mv /tmp/maliciousscript /tmp/myscript
```

in between the two calls.

There is a fix for this problem: On some Unix systems, the kernel passes the file descriptor it has opened for `/tmp/myscript` to `/bin/sh`, rather than the file name `/tmp/myscript`, so

```
/tmp/myscript
```

is translated to

```
/bin/sh /dev/fd/23
```

This works only, however, if file descriptors are accessible as devices.

Job Control

Most current shells support job control, that is, the possibility to stop running processes and to move them to the foreground or the background.

To implement this feature, one needs the concept of *process groups*. Every process belongs to exactly one process group (or *job*) and every process group to exactly one *session*. Usually, all processes produced by a single user command form one process group, and all process groups stemming from a single login form one session. The terminal driver of the controlling terminal stores the number of exactly one process group, the *current terminal process group id*. Processes belonging to this group are foreground processes: they have unlimited access to the terminal. Other processes are called background processes: whenever they try to read from the terminal, they are stopped using the SIGSTOP signal. Conversely, a Ctrl-C or Ctrl-Z typed by the user is sent to all foreground processes; whereas processes whose process group does not match the current terminal process group id are unaffected.

5 Shells and shell differences

Unix Shells

Characteristics:

Both user interface and interpreted programming language.

"Glue" to combine existing programs rather than fully-fledged language (e. g. usually very few data types).

Not a privileged program or process.

One layer of a multi-layer structure: User input is preprocessed before passing it to application programs.

History:

The first Unix shell:

Thompson shell (Version 1 until Version 6, since 1971, Ken Thompson):
input/output redirection, pipes, some control structures, but no shell variables.

"The" Unix shell:

sh (Bourne shell, since Version 7, 1978, Steve Bourne)

The C shell family:

csh (C shell, BSD, seit 1979, Bill Joy):
User interface greatly improved compared to sh (job control, history, aliases, directory stack, tilde expansion, built-in arihmetic, arrays); but syntax incompatible with sh; lacking clean concepts; not recommended for programming.

tcsh (TENEX csh, Ken Greer, Paul Placeway):
Adds Emacs-style command-line editing and completion; some csh bugs fixed.

Extensions of the Bourne shell:

ksh (Korn shell, seit 1982, David G. Korn)
bash (Bourne-again shell, seit 1987, Brian Fox):
Syntax mostly compatible with sh, many features of the csh/tcsh user interface adopted.

zsh (Z shell, seit 1990, Paul Falstad):
"The union of all shells", highly configurable, extended completion features.

Others:

rc (Byron Rakitzis, based on rc for Plan 9 by Tom Duff)
es (Paul Haahr, Byron Rakitzis):
Redesign from scratch; clean, but somewhat minimalistic; es has higher-order functions.

Note: usually there are many different versions of these shells with very different features.

Under Linux, `/bin/sh` is a link to `/bin/bash`, `/bin/csh` is a link to `/bin/tcsh`.

Basic Shell Syntax

sh	csh
`cmd arg1 arg2 arg3`	`cmd arg1 arg2 arg3`
`cmd >file`	`cmd >file`
`cmd >file 2>&1`	`cmd >& file`
`cmd 3>&1 1>&2 2>&3 3>&-` [1]	
`cmd1 | cmd2`	`cmd1 | cmd2`
`cmd1 2>&1 | cmd2`	`cmd1 |& cmd2`

[1] Exchanges stdout and stderr. This is impossible in csh.

sh	csh
`cmd a* b? c[x-z]`	`cmd a* b? c[x-z] d{e,f} ~gh`
`cmd1 || cmd2`	`cmd1 || cmd2`
`cmd1 && cmd2`	`cmd1 && cmd2`
`VAR=value` [2]	`set VAR = value`
`VAR=value; export VAR` [3]	`setenv VAR value`
`echo $VAR`	`echo $VAR`
`$*, $@`	`$argv, $argv[*]`
`$2`	`$argv[2]`
`$#`	`$#argv`
`set value1 value2 value3`	`set argv = (value1 value2 value3)`
`$?`	`$status`
`read VAR`	`set VAR = $<`

`if cmd ; then` [4]
`  ...`
`else`
`  ...`
`fi`
	`if (expr) then`
	`  ...`
	`else`
	`  ...`
	`endif`

`while cmd ; do`
`  ...`
`done`
	`while (expr)`
	`  ...`
	`end`

`for i in word1 word2 word3 ; do`
`  ...`
`done`
	`foreach i (word1 word2 word3)`
	`  ...`
	`end`

[2] "Ordinary" shell variables.
[3] Environment variables.
[4] Exit status 0 (= successful) is interpreted as true; every other exit status is interpreted as false.

sh	csh
case word in	switch (word)
pattern1) ... ;;	case pattern1:
pattern2) ... ;;	...
esac	breaksw
	case pattern2:
	...
	breaksw
	endsw
cmd1 ‘cmd2‘	cmd1 ‘cmd2‘
. file	source file
trap cmd 1 2 3 15	onintr label [5]

[5] only SIGINT = kill -2 = Ctrl-C.