Chapter 4: Shells

This chapter discusses the concept of a UNIX shell, and how to manipulate shells. It includes information on the available and recommended shells and their features. The concept of a shell as an interpretive programming language is introduced.

4.1 Introduction to Shells

The kernel is the real operating system and is loaded into memory at boot time. Typically the user never interacts directly with the kernel. The utilities are programs stored on disk, and loaded into memory by the kernel when invoked.

A shell is a utility. It is run in user mode, and does not have system privileges. You have a default shell, and you can invoke other shells. Invoking shells is discussed in section 4.1.2.

The shell is the interface between the operating system and the user. It interprets the commands you type and the keys you press in order to direct the operating system to take an action. Shell scripts, which are analogous to DCL command files, allow you to use the shell as an interpretive programming language. They are introduced in section 4.4, but a comprehensive treatment of scripts is beyond the scope of this manual.

There are two families of shells: one based on the Bourne shell (this family also includes the Korn (**ksh**) and Bourne-again (**bash**) shells), and the other based on the Berkeley/C shell. The shells themselves will be discussed and compared in section 4.2.

4.1.1 Determining Your Current Shell

There are several commands available in all the shells that furnish this information. We present four examples below with sample output for **csh**. The first three, **`echo`**, **`env`** and **`finger`**, will show only your login shell. If you have invoked another shell, these commands will not reflect the new shell. **`ps`** lists information about all your active processes.

`%` **`echo $SHELL`**

displays the value of the variable name that follows the **`$`**; example output: `/bin/csh`

`%` **`env or printenv`**

shows all defined environment variables, including SHELL; example output: `SHELL=/bin/csh`

`%` **`finger`** ***your_username***

shows user information and login shell; example output:

```
Login name: username                        In real life: {your name}
Directory: /afs/fnal.gov/files/home/room3/{username} Shell: /bin/csh
```

% ps

shows processes, including shell; example output:

```
PID TTY      TIME COMD
6264 pts/11   0:03 csh
```

Note that on some of the more recent OS releases (e.g., AIX+4 and IRIX+6.4, and likely others in the near future) `/bin/sh` is a link (links are described in section 6.3.5) to the korn shell (**ksh**). **ksh** is a superset of **sh**, so this shouldn't present any problems for you. One difference is that your `.shrc` file (see section 9.4.2) gets sourced when you run `/bin/sh` scripts.

4.1.2 Starting a Shell

A shell is started by a login process. A new shell is also started for each invocation of a terminal window or shell script (see section 4.4). Which shell gets invoked is determined by the last field in your entry in the password entry file. In a standard UNIX file system you can display your password entry by the command:

% grep ^*username* /etc/passwd

(**grep** is described in section 6.4.2.; the use of **^** is explained in section 5.4.5.) To display your password entry in an NIS environment, use the command:

% ypmatch *username* passwd

(The NIS command **ypmatch** is not described in this manual.) Sample **ypmatch** output from the FNALU system, for which the default shell has been set to **csh**, looks like:

```
aheavey:!:6302:1525:Name of User:/afs/fnal.gov/files/home/room3/aheavey:/bin/csh
```

When you log in, the login process invokes a shell program (e.g., `/usr/local/bin/tcsh` or `/usr/local/bin/bash`) and transfers control to it. The shell displays a prompt indicating it is ready for your input. The default UNIX prompts are symbols that indicate which shell is invoked (recall from section 2.3 that your prompt is likely to be set differently):

- **%** for the C shell family
- **$** for the Bourne or Korn shells

On FNALU the prompts are set to indicate the host machine, for example **<fsui01>**, or **<fsgi02>**. At any point in your session you can invoke another copy of the same shell or a different shell by typing the shell name at the prompt, for example:

% csh

invokes **csh** (C shell). This new shell, or "subshell", sits on top of your current shell. The execution of the original shell is then suspended (the shell is put to sleep), and the new shell takes control. Upon quitting the new one, the original shell wakes up and resumes control.

The average user at Fermilab does not have the privilege to change the password entry file. Therefore, to change your default shell you will need to ask your system manager.

4.1.3 Exiting a Shell

To exit a shell and return to the calling shell, type **exit** at the prompt. If you repeat the **exit** command once more, your terminal emulation is closed, and the terminal window disappears. Instead of **exit** you may need to enter <Ctrl-d>.

4.2 Features of Available Shells

This section is excerpted from *Shell Choice, A shell comparison* (dated September 28, 1994) by Arnaud Taddei of CERN. His eleven-page document contains a brief description of the six major shells and provides an excellent comparison of features between the shells. It is available on the Web at `http://consult.cern.ch/writeup/shellchoice`.

Of the six major shells, four are in the Bourne family: **sh**, **ksh**, **bash**, and **zsh**; and two are in the Berkeley/C family: **csh**, **tsch**.

The most up-to-date shells are **tcsh** (Berkeley/C), and **bash** and **zsh** (Bourne). These are also the three shells that are public domain (as opposed to vendor-supported). The public domain shells are the same on all platforms, which is not true of vendor shells. This is desirable when attempting to homogenize user environments. **Note that zsh is not supported at Fermilab.**

Some of the common features of these newer shells are:

- specific startup files
- startup files are the same for any platform
- specific shell variables
- specific built-in commands

The **tcsh** is essentially an enhanced **csh**. Some additional features of the **tcsh** are:

- enhanced completion[1] mechanism (programmable for commands, file names, variable names, user names, etc.)
- multiline editing capabilities (command line editing using **emacs** or **vi**-style key bindings)
- enhanced file expression syntax
- spelling correction (see section 5.3)
- enhanced prompt
- step up/down through history list

The following table should give you an idea of the virtues of each of the shells supported at Fermilab. It is adapted from one in Taddei's document referenced above. More complete feature lists for all the shells can be found there.

++	good
+	existing
-	weak
--	absent

Criteria	sh	ksh	bash	csh	tcsh
Configurability	-	+	++	+	++
Execution of commands	+	+	+	+	++
Completion	--	+	++	+	++
Line editing	-	+	++	-	++
Name substitution	+	+	++	+	++

1. This feature allows you to uniquely specify a file without typing in its whole name.

Criteria	**sh**	**ksh**	**bash**	**csh**	**tcsh**
History	--	+	++	+	++
Redirections and pipes	+	+	+	+	+
Spelling correction	--	--	--	--	+
Prompt settings	+	+	+	+	++
Job control	--	+	+	+	+
Execution control	+	+	+	+	+
Signal handling	+	+	+	-	-

4.3 Supported/Recommended Shells at Fermilab

On many systems at Fermilab, **tcsh** is used as the default shell. The Computing Division currently supports **csh**, **tcsh**, **sh**, **bash** and **ksh**. **tcsh** or **bash** is recommended for interactive use, and **sh** for scripts. (The C shell family is not recommended for scripts due to inconsistent syntax at different levels of nesting.) **zsh** is not currently supported. From the *UNIX Resources* Web page you can access UNIX support information under **UNIX Products** to find out about any support policy changes.

4.4 Shell Scripts

As mentioned above, a UNIX shell can be used as an interpretive programming language. Besides executing shell commands within the script, you can:

- create and use variables
- process (read) arguments
- test, branch, and loop
- perform I/O

A *shell script* is a file containing a sequence of commands which can be executed by the shell, and flow control commands. The same syntax is used for commands within scripts as for interactive command entry. Section 5.1 explains briefly how the system runs and interprets shell scripts.

Although you can write complex programs using the shell language, you can also create simple shell scripts for running long commands or a series of commands that you use frequently.

In every shell script you write, include the special characters `#!` followed by the pathname of the shell as the first characters in the file. This indicates (a) that this is a script rather than a compiled executable, and (b) which shell to invoke to run the script.[1] For example:

```
#!/usr/local/bin/bash
```

1. On some of the more recent OS releases (e.g., AIX+4 and IRIX+6.4, and likely others in the near future) `/bin/sh` is a link to the korn shell (**ksh**). Therefore on these platforms, the `.shrc` file (see section 9.4.2) gets sourced for any script starting with `#!/bin/sh`.

at the start of the script invokes **bash** to run it. A `#` found anywhere else in the script is interpreted as the beginning of a comment, and the shell ignores all characters between the `#` symbol and the next newline character.

A good introductory reference for script-writing with examples can be found in *UNIXHelp* on the Web. You can get to it via the *UNIX Resources* page under **The UNIX Operating System**.

Note that in order to execute the script, regardless of shell, the script file must have execute permission for the appropriate users (see section 6.6.1 for a discussion of permissions). After you set this permission, the shell will need to rebuild its "hash table" to include the new script. The hash table is a table of executables that the shell recognizes.

To complete these two operations, enter:

`%` **`chmod a+x`** *filename*

`%` **`rehash`**[1]

To run a script, the shell must be able to locate it. Typing the full path of the filename is perhaps the most cumbersome way of telling the shell where the script is, but it will work. You may want to maintain a `$HOME/bin` directory for all your programs and shell scripts, and include this directory in your path[2]. We refer you to section 9.2 which discusses the *PATH* variable. The shell uses this variable to locate commands and other executables. You generally execute a script by typing the filename by itself, or the filename preceded by **`./`** on the command line, as shown below (The **`./`** explicitly tells the shell to look for the executable file in the current working directory.):

`%` ***filename*** or

`%` **`./`*****filename*** or

`%` **`/`*****full_path*****`/.../`*****filename***

The shell then interprets and executes the commands in the file one by one.

It is important to remember that, like all UNIX commands that are not part of the shell (see section 5.1.1 for an explanation of shell commands), the script file executes in a subshell forked[3] by the parent shell. This subshell retains any environment variables defined in the script as well as any shell variables defined in the file `.cshrc` or `.shrc` (one of these two files may be executed automatically prior to the script, depending on your shell; see section 9.4). At the end of the script, control returns to the parent shell, and any definitions made by the subprocess are *not* passed back to the parent process.

To execute a script for which you do want to pass back changes to the parent shell (for example, setting new shell variables), the syntax for execution differs. For the C shell family, execute the script by typing:

`%` **`source`** *filename*

For the Bourne shell family, type:

`$ .` *filename*

For instance, after you make changes to your `.cshrc` or `.login` file, you can use **`source`** or `.` to execute it from within the login shell in order to put the changes into effect.

---

1. The command in **sh** is **`hash`**; not necessary in other shells.
2. Under FUE, the Fermi files add your `/bin` directory to your PATH.
3. Under UNIX, the term *fork* means create a new process.

4.5 Other Interpretive Programming Languages

We have mentioned that each UNIX shell can be used as the interpreter for its own programming language. Other interpretive languages supported at Fermilab are **perl** (provided in the FUE **shells** product), and **gawk** (a version of **awk**; see section 5.4.4). These languages are beyond the scope of this manual. The O'Reilly & Associates, Inc. publishers provide excellent reference texts on them.