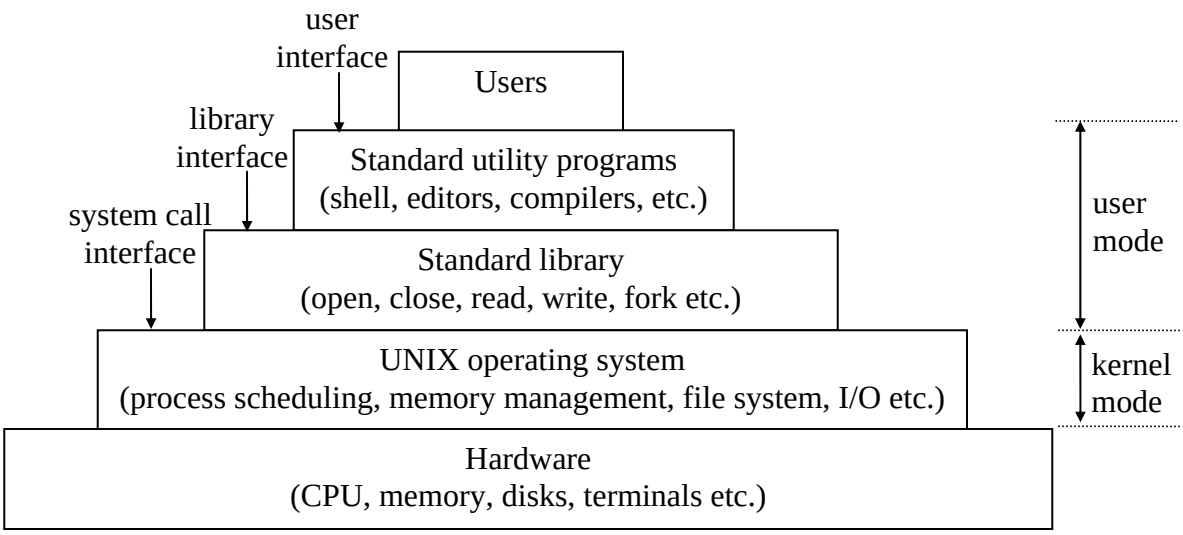
# CHAPTER 2
# THE UNIX SHELLS

The layers in a UNIX system is shown in the following figure.

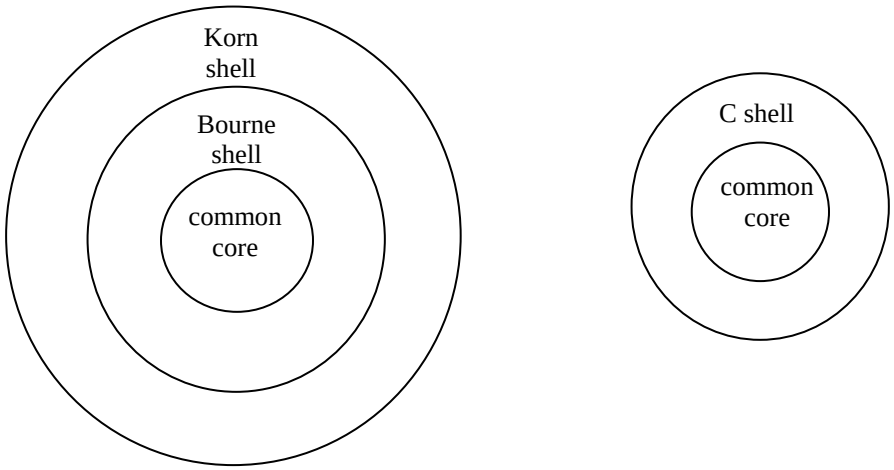



A shell is a program that is an interface between a user and the raw operating system (directly or through standard library). It makes basic facilities such as multitasking and piping easy to use, as well as adding useful file specific features such as wildcards and I/O redirection.

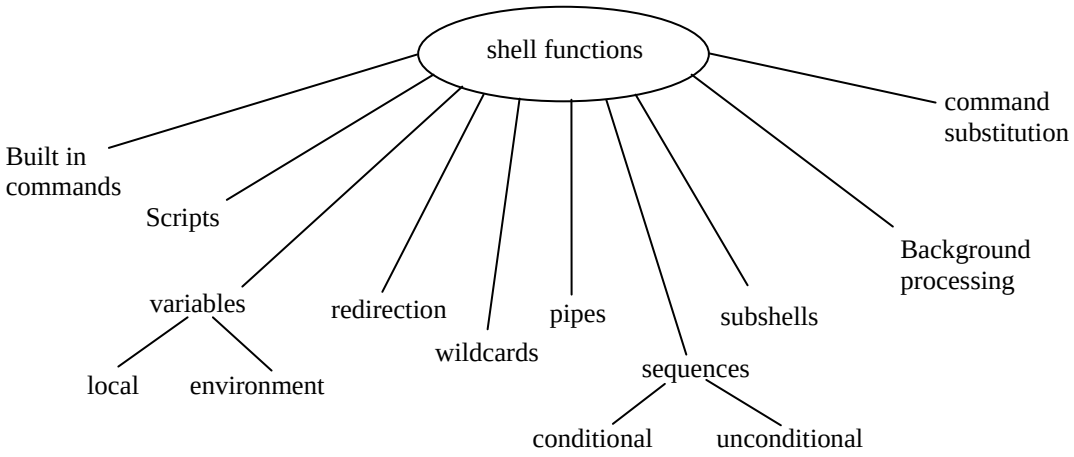
There are three common shells in use.

- The Bourne shell
- The Korn hell
- The C shell

The relation among these shells is shown below.




Basic shell functions are provided in the following figure.




SELECTING A SHELL: chsh

Even though it depends on the system, usually the pathnames for shells are as follows:

Bourne shell: **`/bin/sh`**
Korn shell: **`/bin/ksh`**
C shell: **`/bin/csh`**

The command chsh is used to chnge the current shell

```
$ chsh
changing login shell for halici
Old shell: bin/sh                    ... displayed
New shell: bin/csh                   ... you enter the pathname for the new shell
$ ^D                                 ... terminate login shell
login: halici
password: *********
%                                    ... the prompt of the C shell
```

SHELL is a system variable whose value is the pathname of the active shell. To access the value of SHELL, precede it by $ symbol as $SHELL. To see the value of a variable the echo command can be used. It is a built in shell command that displays all its arguments to the standard output device. By default it appends nw line to the output. The –n option inhibits this behaviour.

```
$ echo these are arguments
these are arguments
$ echo SHELL
SHELL
$ echo $SHELL
/bin/sh                  ... the value of the SHELL variable is displayed
```

SHELL OPERATIONS

When a shell is invoked

1. It reads a special start-up file, typically located user's home directory, that contains some initialisation information

2. It displays a prompt and it waits for a user command

3. If the user enters a control-D character on a line of its own, this is interpreted by the shell as meaning "end of input" and causes the shell to terminte; otherwise the shell executes the user command and returns to step 2.

A shell command may be simple as

```
$ ls
```

or complex-looking pipline sequences as

```
$  ps|sort|lpr  ... etc
```

METACHARACTERS

Some charcters are processed specially by shell and are known as metacharacters. Some of them are explained in the following

| | |
|---|---|
| > | Output redirection; writes standard output to a file |
| >> | Output redirection; appends standard output to a file |
| < | Input redirection; reads standard input from a file |
| * | File substitution wildcard; matches zero or more characters |
| ? | File substitution wildcard; matches any single characters |
| [...] | File substitution wildcard; matches any single character between brackets |
| `cmd` | Command substitution; replaced by the output from command |
| \| | Pipe symbol, sends the output of one process to the input of another |
| ; | Used to sequence commands |
| \|\| | Conditional execution; executes a command if the previous one fails |
| && | Conditional execution; executes a command if the previous one succeeds |
| (...) | Group of commands |
| & | Runs a command in the background |
| # | All characters that follow up to a new line are comment |
| $ | Access a variable |
| <<label | Input redirection; reads standard input from script up to label “lbl” |

When you enter a command, the shell scans it for metacharacters and processes them specially. When all metacharacters have been processed, the command is finally executed.

To turn off the special meaning of a metacharater, precede it by a “\” character.

```
$ echo hi >file
$ cat file
hi
$ echo hi \>file
hi >file                    ... not stored to file but displayed at screen
$
```

OUTPUT REDIRECTION: >, >>

```
$ cat >myfile
Ali Ahmet Can
^D
$ cat myfile
Ali Ahmet Can
$ cat >myfile
Cem Nil
^D
```

```
$ cat myfile
Cem Nil
$ cat >>myfile
Canan
^D
$ cat myfile
Cem Nil
Canan
$
```

INPUT REDIRECTION: <

```
$ mail halici <myletter
```

"**halici**" is the destination address to which the mail is to be sent. The file **myletter** contains the mail body that should be entered using standard input device if input redirection was to be not used.

FILENAME SUBSTITUTION: *, ?, [...]

```
$ ls –R #recursively  list my current directory
a.c b.c cc.c dir1 dir2
dir1:
d.c e.e
dir2:
f.d g.c
$ ls *.c   #any number of characters followed by ".c"
a.c b.c cc.c
$ ls ?.c  #one character followed by ".c"
a.c b.c
$ ls [ac]*  #either "a" or "c" followed by any number of characters
a.c cc.c
$ ls [A-Za-z]* # an alphabetic character followed by any number of characters
a.c b.c cc.c dir1 dir2
$ ls dir*/*.c # all files ending in ".c" in directories starting with "dir"
dir1/d.c dir2/g.c
$ ls */*.c # all files ending with ".c" in a subdirectory
dir1/d.c dir2/g.c
$ ls *2/?.? ?.?
a.c b.c dir2/f.d dir2/g.c
```

PIPES: |

The shell allows you to use the standard output of one process as the standard input of another one by connecting them using the pipe "|" metacharacter.

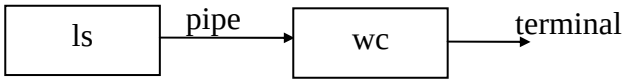
The sequence

**$ command1|command2**

causes the standard output of command1 to flow through to the standard input of command2. Any number of commands may be connected by pipes. A sequence of commands chained together in this way is called a pipeline. The standard error channel is not piped through a standard pipeline, although some shells support this capability.

```
$ ls
a.c b.c cc.c dir1 dir2
$ ls |wc –w
5
$
```

Note that the output of **ls** command is not displayed when it is piped to **wc**.

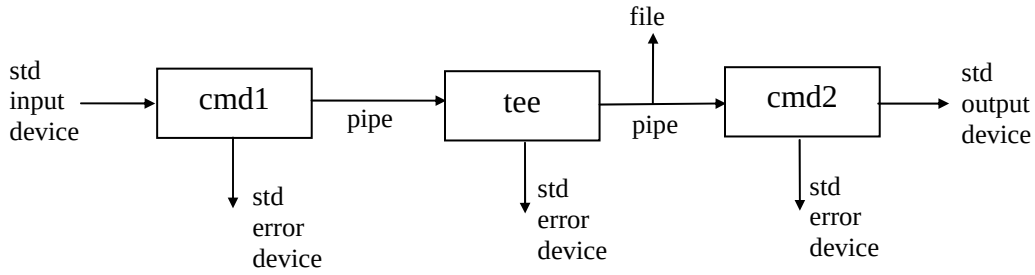



If you desire to see the output of intermediate commands use “tee”.

**tee –a {fileame}+**

a: append input to the files rather than overwriting them.

The **tee** utility copies its standard input to the specified files and to its standard output.




```
$ ls
a.c b.c cc.c dir1 dir2
$ ls |tee myfile|wc –w
5
$cat myfile
a.c b.c cc.c dir1 dir2
$ ls |tee myfile2
a.c b.c cc.c dir1 dir2 myfile
$ cat myfile2
a.c b.c cc.c dir1 dir2 myfile
```

SEQUENCES

If you enter a series of simpe commands or pipelies seperated by semicolons, the shell will execute them from left to right.

```
$ date; pwd; ls
Mon April 27 15:41:42 MEDT 2005
/home122/halici
a.c b.c cc.c myfile myfile2
```

Each command in a sequence may be individually redirected.

```
$ date >date.txt;ls;pwd >pwd.txt
a.c b.c cc.c myfile myfile2 date.txt
$ cat date.txt
Mon April 27 15:43:42 MEDT 2005
$ cat pwd.txt
/home122/halici
```

CONDITIONAL SEQUENCES

Every UNIX process terminates with an exit value. By convention, exit value 0 means that the process completed succsfully, and a non-zero exit value indicates failure. All built in shell commands return 1 if they fail.

```
$ cc myprog && echo compilation succeeded
```

Above, the C compiler named **cc** compiles a program called **myprog** and if compilation succeeds **echo** command will be executed. In the following case, **echo** command will be executed if compilation fails.

```
$ cc myprog || echo compilation failed
```

GROUPING COMMANDS

Commands may be grouped by placing them between parantheses, which causes them to be executed by a child shell (subshell). The group of commands shares the same standard input, standard output and standard error channels, and may be redirected as if it were a simple command.

```
$ date; ls; pwd >out.txt  # execute a sequence
Mon, May 3 15:35:12 MEDT 2005
a.c b.c
$ cat out.txt  # only pwd was redirected
/home122/halici
$  (date; ls; pwd) >out.txt  # execute a group
```

```
$ cat out.txt  # all the commands were redirected
Mon, May 3 15:35:12 MEDT 2005
a.c b.c
/home122/halici
```

BACKGROUND PROCESSING

If you follow a simple command, pipeline, sequence of pipelines or group of commands by the **&** metacharacter, a subshell is created to execute the commands as a background process. The background processes run concurrently with the parent shell, but does not take the control of the keyboard.

To explain background processing we will use the **find** command as example. In the following form it is used to list the filenames which matches the given pattern.

```
find pathlist –name pattern –print
```

Since it searches all the subdirectories its execution takes time. When a process executes in background, since it is executing concurrently with the parent shell, its output will interleave with foreground process' outputs.

```
$ find . –name a.c –print     ... search for a.c in the current directory and its subdirectories
./wild/a.c                    ... output from find
./reverse/a.c                 ... some more output as it continues its execution
$ find . –name a.c –print &   ... run the same command in background
27074                         ... process id of the background process
$ date                        ... date is executing in foreground
./wild/a.c                    ... some output from background find
Mon May 3 15:40:10 MEDT 2005  ... output from foreground date
$ ./reverse/a.c               ... foreground prompt and more output from background find
```

You may specify several background commands on a single line by seperating them by &.

```
$ date & pwd &                ... run date and pwd in background
27110                         ... process id of the background process date
27111                         ... process id of the background process pwd
/home122/halici               ... output from background pwd
$ Mon May 3 15:42:11 MEDT 2005 ... foreground prompt and output from background date
```

REDIRECTING BACKGROUND PROCESSES

Outputs of background processes can be redirected in the usual way.

```
$ find . –name a.c –print >find.txt &  ... background process, output redirected
27170
$ ls –l find.txt                       ... look at find.txt
```

```
-rw-r--r-- 1 halici 0  May 3 15:45:21 find.txt
$ ls –l find.txt                          ... watch it as it grows
-rw-r--r-- 1 halici 29 May 3 15:46:32 find.txt
$ cat find.txt
./wild/a.c
./reverse/a.c
$
```

Some utilities also produce output on the standad error channel, which must be redirected in addition to the standard output channel. Note that 0 represents standard input channel, 1 represents standard output channel and 2 represents standard error channel.

```
$ man ps >ps.txt &          ... save documentation about command ps in background
27203
$ Reformatting page, wait   ... std error output from "man"
done                        ... more std error output from "man
$ man ps >ps.txt 2>&1 &     ... redirect std error channel to the place redirected for std output
```

When a background process attempts to read from a terminal, the terminal automatically sends it an error signal which causes it terminate. To overcome the problem, use input redirection.

```
$ mail halici &             ... run mail, which needs input, in background
27210
$ No message !?!            ... it does not wait for keyboard and returns
$ mail halici <letter.txt & ... run mail in background with input redirection, no problem
27212
$
```