CHAPTER 3
SHELL PROGRAMS: SCRIPTS

Any series of commands may be stored inside a regular text file for later execution. A file that contains shell commands is called a script. Before you can run a script, you must give it execute permission by using **chmod** utility. Then to run the script, you only need to type its name.

When a script is run, the kernel determines which shell the script was written for, and then executes the shell using the script as its standard input. The shell is determined as specified in the following.

1. if the first line is just a #, then the script is interpreted by the C shell,
2. if the first line is of the form #!pathname, then the script is interpreted by the shell specified by the pathname,
3. if neither rule 1 nor rule 2 applies, then the script is interpreted by the Bourne shell.

```
$ cat >script.csh  # file extension has no importance
# this is a sample C shell script
echo –n the date today is
date # output today's date
^D
$ cat >script.ksh
#!/bin/ksh
# this is a sample Korn shell script
echo –n the date today is
date # output today's date
^D
$ chmod +x script.csh script.ksh # make them executable
$ ls –l script.csh script.ksh
-rwxr-xr-x halici 84  May 3 ... script.csh
-rwxr-xr-x halici 104 May 3 ... script.ksh
$ script.csh
the date today is Mon May 7 10:40:40 MEDT 2005
$ script.ksh
the date today is Mon May 7 10:41:10 MEDT 2005
```

The "**csh**" and "**ksh**" extensions of these scripts are used only to clarity, they do not have any effect on the shell type. A script doesn't even need an extension.

SUBSHELLS

When you log into UNIX, it supplies you with an initial login shell. Any simple comand that you enter is excuted by this initial shell. However there are several circumstaces when your current (parent) shell created a new (child) shell to perform some tasks.

1) When a grouped command is executed (if not executed in background, the parent shell sleeps until the child shell terminates)
2) When a script is executed (if not executed in background, the parent shell sleeps until the child shell terminates)
3) When a background job is executed (the parent shell continues to run concurrently with the child shell)

A child shell (subshell) has its own working directory, and so **cd** command executed in a subshell does not affect the working directory of the parent shell

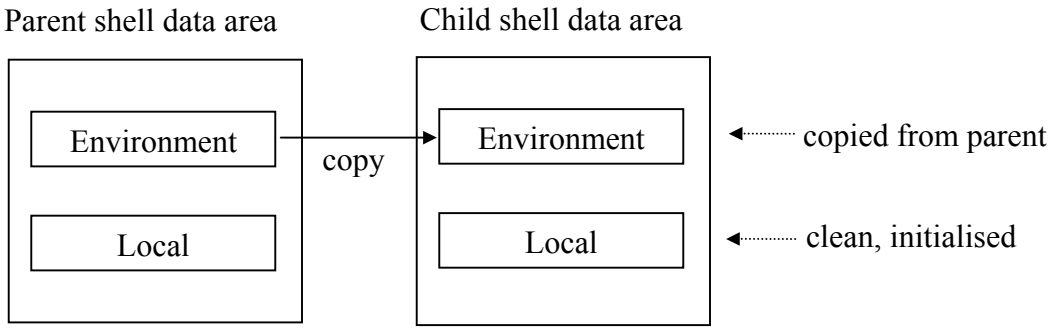
```
$ pwd
/home122/halici
$ (cd class; pwd)              ... the subshell moves to subdirectory and executes pwd
/home122/halici/class
$ pwd                          ... my login shell not moved
/home122/halici
```

VARIABLES

The shell supports two kind of variables

1) Local variables
2) Environment variables

A child shell inherits a copy of is its parent's environment variables while has a clean local variable space.



Every shell has a set of predefined environment variables that are initialised by the start up files. Similarly every shell has a set of local varibles that has special meanings to the shell and are particularly useful when writing scripts.

PREDEFINED ENVIRONMENT VARIABLES

Some of the predefined environment varibles are listed below:

HOME : The full pathname of your home directory
PATH : A list of directories to search for commands
MAIL : A full pathname of your mailbox
USER : Your user id
SHELL : The full pathname of your login shell
TERM : The type of your terminal

The syntax for assigning value to variables differs between shells, but the way to access a variable is the same. If you write $ followed by the name of a variable, this token sequence is replaced by the shell with the value of the named variable.

```
$ echo HOME=$HOME, PATH=$PATH
HOME=/home122/halici, PATH=.:/bin:/home122/bin
$ echo MAIL=$MAIL
MAIL=/home122/spool/mail/halici
$ echo USER=$USER, SHELL=$SHELL
USER=halici, SHELL=/bin/sh, TERM=vt100
```

The syntax for assigning value to a varible in the Bourne and Korn shells is as follows:

variable=*value* *...place no spaces around =*

```
$ firstname=Ugur  # assign value to a local variable named firstname
$ lastname=Halici # assign value to local variable lastname
$ echo $firstname $lastname # see their values
Ugur Halici
$ export lastname # lastname becomes environment variable
$ sh # start a child shell
$ echo $firstname $lastname # notice that firstname is not copied
Halici
$ lastname=NewOne     # change lastname
$ ^D  #terminate child shell
$ echo $firstname $lastname # parent shell remains unchanged,
Ugur Halici
$
```

BUILT IN VARIABLES WITH SPECIAL MEANINGS

The special meanings of some built in variables are listed below:

$$: The process id of the shell
$0 : The name of the shell script (if applicable)
$1..$9 : $n refers to the nth command line argument (if applicable)
$* : a list of all the command line arguments

```
$ cat script.sh
echo the name of this script is $0
echo the first argument is $1
echo a list of all arguments is $*
echo this script places the date  into a temporary file called $1.$$
date >$1.$$ # redirect the output of date
ls $1.$$ # list the file
rm $1.$$ # remove the file
$ script.sh ali cem can nur #execute the script with four arguments
the name of this script is script.sh
the first argument is ali
a list of all arguments is ali cem can nur
this script places the date  into a temporary file called ali.28107
ali.28107
$
```

The **shift** shell command causes all of the positional parameters **$2..$n** to be renamed as **$1..$(n-1)** and **$1** to be lost. If there are no positional arguments left to shift an error message is diplayed.

```
$ cat shift.csh
#!bin/csh
echo first argument is $1, all args are $*
shift
echo first argument is $1, all args are $*
$ shift.csh a b c d
first argument is a, all args are a b c d
first argument is b, all args are b c d
$ shift.csh a
first argument is a, all args are a
first argument is , all args are
$ shift.csh
first argument is , all args are
Shift: No more words              ... error message
$
```

QUOTING

To inhibit wildcard replacement, and variable/command substitutions quoting is used as follows:

- Single quotes (') inhibits wildcard replacement (***,?, [...]**), variable substitution (**$**), and command substitution (**`cmd`**),
- Double quotes (") inhibits wildcard replacement only,
- When quotes are nested, its only the outer quotes that have any effect

```
$ ls
a.c b.c cc.c
$ echo 3 * 4 = 12 #remember * is a wildcard
3 a.c b.c cc.c 4 = 12
$ echo "3 * 4 = 12"
3 * 4 = 12
$ name=ugur
$ echo 'my name is $name and the date is `date` '
my name is $name and the date is `date`
$ echo "my name is $name and the date is `date` "
my name is ugur and the date is Mon, May 11 10:41:10 MEDT 2005
```

HERE DOCUMENTS: <<label

Here document facility is used to provide an immediate input to a command in script. The input starts just after <<label and ends at where label appears alone in a line.

```
$cat here.sh
mail $1 <<EndOfText
Dear $1
    Please see me immediately!
- $USER
EndOfText
echo mail is sent to $1
$ here.sh halici
mail is sent to halici
$ mail # look at my mail
&1 # read mail 1
From: Ugur Halici <halici@metu.edu.tr>
To: halici@metu.edu.tr
Dear halici
    Please see me immediately!
- halici
&q  # quit out of mail
$
```

JOB CONTROL

Some commands used for job control are listd below:

ps : generates a list of processes and their attributes including their name, process id etc.
kill : allows you to terminate a process based on its process id
wait : allows a shell to wait for one of its child process to terminate
sleep n : sleeps for the specified numbr of seconds and then terminates

```
$ (sleep 10; echo done) &
27387
$ ps
PID      T    STAT  TIME COMMAND
27355    p3   S      0:00 -sh(sh)  the login shell
27387    p3   S      0:00
```