MODULE 5. PROCESS AND SYSTEM ADMINISTRATION

5.1 PROCESS BASICS

A process is an instance of a running program. A process is said to be born when the program starts execution and remains alive till the program is active. Once the program execution is completed, the process is said to die. A process has a name, usually same as that of the program. For example, when we execute ***date*** command, a process name ***date*** is created. But, a process cannot be considered synonymous with a program. Because, when two users run the same program, there is only one program on the disk, but there will be two processes in memory.

Processes are managed by kernel. Kernel allocates CPU resources based on time and priorities attached to each process. That is, CPU scheduling and memory management etc. are done by kernel. Processes have attributes, stored in a separate structure called as ***process table.*** Two important attributes of a process are –

- **Process ID (PID):** Each process is uniquely identified by an integer called PID, which is allotted by the kernel when the process is born. The PID is useful in controlling the processes, for example, killing it.
- **Parent PID (PPID):** The PID of the parent process is known as PPID. When a child process is spawn through its parent, then it will have PPID. When there are several processes with same PPID, it is ideal to kill the parent rather than each of the children.

Sometimes, process may not get completed in the expected time. Then, one may need to suspend it, or move it the background or to kill it. UNIX provides tools for all such actions to be carried out on processes.

5.1.1 The Shell Process

Whenever we log into UNIX system, the shell process is created, which may be sh (Bourne Shell), ksh(Korn shell), csh(C Shell) or bash (Bourne again shell). Any command that we give in UNIX later, will be treated as standard input to the shell process. This shell process remains alive till we logout.

The shell maintains a set of environment variables like PATH, SHELL etc. The pathname of shell is stored in SHELL and PID is stored in a special variable $$. So, to check the PID of the current shell, use the command –

```
$ echo $$
11662
```

The PID of your login shell cannot change till you are logged in. Once you logout and login again, it may be different. The lower value of PID indicates that the process was initiated quite early.

5.1.2 Parents and Children

Every process has a parent process. If you run the command,

```
$ cat emp.lst
```

then the process representing ***cat*** command is created by the shell process. Now, the shell is said to be parent process of ***cat.*** The hierarchy of every process is ultimately traced to the first process (PID 0), which is set up when the system is booted. It is like the root directory of the file system.

A process can have only one parent, but multiple children. The following command creates two processes ***cat*** and ***grep*** both are spawned by the shell –

```
$cat emp.lst | grep 'director'
```

5.2 ps: PROCESS STATUS

The ***ps*** command is used to display some of the process attributes. This command reads the data structure of kernel and process tables to fetch the characteristics of processes. By default, ***ps*** display the processes owned by the user running the command. For example –

```
$ ps
  PID TTY          TIME CMD
11703 pts/1    00:00:00 bash
11804 pts/1    00:00:00 ps
```

After the head, every line shows the PID, terminal (TTY) with which the process is associated (controlling terminal), the cumulative processor time (TIME) that has been consumed since the process has been started and the process name (CMD).

5.2.1 ps Options

Some of the options of ***ps*** command are discussed hereunder.

- **Full Listing (-f):** The –f option is used to get detailed listing of process attributes.

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
John     11703 11701  0 06:43 pts/1    00:00:00 -bash
John     11857 11703  0 07:27 pts/1    00:00:00 ps -f
```

  It shows the parent (PPID) and the owner (UID) of every process. STIME shows the starting time of process.

- **Displaying Processes of a User (- u):** The system admin may need to use –u option to know the activities by any user, by specifying user name as shown below–

```
$ ps -u chetana
  PID TTY          TIME CMD
11703 pts/1    00:00:00 bash
11894 pts/1    00:00:00 ps
```

- **Displaying all User Processes (-a):** This option is to list all the processes by all the users, excluding the system processes.

```
$ ps -a
  PID TTY          TIME CMD
11899 pts/1    00:00:00 ps
```

5.3 MECHANISM OF PROCESS CREATION

There are three major steps in the creation of a process. Three important system calls or functions viz. ***fork, exec*** and ***wait*** are used for this purpose. The concept of process creation cycle will help to write the programs that create processes and helps in debugging shell scripts. The three steps are explained here –

- **Fork:** A process in UNIX is created with the help of system call ***fork.*** It creates a copy of the process that invokes it. The process image is identical to that of the calling process, except few parameters like PID. When a process is forked, the child gets a new PID. The forking mechanism is responsible for the multiplication of processes in the system.

- **Exec:** A new process created through *fork* is not capable of running a new program. The forked child needs to overwrite its own image with the code and data of the new program. This mechanism is known as ***exec***. The child process is said to exec a new program.

- **Wait:** The parent then executes the system call ***wait*** for the child process to complete. It picks up the *exit status* of the child and then continues with its other functions. A parent may not need to wait for the death of its child.

For illustration, consider you are running a ***cat*** command from the shell. Now, the shell first forks another shell process. The newly forked shell then overlays itself with the executable image of ***cat***, and then it starts to run. The parent shell waits for ***cat*** to terminate and then picks up the exit status of the child. This is a number is returned to kernel by the child process.

When a process is forked, the child has different PID and PPID compared to parent. But, it inherits most of environment of its parent.

5.4 INTERNAL AND EXTERNAL COMMANDS

From the process point of view, the shell can recognize three types of commands as below–

- **External Commands:** The most commonly used UNIX utilities and programs like ***cat, ls, date*** etc. are called as external commands. The shell creates a process for each of such commands when they have to be executed. And, the shell remains as their parent.
- **Shell Scripts:** The shell spawns another shell to execute shell scripts. This child shell then executes the commands inside the script. The child shell becomes the parent of all the commands within that script.
- **Internal Commands:** Some of the built-in commands like ***cd, echo*** etc. are internal commands. They don't generate a process and are executed directly by the shell. Similarly variable assignment statements like x=10 does not create a child process.

5.5 RUNNING JOBS IN BACKGROUND

UNIX is a multitasking system, which allows to run multiple jobs at a time. As there can be only one process in the foreground, the other jobs must run in the background. There are two different ways of running the jobs in background – using & operator and using ***nohup*** command. These methods are discussed here.

5.5.1 No Logging out: &

The & operator of shell is used to run a process in the background. The usage is as shown below –

```
$ sort –o newlist.lst emp.lst &    #terminate line with & symbol
4080                               # PID is displayed
$                        #prompt is displayed to run another job
```

The & symbol at the end of the command line will make the job to run in the background. Immediately, the shell returns PID of the invoked command. And the parent shell will not wait till the job is completed (or the death of the child process), instead, the prompt is appeared so that a new job can be executed by the user. However, the shell remains as the parent of background process. Using & symbol, any number of jobs may be run in the background, if the system can sustain the load.

5.5.2 Log Out Safely : nohup

Normally, when the background processes are running, the user cannot logout. Because, shell is the parent of all the background processes and logging out kills the shell. And, by default when the parent shell is killed, all child processes will also die automatically. To avoid this, UNIX provides an alternative command ***nohup*** (no hangup). When this command is prefixed with any command, even after logging out, the background process keeps running. Note that the symbol & has to be used even in this case. For example,

```
$nohup sort emp.lst &
4154
nohup: appending output to ‘nohup.out’
```

Here, the shell returns the PID and some shells display the message as shown. When ***nohup*** command is run, it sends the standard output of the command to the file *nohup.out.* Now, one can safely logout the system, without aborting the command.

5.6 JOB EXECUTION WITH LOW PRIORITY: nice

In UNIX system, processes are executed with equal priority, by default. But, if high-priority jobs make demand, then they have to be completed first. The ***nice*** command with & operator is used to reduce the priority of jobs. Then more important jobs have greater access to the system resources.

To run a job with a low priority, use the command prefixed with ***nice*** as shown –

```
$nice wc emp.lst
15  31 741 emp.lst
```

If we would like to make it as a background process, use as –

```
$nice wc emp.lst &
```

The ***nice*** command is built-in command in C Shell. The value of nice is system dependent and usually ranges from 1 – 19. Commands execute with a nice value is generally in the middle of the range, say 10. The higher nice value implies lower priority. The ***nice*** command reduces priority of any process, and hence raising its nice value. The nice value can be explicitly given using –n option as –

```
$nice –n 3 wc emp.lst &  #nice value is incremented by 3 units
```

The priority of a process can be raised only by the super-user.

5.7 KILLING PROCESSES WITH SIGNALS

The UNIX system (in fact, any OS) needs to communicate with a process about the occurrence of any event. Such communication is done by sending ***signal*** to the process. Each signal is identified by a number, and has a specific meaning/functionality. Every signal is represented by their symbolic names with a prefix SIG. The ***kill*** command is used to send specific signal to the process.

If a program is running longer than expected, we may need to terminate it. So, we may press interrupt key. This action sends the SIGINT (with number 2) signal to process. By default, this will kill the process. A process may ignore a signal or it may execute a user-defined code to handle that signal. But, there are two signals viz. SIGKILL and SIGSTOP, which are never ignored by a process.

5.7.1 Premature Termination of a Process: kill

The ***kill*** command sends a signal with the intention of killing one or more processes. It is an internal command in most of the shells. The external command **/bin/kill** is executed only when the shell lacks the kill capability. One or more PID are given as arguments to ***kill*** command and by default, it uses the SIGTERM (number 15) signal.

```
$kill 12815
```

The above command kills the process with PID as 12815. If you don't know the PID, use *ps* command and then kill required process.

When there are more than on jobs (either in the background or in different windows), all of them can be killed with a single ***kill*** command by specifying their PIDs as below –

```
$kill 12815 23165 22810 22845
```

If all these processes have same parent, we can just kill the parent and the child processes will be automatically gets killed. However, any job is running in the background with the help of ***nohup***, the parent cannot be killed as ***init*** has acquired its parentage. Then each process has to be killed individually.

Killing the last background job: In most of the shells, the system variable $! stores the PID of the last background job. So, without giving actual PID, it can be killed. For example,

```
$sort –o new.lst emp.lst &          #running in background
23166
$ kill $!                  # background process will be killed.
```

Using kill with other signals: By default, ***kill*** uses SIGTERM signal to terminate the process. But, some processes may ignore it and continue execution. In such situations, the process can be killed with SIGKILL (number 9) signal. It is done as below –

```
$kill –s KILL 23165       #process with PID 23165 is killed
```

5.8 EXECUTE LATER: at AND batch

The user can schedule a job to run at a specific time of a day. When system is loaded with many jobs, it is ideal to run low-priority (or less urgent) jobs whenever system overhead is low (that is, when system is considerably free). There are two commands ***at*** and ***batch*** for job scheduling in UNIX.

5.8.1 One-Time Execution: at

The ***at*** command takes one argument as the time specifying when the job has to be executed. For example,

```
$ at 10:44                 #press enter by giving time
at> ls –l > out.txt        #Give the command to be executed
at> <EOT>                  #Press [CTRL+d]
job 10 at 2018-01-01 10:44
```

The job goes to the queue and at 10.44 on Jan 1st, 2018 the command *ls –l* will be executed and the output would be stored in the file `out.txt`. One can check this file later for the output.

5.8.2 Execute in Batch Queue: batch

The ***batch*** command schedules the job when the system overhead reduces. This command does not use any arguments, but it uses internal algorithm to determine the actual execution time. This will prevent too many CPU-hungry jobs from running at the same time.

```
$ batch < evalEx.sh        #evalEx.sh will be executed later
job 15 at 2007-01-09 11:44
```

Any job scheduled with ***batch*** goes to special ***at*** queue from where it will be taken for execution.

5.9 RUNNING JOBS PERIODICALLY: cron

The ***cron*** is a software utility (NOT a command) which behaves as a time-based job scheduler. Certain jobs in OS need to be executed in regular intervals of time. For example, system maintenance or administration may need to download/update general purpose utilities at regular intervals. Instructions (or commands) to be executed like this are stored

in a control file (**crontab** file) in the path - `/var/spool/cron/crontabs`. The cron utility is usually will be in a sleeping status. It wakes up every minute and checks the above control file whether any command has to be executed. User may keep a crontab file named after his/her name (like `/var/spool/cron/crontabs/john`) and the format of the entry would be like –

```
00-10 17 * 3,6,9,12 5 cat emp.lst
```

There are six fields in the above line:

1st Field : It specifies the number of minutes after the hour when the command is to be executed. It may range from 00 to 59. In the above example, 00-10 indicates that execution should happen every minute in the first 10 minutes of the hour.

2nd Field : Indicates the hour in 24-hour format. It may range from 1 to 24. In the above example, 17 indicates 5pm.

3rd Field : It is day of the month. Here, * means every day. Hence, the command has to be executed every minute for the first 10 minutes starting at 5pm every day.

4th Field : Specifies month (can be 1-12). Here, 3, 6, 9 and 12 indicates months March, June, September and December.

5th Field : Indicates day of the week (Sunday is treated as 0. Hence, values can be 0 to 6). In the above example, 5 indicates Friday.

6th Field : The actual command to be executed.

One can observe that 3rd field and 5th field are contradictory here. Because, 3rd fields says 'execute every day' and 5th field says 'execute on every Friday'. But, 5th field appears after 3rd field and hence it overrides 3rd field. Thus, meaning of above statement would be – execute the command `cat emp.lst` on every Friday for 10 minutes after 5 pm (ie. 5pm to 5:10pm) on the months March, June, September and December of every year.

5.10 TIMING PROCESSES: time

The ***time*** command is used to determine the duration of execution of a particular command/program. This required to check how efficiently the system resources are being used by the programs. The ***time*** command executes the program and also displays the time usage. For example,

```
$ time ls
avgSal.awk  elist.lst     myf1           Shell1.sh   test.bak
caseEx.sh   emp1.lst      myfile         Shell2.sh   test.lst
cmdArg.sh   emp.lst       myFirstShell   t1          test.sh

real    0m0.004s
user    0m0.000s
sys     0m0.000s
```

Here, we are checking execution time required for *ls* command. It first executes *ls* (displays all files/directories) command and then displays time. The *real* time is the clock elapsed time from invocation of the command until its termination. The *user* time is the time spent by the program in executing itself. The *sys* time is the time required by the kernel in doing

work on behalf of the user process. Ideally, the sum of user time and sys time must be equal to CPU time. But, due to system load (multiple tasks running at a time), it may differ.

5.11 ESSENTIAL SYSTEM ADMINISTRATION

A system administrator (or super user or root user) has vast powers and access to almost everything in UNIX system. The stability of any UNIX installation depends on the effectiveness of system admin. System admin is responsible for managing entire system like maintaining user accounts, security, disk space, backups etc.

5.11.1 The System Admin's Login: root

In every UNIX system, the system admin has a special login name called as ***root***. It is automatically available in every system, and no need to create exclusively. The password is set during system installation.

```
login: root
password: ******* [press enter]
#
```

The prompt for *root* is # in most of the systems. After logging in, the system admin is placed in root's home directory (either / or /root).

5.11.2 Acquiring Superuser Status

Any user can acquire the status of superuser, if he/she knows the root password. For example the user *john* may become superuser using the command ***su*** as shown –

```
$ su
password: *******          #give root's password
# pwd                      #working directory unchanged
/home/john
```

Observe that current working directory has not changed for the user *john* but, he will get the privileges of superuser now onwards. If he wishes to enter root's home directory, the ***su –l*** command must be used.

Whenever a program stops running in a user's machine, the admin tries to run it in a simulated environment. The following command recreates the user's environment without using user's login-password –

```
$ su – john
```

This executes *john's* `.profile` file and temporarily creates *john's* environment. The ***su*** command runs a separate sub-shell and can be terminated either using [Ctrl+d] or ***exit***.

5.12 THE ADMINISTRATOR'S PRIVILEGES

Following are some of the important privileges of a system administrator:

- Changing contents or attributes (like permission, ownership etc) of any file. He/she can delete any file even if the directory is write-protected.
- Initiate or kill any process.

- Change any user's password without knowing the existing password
- Set the system clock using ***date*** command.
- Communicate with all users at a time using ***wall*** command.
- Restrict the maximum size of files that users can create, using ***ulimit*** command.
- Control user's access to the scheduling services like ***at*** and ***cron***.
- Control user's access to various networking services like FTP, SSH (Secured Shell) etc.

Some of the above jobs of admin are discussed hereunder.

5.12.1 Setting the System Date: date

The ***date*** command – when used by the system admin – allows to set the system date. It takes the argument, usually as 8-character string in the form of MMDDhhmm, optionally followed by a 2 or 4 digit year as shown –

```
# date 01032124
Thu Jan 3 21:24 IST 2017
```

5.12.2 Communicating with Users: wall

The ***wall*** command is used to communicate with all users who are currently logged in. In most of the systems (except Linux), this command can be run only by the admin. Example of this command would be –

```
# wall                                          #command
This server will shutdown at 4pm today.         #message
[Ctrl+d]
```

Now, the message given here will be broadcasted to all the users who are currently logged into the system.

5.12.3 Setting Limits on File Size: ulimit

Sometimes, programs may grow rapidly (due to negligence or error in the program) and consume much space on the disk. Hence, the admin can use ***ulimit*** command to restrict the maximum possible size that can be consumed by a user created file. By default, size of the user file may be unlimited in most of the machines. The restrictions can be imposed as–

```
#ulimit 20963260
```

Here, the number indicates the size of file in bytes. The admin can keep the above line in `/etc/profile` of every user.

5.12.4 Controlling use of at and cron

The use of ***at*** and ***batch*** is restricted and controlled by the files `at.allow` and `at.deny` in `/etc/cron.d` (`/etc` in Linux). If the file `at.allow` is present, only the users listed in it are permitted to use ***at*** and ***batch*** commands. If this file is not present, the system checks

`at.deny` file for the users who are barred from using these commands. If both of these files are not present, only system admin is permitted to use these commands.

The ***cron*** scheduler is used by the admin to make commands like ***find*** and ***du*** (disk usage) compile useful information about the system or for automating the backup operations. To control the authorization, two files `cron.allow` and `cron.deny` available in `/etc/cron.d` are used.

5.13 STARTUP AND SHUTDOWN

Whenever the system is about to start or about to shutdown, series of operations are carried out. We will discuss few of such operations here.

- **Startup:** When the machine is powered on the system looks for all peripherals and then goes through a series of steps that may take some time to complete the boot cycle. The first major event is the loading of the kernel (`/kernel/genunix` in Solaris and `/boot/vmlinuz` in Linux) into memory. The kernel then spawns ***init***, which in turn spawns further processes. Some of these processes monitor all the terminal lines, activate the network and printer. The ***init*** becomes the parent of all the shells.

  A UNIX system boots to any one of two ***states*** (or mode) viz. single user mode or multiuser mode. This state is represented by a number or letter called as ***run level.*** The default run level and the actions to be taken for each run level are controlled by ***init***. The two states are discussed here –
  - **Singer-user Mode:** The system admin uses this mode to perform administrative tasks like checking and backing up individual file systems. Other users are prevented from working in single-user mode.
  - **Multiuser Mode:** Individual file systems are mounted and system daemons are started in this mode. Printing is possible only in multiuser mode.

  The ***who –r*** command displays the run level of our system:

```
$ who -r
      .  run-level 3  2007-01-06 11:10          last=S
```

  The machine which runs at level 3 supports multiuser and network operations.

- **Shutdown:** While shutting down the system, the administrator has certain roles. The system shutdown performs following activities –
  - Through the ***wall*** command notification is sent to all the users about system shutdown and asks to logout.
  - Sends signals to all running processes to terminate normally.
  - Logs off the users and kills remaining processes
  - Un-mounts all secondary file systems.
  - Writes file system status to disk to preserve the integrity of file system
  - Notifies users to reboot or switch-off, or moves the system to single-user mode.

5.14 MANAGING DISK SPACE

In the due course of time, the system disk space gets filled up by various files. If not managed properly, the whole disk many become full. So, the system admin must regularly scan the disk and identify the files which are no longer in use. Then such files may be deleted to create a space in disk. For doing these jobs, admin needs three commands ***df, du*** and ***find***. These commands can be used by even any normal user.

5.14.1 Reporting Free Space: df

The ***df*** command is used to check the amount of free space available in every file system.

```
$ df
                                                    Mounted on
Filesystem    1K-blocks  Used    Available Use%     /dev/mapper/

              16663980   6490020   9313800  42%        /
/dev/hda3       101105     13684     82200  15%        /boot
tmpfs           891796         0    891796   0%        /dev/shm
none            891796        40    891756   1%        /var/lib
```

This command displays file system, number of total blocks, used blocks, available space (in blocks and percentage) and in which directory, the file system is mounted.

5.14.2 Disk Usage: du

Usually, the user will be interested in the space consumption of a specific directory, rather than entire file system. The ***du*** command is used to check the usage of space in a specific directory recursively.

```
$ du /home/john
16      /home/john/.kde/Autostart
24      /home/john/.kde
16      /home/john/myDir
8       /home/john/.mozilla/extension
8       /home/john/.mozilla/plugins
24      /home/john/.mozilla
548     /home/john
```

This command lists the disk usage of each sub directory and finally gives the summary. If the list of sub-directories is very large and if we are interested only in the summary, ***-s*** option can be used as –

```
$ du -s /home/john
548     /home/john
```

If the system admin would likes know the space consumed by all the users, use the command as –

```
$du –s /home/*
```

5.15 A BACKUP PROGRAM: cpio

Taking backup of the system in regular intervals is very important safety measure to be carried out by the system admin. There are two backup programs ***cpio*** and ***tar***. Both of these programs combine a group of files into an archive, with suitable headers preceding the contents of each file. The backup device can be a magnetic (or cartridge) tape, floppy disk or even a disk file.

The ***cpio*** (copy input-output) command copies files to and from a backup device. It uses standard input to take the list of file names. Then the files are copied along with their contents and header to standard output. Later, from standard output it is redirected to a file or a device. The ***cpio*** command uses two options viz. -o for output and –i for input, of which any one of them must be used.

5.15.1 Backing Up Files (-o)

The following example uses the ***ls*** command to generate list of all filenames as an input to ***cpio*** command. Then –o option is used to create archive on the standard output, which needs to be redirected to a device file.

```
# ls | cpio –ov > /dev/fd0
caseEx.sh
test.lst
emp.lst
here.sh
tputEx.sh
4 blocks            #total size of the archive
```

The –v option displays each filename on the terminal while it is being copied. The above command compresses the files listed and stores the compressed file into `/dev/fd0`.

5.15.2 Restoring Files (-i)

Compressed files can be restored using –i option as shown below –

```
# cpio –iv < /dev/fd0
caseEx.sh
test.lst
emp.lst
here.sh
tputEx.sh
4 blocks
```

While restoring the subdirectories, the ***cpio*** command maintains the same subdirectory structures in the hard disk.

NOTE: In above examples, the backup is stored in floppy drive (`/dev/fd0`). But, one can store anywhere by specifying proper path.

5.16 THE TAPE ARCHIVE PROGRAM: tar

The ***tar*** is a tape archive command that was being used before the emergence of ***cpio***. It can create archives on tapes and floppy drives. Some features are –

- It doesn't use standard input to obtain a file list
- It accepts file and directory names as arguments
- It copies one or more directory trees, in a recursive manner
- It can append to an existing archive, without overwriting.

The ***tar*** has three important options, viz. –c (copy), -x (extract) and –t (table of contents). The –f option has to be specified for mentioning the device name where the backup has to be stored. And just as in ***cpio***, the –v option can be used to view the files being processed.

5.16.1 Backing Up Files (-c)

The illustration of ***tar*** command for archiving files is –

```
# tar –cvf /dev/fd0  /home/john/*.sh
a /home/john/test.sh 1 tape blocks
a /home/john/caseEx.sh 1 tape blocks
a /home/john/evalEx.sh 5 tape blocks
```

The above command indicates that all the files with extension `.sh` in `/home/john` are archived into `/dev/fd0`. The character a at every line of output indicates the files are being appended into the device.

5.16.2 Restoring Files (-x)

Files archived in specific device are restored using –x option. When file/directory name is not specified, all the files from the backup device are restored.

```
# tar –xvf /dev/fd0
x /home/john/test.sh 1 tape blocks
x /home/john/caseEx.sh 1 tape blocks
x /home/john/evalEx.sh 5 tape blocks
```

5.16.3 Displaying the Archive (-t)

The –t option is used to display the contents of the device in a long format (similar to ls –l).

```
# tar –tvf /dev/fd0
rwxr-xr-x 203/50    472   Jun 4 10.23 2017 ./test.sh
rwxr-xr-x 203/50    583   Jun 4 11.51 2017 ./caseEx.sh
rwxr-xr-x 203/50     87   Jun 4  9.37 2017 ./evalEx.sh
```

Observe that file names are displayed as relative paths.

5.17 CUSTOMIZING THE ENIVIRONMENT

Interaction with the OS takes much time of the user. The user has to constantly change directories, list of files, edit/compile programs, repeating previous commands etc. For all such tasks, the environmental settings must be pleasant. UNIX environment can be

customized by manipulating the settings. The commands can be made to change their default behavior.

5.18 ENVIRONMENT VARIABLES

There are two types of shell variables: local and environment variables. PATH, HOME, SHELL etc. are environment variables, because they are available in user's environment. The user's environment constitutes sub-shells which run the shell scripts, main commands and editors. The local variables are local to the scope. For example, set the local variable DOWNLOAD_DIR as –

```
$ DOWNLOAD_DIR=/home/john/download
$ echo DOWNLOAD_DIR
/home/john/download
```

Now, spawn a child shell and then check the value of DOWNLOAD_DIR variable –

```
$ sh
$ echo DOWNLOAD_DIR
                          #prints nothing
$
```

The above example shows that the scope of local variable DOWNLOAD_DIR is limited to only that shell, but not available to even a child shell.

But, environment variables like PATH will be available everywhere. The ***set*** statement displays all the variables available in current shell. The ***env*** command displays only the environment variables. The following is a partial list of ***environment variables.***

```
$ env
HOME=/home/henry
IFS= '
'
LOGNAME=john
MAIL=/var/mail/john
MAILCHECK=60
PATH=/bin:/usr/bin:.:/usr/ccs/bin
PS1= '$'
PS2= '>'
SHELL=/usr/bin/bash
TERM=xterm
```

The PATH variable instructs the shell about the route it should follow to locate any executable command. The current value of PATH can be displayed by using ***env*** command or using the statement `echo $PATH`. To add any directory, say, `/usr/test` to existing path, reassignment is possible as below –

```
$ PATH=$PATH:/usr/test
```

Note that various directories in PATH are separated by colon. Hence, while appending any other directory, we must use that.
The HOME is another important environment variable. When any user logs in the UNIX places him/her in a directory named after the user's login name. This directory is called as home directory or login directory. This is available in environment variable HOME.

```
$ echo $HOME
/home/john
```

The system admin can set $HOME in `/etc/passwd`. This setting looks like –

```
john:x:208:50::/home/john:/bin/ksh
```

Here, 208 is userID and 50 is GroupID.

5.19 ALIASES

The bash and ksh supports ***aliases*** (alternative names), which lets the user to assign any shorthand names to frequently used commands. This is done with the command ***alias***.

The `ls -l` command is used many times in UNIX. So, alias can be set for this command as –

```
$ alias el='ls -l'
```

Now onwards, in place of `ls -l`, user can use `el`. Note that, there must not be a space before and after = symbol in the above statement.

When we need to use ***cd*** command frequently to change directory to some long pathname as /home/john/ShellPgms/Scripts, then alias can be set as –

```
$ alias cdSys="cd /home/john/ShellPgms/Scripts"
```

So, we can just use `cdSys` to change the directory.

The alias set once can be revoked using the command ***unalias***. So, to unset the alias `cdSys`, use the statement as –

```
$ unalias cdSys
```

5.20 COMMAND HISTORY

In the Bourne shell, the command has to be retyped when it has to be re-executed. This drawback is overcome in bash and ksh using the command ***history***. It is a feature that treats the previous command as an ***event*** and associates it with an event number. Using this event number, one can recall the previous commands (executed even in previous sessions, that is, even after logging out the system) and execute them.

The ***history*** command in bash displays all commands that are previously executed, and in ksh, it displays last 16 commands. One can mention how many commands he/she would like to see, by giving the number as an argument to ***history*** command as –

```
     $ history 5          #in bash
Or   $ history -5         #in ksh
```

The output of above command shows 5 recent commands along with the event number –

```
$ history 5
999  du -s /home/*
1000  exit
1001  emacs emfile
1002  clear
1003  history 5
```

By default, the bash stores all previous commands in `$HOME/.bash_history`. Once we know the previous command with event number, we may need to re-execute it, either as it is or with modification. Some of such tasks can be done as discussed below:

- **Accessing Previous Commands by Event Numbers (! And r) :** The ! symbol (in ksh, we should use r) is used to repeat the previous command. To repeat the immediate previous command, use –

  ```
  $ !!      #it repeats previous command and executes.
  ```

  The same job can be done with `r` in ksh.

  If one needs to execute the specific command, the event number must be attached as –

  ```
  $ !35          # executes command with event number 35
  ```

  Note that, the above format executes the 35th command from the history. If we would like know the actual command, before really executing it, the ***p*** (print) modifier has to be used as –

  ```
  $ !67:p
  clear          #67th command in history is clear
  ```

  Relative addressing of event numbers is possible. That it, when we would like to execute 3rd previous command from now, use the statement as –

  ```
        $ !-3          # in bash
  Or    $ r-3          # in ksh
  ```

- **Executing Previous Commands by Context:** It is quite difficult for any user to remember the event number of all the previous commands he/she used in past. But, there is a chance that he/she will be remembering first character/string of the command. The ! or r will help the user to recollect the commands by giving the specific character as below –

  ```
  $ !g #executes last command which started with g (in bash)

  Or $ r g     # same task, in ksh
  ```

- **Substitution in Previous Commands:** Frequently, user may need to execute any of the previous commands only after some modification in them – say changing the pattern in grep command, changing the string etc. This is possible with the help of modifier ***:s*** and the / as the delimiter between old and new patterns.

  Consider an example – assume that, we had executed the *echo* command as –

```
$ echo $HOME  $PATH
```

  This would display the environment variables *HOME* and *PATH*. Now, we would like to replace PATH by SHELL. This can be done using the following command –

```
$ !echo:s/PATH/SHELL
echo $HOME $SHELL
/home/john  /bin/bash
```

  Here, the command has the meaning – "in the previous echo command, replace PATH by SHELL"

- **Using Last Argument to Previous Command ($_):** Frequently we use several commands on the same file. In such situations, instead of specifying filename every time, we can use $_ as the abbreviated filename. The $_ indicates that *last argument to previous command*. For example, if we create a directory *Test* using *mkdir* command and then we would like to change the directory to this new directory, we can use the command as –

```
$ mkdir Test        # create new directory Test
$ cd $_             # change directory to Test
$ pwd               # verify
/home/john/Test
```

- **The History Variables:** The behavior of history mechanism depends on few environment variables. The HISTFILE variable determines the filename that saves the history list. The .bash_history is a default history file in the home directory. The history list is maintained in memory as well. For this purpose, it uses two variables HISTSIZE and HISTFILESIZE for setting the size of history in the memory and on the disk. For example,

```
HISTSIZE=500
HISTFILESIZE=1000
```

  Based on the disk space of one's machine, the user can set these values to keep more data in these files.

5.21 IN-LINE COMMAND EDITING

The command line (of terminal) in bash and ksh provides editing of the command using the characters similar to that used in vi editor. For this purpose, one need to do the following setting:

```
$ set –o vi
```

Now onwards, the command line of our terminal can be used as if we are using vi editor (with command mode, insert mode etc). Switching between the modes is done by pressing *Esc* key. And one can use the commands like ***i (insert), a(append), r(replace), x(delete), 3x(delete 3 characters), dd(delete entire line)*** etc. (It is assumed that the student knows these commands in vi editor).

To exit the in-line editing mode, reset it using –

```
$ set +o vi
```