```
        personnel
        admin
```

Now you can use **fgrep** in the same way you used **egrep**

```
        #fgrep   f pat1.lst emp.lst
```

8. PROCESS MANAGEMENT COMMANDS

A process is simply an interface of a running program. A process is said to be born when the program starts execution, and remains alive as long as the program is active. After execution is complete, the process is said to die. This process also has a name, usually the name of the program being executed. For example when you execute the **grep** command, a process named **grep** is created.

It is the kernel (and not the shell) that is ultimately responsible for the management of these processes. It determines the time and priorities that are allocated to processes so that multiple processes are able to share CPU resources. It provides a mechanism by which a process is able to execute for a finite period of time, and then relinquish control to another process.

Typically, hundreds, or even thousands of processes can run in a large systems. Each process is uniquely identified by a number called the PID (process identifier) that is allotted by the kernel when it is born.

The sh process:

When you log into a system, a process is immediately set up by the kernel. This is technically a Linux command which may be **sh**(Bourne shell), **ksh**(Korn shell) or **bash**(Bourne again shell).Any command you type in at the prompt is actually standard input to the shell process(or program).This process remains alive until you log out.

The shell maintains a set of variables that are available to the user. The sh process, identified by its PID, is stored in a special variable **$$**.To know the PID of your current shell, simply type

```
        #echo $$                    #The process number of the current shell
        659
```

Parents and children:

Just as a file has a parent, every process also has one. This parent itself is another process, and a process born from it is said to be its child. When you run the command

```
                cat emp.lst
```

From the keyboard, a process representing the cat command, is started by the sh process(the shell).This cat process remains active as long as the command is active.sh is said to be the parent of cat, the cat is said to be the child of sh. Since every process has a parent, you can t have an orphaned process (i.e., a process that has no parent) in Linux system. Like a file, a process can have only one parent. Moreover, just as a directory can have more than one file under it, the multitasking nature permits a process to generate (or spawn) one or more children. This is most easily accomplished by setting up a pipeline. The command

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

sets up a two processes for the two commands. These processes have the names cat and grep and both are spawned by sh.

Note: All commands don t sets up a process. Built-in commands of the shell like pwd.cd etc. don t create processes.

ps: process status:

The ps command is used to display the attributes of a process.It is one of the few commands of the system that has knowledge of the kernel built into it.It reads through kernel data structures or process tables to fetch the process characteristics. By default, ps lists out the processes associated with a user at that terminal:

```
#ps
PID    TTY    TIME      CMD
476    tty03  00:00:01  login
659    tty03  00:00:01  sh
684    tty03  00:00:00  ps
```

Like who, ps also generates a header information. Each line shows the PID, the terminal(TTY) with which the process is associated, the cumulative processor time(TIME) that has been consumed since the process has been stated, and the process name(CMD).Linux shows an additional column that is usually an S(sleeping) or R(running).

You can see that your login shell (sh) has the PID 659, the same number echoed by the special variable $$.ps itself is an instance of a process identified by the PID 684.

Process options:
Displaying process ancestry (Full listing):

The ps command, by default, doesn t provide about the ancestry of a process. To get them in SCO UNIX, you have to use the f (full) option:

```
#ps  f
UID    PID   PPID  C    STIME     TTY    TIME      CMD
Root   476   1     0    17:51:58  tty03  00:00:01  /bin/login kumar
Kumar  659   476   4    18:10:29  tty03  00;00:01  -sh
Kumar  685   659   15   18:26:44  tty03  00:00:00   ps  f
```

The PPID (parent PID) is the PID of the parent process that spawned this process. The login shell has the PID 659 and PPID 476,which means that it was set up by a system process having PID 476(the login program).The ps command itself has the PPID 659 that also happens to be the PID of the sh process.

The first column (UID) displays the user s login name.C indicates the amount of CPU time consumed by the process. STIME shows the time the process started. TIME shows the total CPU time used by the process.CMD displays the full command line with its arguments.

Displaying processes of a user (-u):
The user (-u) option lets you know the activities of any user, for instance, the user local:

```
#ps  u local
```

Displaying all user processes (-a):
The a (all) option lists out the processes of all users, but doesn t display the system processes:

```
#ps  a
PID    TTY    TIME       CMD
662    tty02  00:00:00   ksh
705    tty04  00:00:00   sh
1005   tty01  00:00:00   ksh
1017   tty01  00:00:04   vi
680    tty03  00:00:00   ksh
1056   tty02  00:00:00   sort
1058   tty05  00:00:00   ksh
```

```
        1069    tty02   00:00:00        ps
```

Five users are at work here, as evident from the terminal names displayed. Most of them seem to be users of the korn shell. Not much work seems to be going on here, except for sorting and a file editing operation with vi.

System processes (-e):

Over and above the processes that a user generates, there are a number of system processes that keep running all he time. Most of them are spawned during system startup, and some of them start when the system goes to the multi-user state. To list them you have to use e option (or x).

```
        #ps   e
        PID     TTY     TIME            CMD
        0       ?       00:00:00        sched
        1       ?       00:00:01        init
        2       ?       00:00:00        vhand
        3       ?       00:00:01        bdflush
        260     ?       00:00:00        cron
        282     ?       00:00:00        lpsched
        308     ?       00:00:00        rwalld
        336     ?       00:00:00        inetd
        339     ?       00:00:00        routed
        403     ?       00:00:00        mountd
        408     ?       00:00:00        nfsd
```

The characteristic feature of system processes is that most of them are not associated with any terminal at all (shown by?)Some of these processes are called daemons that do important work for the system.

Running jobs in background:

A multi-tasking system lets user do more than one job at a time. Since there can be only one job in the foreground, the rest of the jobs have to run in the background. The & is the shell s operator used to run a process in the background (i.e., not wait for its death) Just terminates the command line with an &; the command will run in the background:

```
        #sort   o emp.lst emp.lst &                    #the job  s PID
```

The shell immediately returns a number-the PID of the invoked command (550).The prompt is returned, and the shell is ready to accept another command, even though the previous command has not been terminated yet.

Kill: premature termination of a process

The system often requires communicating the occurrences of an event to a process. This is done by sending signal to the process. If you have program running longer than you anticipated, or if you have changed your mind and want to run something else, simply send a signal to the active process with a specific request of termination.

You can terminate a process with the kill command. The command uses one or more PID s as its arguments. Thus,

```
                kill 105
```

Terminates the job with the PID 105.If you run more than one job in the background, they can all be killed with a single kill statement by specifying the PID s of all the background jobs in the command line:

kill 121 122 125 132 138 144

If all these processes have same parent, you may simply kill the parent in order to kill all its children.

Killing with signal numbers: Kill, by default uses the signal 15 to terminate the process. The process can be killed with signal number 9, sometimes known as the sure kill signal. This signal can t be generated at the press of a key, so kill lets you use the signal number as an option:

```
kill -9 121                    #kills process 121 with signal number 9
kill  n -9 121                 #same as above
```

You can also kill all processes in your system except the login shell, by using a special argument 0:

```
kill 0
```

This terminates all background processes with the signal number 15.The login shell ignores this signal, so you have to kill it by using any of the following commands:

```
kill -9 $$                     #$$ stores PID of current shell
kill -9 0                      #kills all processes including the login shell
kill -1 0                      #same
```

Nice: job execution with low priority

Processes in the system are executed with equal priority. This is not always desirable since high-priority jobs must be completed at the earliest. Linux offers the nice command, to reduce the priority of jobs.

To run a job with a low priority, the command name should be prefixed with nice:

```
nice wc  l uxmanual
```

Or better still with

```
nice wc  l uxmanual &
```

Nice values ranges from 0-39,and the commands run with a nice value of 20 in the Bourne shell, a higher nice value meaning a lower priority. nice reduces the priority of any process by 10 units, raising its value to 30.The amount of reduction can also be specified with the n option:

```
nice  n 15 wc  l &             #nice value becomes 35
```

An ordinary user can t increase the priority of a process; that power is reserved for the super user.

Job control in the Korn and Bash shells:

If you are using Korn or Bash shell, you can use its job control facility to manipulate jobs. You can relegate a job to the background, bring it back to the foreground, suspend or even kill it. You often have to resort to these methods when you expect a job to complete in 10 minutes, and it goes on for half an hour. Surely, you won t like to cancel it because a lot of work has been done already. To handle this situation, you will need the built-in bg, fg, suspend and kill commands.

If you have invoked a command and the prompt has not yet returned, you can simply suspend the job by pressing <ctrl-z>.You will then see the following message:

```
        #xeyes &
[1] + stopped
```

Mind you, the job has not yet been terminated. Now if you want to do some other work in the foreground, use the bg command to push the last spell checking job to the background:

```
        #bg
[1]       #xeyes &
```

The & at the end of the line indicates that the job is now being run in the background. So, a foreground job goes to the background, first with <ctrl-z>, and then the bg command. You can ofcourse start a job in the background itself:

```
        #gedit /tmp/hello.c &
[2] + stopped
```

The [2] shows that this is the second job currently running in the background. You can run more jobs in this way:

```
        #xload   &
[3] + stopped
```

Now you can run one more jobs in this way:

```
        Sleep 100000 &
```

Now that you have three jobs running, you can have a listing of their status with the jobs commands:

```
#jobs
[1] +Running                    xeyes  &
[2] - Running                   gedit  /tmp/hello.c  &
[3]   Running                   xload  &
[4]   Running                   sleep  100000  &
```

You can bring any of the background to the foreground with the **fg** command. This, along with **bg**, can be used in a number of ways. With four background jobs now running, if you run **fg** without any argument, it will bring the most recent background job, i.e. the sleep command, to the foreground.

fg and **bg** optionally also use a job identifier prefixed by a % symbol. If you specify fg %1, the first job is brought to the foreground. If you use **fg %sort**, the **sort** job comes to the foreground.

The kill %1 kills the first background job, and same as kill %4 kills the four background job. If you use fg %sort, the sort job comes to the foreground. You can also use the **stop** command to kill a background job. However, unlike the fg command which, by default, brings the most recent job to the foreground, stop needs one or more job identifiers.

Thus, **stop %1** terminates execution of the first job, while **stop %sort** halts execution of **sort** command.