CHAPTER 11

Processes in Linux

11.1 INTRODUCTION

The boot process in Linux (in most of UNIX variants) has two stages: the boot loader stage and the kernel stage. In the following pages we describe booting process in general in UNIX and Linux specifically.

The main components of the boot loader stage are the hardware stage, the firmware stage, the first-level boot loader, and the second-level boot loader. The booting process begins when the hardware is powered on. after some initialization (power of self test, POST), control goes to the firmware. Firmware, also referred to as "BIOS" on some architectures, detects the various devices on the system, including memory controllers, storage devices, bus bridges, and other hardware. The firmware, based on the settings, hands over control to a minimal boot loader known as the master boot record, which could be on a disk drive, on a removable media, or over the network. The boot loader may be available in the boot block of bootable partition. In BIOS setting, in what sequence drives are required to be checked for this boot loader is specified. On those systems in which multiple operating systems are installed, this boot loaders (such as LILO, GRUB, Windows NT loader, OS/2 Loader) will be displaying a menu from which user can select which OS they want to load now. Normal usage is that if only one OS is installed on the system bootstrap program is said to be available in the MBR or boot block. Otherwise they are said to be having boot loader. For example, if we install only DOS on a disk it contains 446 bytes long bootstrap program is seen in the boot block. Where as if Linux is installed, boot loader such as LILO or GRUB is available in boot area of the bootable partition. The actual job of transferring control to the operating system is performed by the second-stage boot loader (commonly referred to as simply the "boot loader"). This boot loader allows the user to choose the kernel to be loaded, loads the kernel and related parameters onto memory, initializes the kernel, sets up the necessary environment, and finally "runs" the kernel.

The next stage of booting is the kernel stage, when the kernel takes control. It sets up the necessary data structures, probes the devices present on the system, loads the necessary device drivers, and initializes the devices.

The kernel will begin initializing itself and the hardware devices for which support is compiled in. The process will typically include the following steps.

- Detect the CPU and its speed, and calibrate the delay loop
- Initialize the display hardware
- Probe the PCI bus and build a table of attached peripherals and the resources they have been assigned
- Initialize the virtual memory management system, including the swapper kswapd
- Initialize all compiled-in peripheral drivers; these typically include drivers for IDE hard disks, serial ports, real-time clock, non-volatile RAM, and AGP bus. Other drivers may be compiled in, but it is increasingly common to compile as stand-alone modules those drivers that are not required during this stage of the boot process. Note that drivers must be compiled in if they are needed to support the mounting of the root file system. If the root file system is an NFS share, for example, then drivers must be compiled in for NFS, TCP/IP, and low-level networking hardware.
- The kernel can then run the first true process (called /sbin/init. Depending on your vendor and system, the *init* utility is located in either /etc or /sbin.) to the root file system (strictly speaking, kswapd and its associates are not processes, they are kernel threads)., although the choice can be overridden by supplying the boot= parameter to the kernel at boot time. The init process runs with uid zero (*i.e.,* as root) and will be the parent of all other processes. Note that kswapd and the other kernel threads have process IDs but, even though they start before init, init still has process ID 1. This is to maintain the UNIX convention that init is the first process.
- This init process uses the configuration file /etc/inittab information and creates terminal handling activity (process) and checks the integrity of file systems, mounts file systems, sets up swap partitions (or swap files), starts system services.

Content of /etc/inittab :

```
`       #
        # inittab   This file describes how the INIT process should set up
        #           the system in a certain run-level.
        #
        # Author: Miquel van Smoorenburg,
        #           Modified for RHS Linux by Marc Ewing and Donnie Barnes
        #
        # Default runlevel. The runlevels used by RHS are:
        # 0 - halt (Do NOT set initdefault to this)
        # 1 - Single user mode
        # 2 - Multiuser, without NFS (The same as 3, if you do not have networking)
        # 3 - Full Multiuser mode
        # 4 - unused
        # 5 - X11
        # 6 - reboot (Do NOT set initdefault to this)
        #
        id:5:initdefault:
        # System initialization.
        si::sysinit:/etc/rc.d/rc.sysinit
        l0:0:wait:/etc/rc.d/rc 0
```

```
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
# Things to run in every runlevel.
ud::once:/sbin/update
# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
# When our UPS tells us power has failed, assume we have a few minutes
# of power left.  Schedule a shutdown for 2 minutes from now.
# This does, of course, assume you have powered installed and your
# UPS connected and working correctly.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"
# If power was restored before the shutdown kicked in, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
# Run xdm in runlevel 5
# xdm is now a separate service
x:5:respawn:/etc/X11/prefdm -nodaemon
```

The above file contains records with specific structure. Here is an explanation of them.

- The first field us just a descriptor or identifier and should kept as unique.
- The 2nd field is which runlevel this entry applies to.

A runlevel is a state for the system. Usually, you have runlevels 0,1,2,3,4,5,6 and additional levels are also supported in other systems.

For example, runlevel 1 (or S) usually means a single shell running, as few processes as possible, maybe no login, maybe just asking for root's password. While runlevel 5 may mean 6 logins in text mode, a graphical login, and a web server running. The system starts, when init loads in an undefined state (sometimes called N), and then will switch to one runlevel or another depending on what the runlevel argument to the boot loader to the kernel was, and the contents of /etc/inittab.

- The 3rd field seems to be some specific keyword that /sbin/init understands such as wait, respawn, once, etc given as :
  - boot — The process is started only on bootup and is not restarted if it dies. init doesn't wait for it to complete running before continuing to the next command and can run many processes simultaneously. This action is rarely used.

- — bootwait — The process is started only on bootup, and init waits for it to finish running and die before continuing. It doesn't restart the process once it finishes or dies. Notice that line 2 of the listing employs bootwait with a utility to mount and check file systems.
- — off — If the process is currently running, a warning signal is sent and after 20 econds, the process is killed by the dreaded kill -9 command. Line 16 shows that when the run level is changed to 2 (multiple user), terminal 1 is killed. The user who was logged on is now logged off and must log on again — adding a level of security that keeps the root user from changing run levels and then walking away from the terminal, thereby giving access to anyone who happens to sit there.
- — once — When the specified run level comes, the process is started. init doesn't wait for its termination before continuing and doesn't restart it if it dies. Like boot, once isn't used very often.
- — ondemand — This action has same meaning as respawn but is used mostly with a, b, and c levels (user defined). See respawn, below, for more information.
- — powerfail — The action takes place only when a power failure is at hand. A signal 19 is the most common indication of a power failure. Usually, the only action called by a powerfail is a sync operation.
- — powerwait — When a power failure occurs, this process is run and init waits until the processing finishes before processing any more commands. Again, sync operations are usually the only reason for the action.
- — respawn — This action restarts the process if it dies after it has been started. init doesn't wait for it to finish before continuing to other commands. Notice in lines 8–15 that respawn is the action associated with the terminals. Once they are killed, you want them to respawn and allow another login.
- — syncinit — Not available on all systems, this action tells init to reset the default sync interval, which is the interval, in seconds, between times the modified memory disk buffers are written to the physical disk. The default time is 300 seconds, but it can be set to anything between 15 and 900.
- — sysinit — Before init tries to access the console, it must run this entry. This action is usually reserved for devices that must be initialized before run levels are ascertained. Line 1 shows that the TCB — used for user login and authentication — is initialized even before the console is made active, allowing any user to log on.
- — wait — This action starts the process at the specified run level and waits until it completes before moving on. It is associated with scripts that perform run-level changes. You want them to fully complete operation before anything else happens. Notice that lines 4–7 use this action for every run level change.

• The 4th field the program/script that is to be called along with any parameters.

Some the items in /etc/inittab are given and explained their use in the following paragraphs.

si::sysinit:/etc/rc.d/rc.sysinit

This line calls /etc/rc.d/rc.sysinit.

Also note that any "wait" lines will wait until the system has booted before they start.

This rc.sysinit loads hostname, starts system logs, loads keyboard keymap, mounts swap partitions, initializes usb ports, checks file system, and mount the filesystems read/ write.

After /sbin/init finishes with /etc/rc.d/rc.sysinit (which was specified by the "sysinit" line), it then switches to the default runlevel (which is defined by the "initdefault" line in / etc/inittab).

Changing runlevels should leave any processes running that are in both the old and new runlevels.

Scripts prefixed with S will be started when the runlevel is entered, eg /etc/rc5.d/ S99xdm :

- Scripts prefixed with K will be killed when the runlevel is entered, eg /etc/rc6.d/ K20apache
- X11 login screen is typically started by one of S99xdm, S99kdm, or S99gdm.

1:2345:respawn:/sbin/getty 9600 tty1

- Always running in runlevels 2, 3, 4, or 5
- Displays login on console (tty1)

2:234:respawn:/sbin/getty 9600 tty2

- Always running in runlevels 2, 3, or 4
- Displays login on console (tty2)

l3:3:wait:/etc/init.d/rc 3

- Run once when switching to runlevel 3.
- Uses scripts stored in /etc/rc3.d/

ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now

- Run when *control-alt-delete* is pressed

Usually on those terminals which are defined in /etc/inittab, *getty* prompts for the user's login name. Then, *login* prompts the user to type his/her password by printing a prompt. If the user enters the password (which does not appear on the screen) and the password is incorrect, the system responds with a generic message. In reality, the *login* command accepts the password typed by the user and encrypts it using the same mechanism as the *passwd* command uses to put the password in the /etc/passwd file. If the encrypted values match, the password is correct. Otherwise, the password is incorrect. The *login* command can't decrypt the password once it has been encrypted. When the password is typed properly, the login process enters the next phase. The next phase of the process starts after the user has typed the correct password for the login. This phase establishes the environmental parameters for the user. For example, the user's login shell is started, and the user is placed in the home directory. The *init* command starts the user's login shell as specified in the /etc/passwd file. The user's initial environment is configured, and the shell starts executing. Once the shell is started, the user executes commands as desired. When the user logs off, the shell exits, *init* starts up *getty* again, and the process loops around.

To know What Is Running and How Do We Change It?

The *-r* parameter of the *who* command shows the run level at which your machine is currently operating as well as the two most recent previous run levels. For example,

```
who -r

run level 2 May 4 10:07 2    1     0
```

The above command shows that the current run level is 2 and has been since May 4 at 10:07. On some systems, the three numbers to the right show the current run level, the previous run level, and the next previous run level. On other systems, the three numbers represent the process termination status, process ID, and process exit status.

Changing run levels requires root permission and can be done with either the *init* or the *shutdown* command.

During system reboot, the boot loader stage is preceded by a shutdown of the previously running system. This involves terminating running processes, writing back cache buffers to disk, unmounting file systems, and performing a hardware reset.

The *shutdown* command, on the other hand, is usually in /usr/sbin. The *init* command is very simple. It lets you specify a number behind it and the machine then changes to that run level. For example

```
init 3
```

Immediately begins changing the machine to run level 3.

The *shutdown* command interacts with *init* and offers more parameters and options. A *-g* option lets you specify a grace period of seconds to elapse before beginning the operation (the default is 60), *-i* signifies which run level you want to go to, and *-y* carries out the action without asking for additional confirmation. Thus, to change to run level 3 in 15 seconds, the command would be

```
shutdown -g15 -i3 -y
```

Once the command is typed, a warning message is broadcast telling users that the run level is changing (this is true with *init* as well). The system then waits the specified number of seconds — giving users the chance to save files and log off — before making the change. Contrast this with *init* command, which tells users that the run level is changing and immediately begins changing it without giving them time to prepare.

```
init 6
```

This command also makes the system to shutdown properly.

11.2 USERS PROCESSES

As mentioned earlier, PID of the init process is 1. This process starts terminal handling processes (such as getty, mingetty, agetty, uugetty) on each of the lines mentioned in the /etc/inittab file. Thus, these processes becomes child processes to init process. In UNIX system, child processes PID will be larger than parent. When a user log's in with legal user name and password, getty process will die in place of it shell will become active. Thus, on some terminals on which user is logged in shell processes will be running where as on other terminals getty process will be running. We can check by running the commands "ps –al" or "ps –Al".

The command "ps" displays details of the processes running on the current terminal and which belongs to the user.

```
PID TTY       TIME CMD
1175 tty1     00:00:00 bash
1283 tty1     00:00:00 ps
```

The command "ps -Al" displays details of all the processes running on the system. For brevity reasons, only few lines of the output only displayed here.

```
F S  UID  PID   PPID  C  PRI  NI  ADDR SZ  WCHAN  TTY     TIME  CMD
4 S  0    1     0     1  75   0   -        343    schedu  ?     00:00:04  init
4 S  0    1103  1     0  82   0   -        336    schedu  tty3  00:00:00 mingetty
4 S  0    1104  1     0  82   0   -        336    schedu  tty4  00:00:00 mingetty
4 S  0    1105  1     0  82   0   -        337    schedu  tty5  00:00:00 mingetty
4 S  0    1106  1     0  82   0   -        337    schedu  tty6  00:00:00 mingetty
4 S  0    1175  1101  0  76   0   -        1091   wait4   tty1  00:00:00 bash
4 S  0    1229  1102  1  85   0   -        1089   schedu  tty2  00:00:00 bash
4 R  0    1284  1175  0  81   0   -        791    -       tty1  00:00:00 ps
```

We can observe from the above output that on terminals tty1 and tty2, we have logged in. Thus, Shell (bash) is running. Where as on other terminals, mingetty is running. Also, please note that the command ps is also became as a process while gathering information about the processes. Also, note that bash is in sleeping state while "ps" is running. Also, note that PPID's of mingetty and bash are same. That is both of them are child processes of init process. When we login with valid user name and password mingetty will die and in place bash (shell) becomes active and is also child to init process whose PID is 1.

When we run any piping command, each command will be made as a process.

Run the following command.

```
ps -Al|more|tail -4|tee aa
```

```
4 S   0  1229   1102   0   85   0   -   1089   schedu  tty2    00:00:00 bash
4 R   0  1326   1175   0   80   0   -   792    -       tty1    00:00:00 ps
0 S   0  1327   1175   0   76   0   -   926    pipe_w  tty1    00:00:00 more
0 S   0  1328   1175   0   76   0   -   926    pipe_w  tty1    00:00:00 tail
0 S   0  1329   1175   0   76   0   -   852    pipe_w  tty1    00:00:00 tee
```

The programs written by the users also becomes as processes when we start them. When a user enters a command at the dollar prompt it will be first received by the shell then it parses the command and identifies from where input has to be taken and to where output has to be sent. Then it calls a system call known as fork() which in turn returns PID of a new process which resembles the shell. Now shell assigns the duty to this new process to run the command typed by the user and goes to sleeping state while the new child process starts continuing the assigned duty. When it completes or encounters an error it indicates the same to the parent (shell) and then dies. Thus, in UNIX systems processes will be getting created and completing the assigned duties.

For example, compile and run the following program "aa.c".

```c
#include<stdio.h>
int main()
{
/* This program is an infinite loop program doing nothing. */
while(1);
}
```

To Compile

```
gcc -o aa aa.c
```

To Run

```
aa or ./aa
```

As the above program is infinite loop program we will not see dollar prompt. By pressing* ALT + F2 or other function keys F3, F4, F5 or F6 we can get another terminal. Login into it and run the "ps -Al" command. We find the following line.

```
0 R   0  1371   1175   96  85   0    -     335    -       tty1    00:00:57 aa
```

The line shows that process "aa" is running on terminal tty1 since 57 seconds.

With the help of kill command we can kill any process. Of course, only legal owner of the process can kill his process (exception for super user).

For example, you can run from another terminal the following command to kill the process "aa". After executing command, press ALT + F1 to goto tty1 and their you will find the message "Killed".

```
kill -9 1371
```

In the above command, the number 9 is known as a signal number. Usually, when we press some key sequences such as ctrl +c or ctrl +d etc., some special SW signals are sent to the current process. They are called as SW interrupts. Please do not confuse with events of current days programming languages such as Java, etc. These SW interrupts or signals are like processor interrupts (which arrives from peripherals and which runs their service routines), and these signal's arrivals also makes processes to run some programs known as signal handlers. For example, when we press ctrl + c, the process terminates. In UNIX terminology, this ctrl + c is also called as SIGINT. Similarly, there are many signals available in UNIX system and each signal has its default behavior. If one wants, we can make to run some other program when a signal arrives to a process and this is known as signal handling. We can make some signals to be ignored by a process. However, not all the signals to be made ignored by a process. One such a signal is signal number 9 or SIGQUIT which is called as uninterruptible signal. That is, it will be delivered to the process at any cost and the default action is going to take place. This signal's default action is to kill the process. Thus, when we run the above command, process "aa" which is an infinite loop program will be terminated.

We can logout by killing the bash (shell) process. For example,

```
kill -9 1229
```

11.2.1 Background and Foreground Processes

UNIX supports background processes. To run any command in background, simply we have to append & while running the command. It displays job ID and PID of the background process in responses.

For example execute :

```
ls &

aa &
```

We can checkup that the process "aa" is running by typing "ps -Al" command.

A process is said to be in background process, if its parent shell can accept another command to the user. That is its parent shell is Running state. A process is said to be in foreground process if its parent shell is in sleeping state. That is, it (shell) can not take any more commands from the user.

If we happened to have a dumb terminal (normally used in old UNIX flavors) and you can have only one terminal we have in it then it is not possible to enjoy the benefit of multi tasking as shell can normally take one command at a time. Thus, by using background concept, we can start a program and put it in the background such that the shell can take another command from the user.

However, if we can not make a program which requires interactive input to be in background; we may get error message such as "stopped tty output".

Try at the command prompt the following command.

```
vi  filename&
```

For a background process, key board (standard input) is not logically connected thus the programs which requires interactive input can not be kept in the background. If we want them to be run in background, then we have to create a data file which contains the required data for that program and then start this program specifying through input redirection operator (<) that the program is supposed to take necessary data from the data file. For example :

```
program <datfile &
```

Also, output of a background process appears on to the same terminal from which it is invoked. It may be possible that this output may mingle with current foreground process on that terminal and may make the screen messy. In order to take care of this situation, the background process can be started such that its output is sent to a file. For example :

```
program >output &
```

For example, consider the following program whose executable file name as "bb".

```
    #include<stdio.h>
    int main()
    {
    while(1) printf("1");
}
```

This program continuously prints 1's. To know the effect of the output of a background process, execute the following commands at command prompt one after another.

```
sleep 10

bb &

vi filename
```

We may find, even if we don't type anything, 1's will be coming on to the vi editor screen. We may find difficult to type anything into the file. Of course, when we save finally these 1's will not be saved into the file. However, vi editor working becomes clumsy because of this background process. Thus it is better to redirect output of a background process.

In total, if we want a process which requires to be kept in background and needs interactive input and gives standard output then the same can be started in the following manner.

```
program <inputfile >outputfile &
```

For example, consider the following C program which takes three integers and writes their values.

```
#include<stdio.h>
void main()
{
int x,y,z;
scanf("%d%d%d", &x, &y, &z);
printf("%d\n%d\n%d\n", x, y, z);
}
```

Let the file name be a.c and by using the either of the following commands, its machine language file a is created.

gcc -o a a.c

cc -o a a.c

When we start this program **a** by simply typing a at the dollar prompt, it takes 3 values and displays given values on the screen.

a>res &

We should get an error.

```
cat>res
10
20
11
^d
```

This program takes three values interactively and writes the same into file res. We can check by typing cat res.

a<res &

This should give results on the screen.

a <res >ass &

This command takes necessary input from the file res and displays the results in the file "ass".

If we want a piping command to be kept in background, then for each component of the piping sequence we have to append &.

command1&|command2&|command&

A background process gets killed if its parent (shell) dies. That if we logout. However, if we start a background process prepended with nohup it is continue to run even if logout.

For example

nohup command &

Similarly, if we want a piping command we want run in background and continue to run even after we log out then we have start the same in the following manner.

nohup command1&| nohup command2&| nohup command&

Job Control

Job control refers to the ability to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. A user typically employs this facility via an interactive interface supplied jointly by the system's terminal driver and Bash. With the help of the following commands we can convert a background job to foreground, and vice versa. Bash also supports keyboard shortcuts such as ^z and ^y to achieve the same with limited control.

The bg command: This command usage is :

```
bg [jobspec ...]
```

Resume each suspended job *jobspec* in the background, as if it had been started with '&'. If *jobspec* is not supplied, the current job is used. The return status is zero unless it is run when job control is not enabled, or, when run with job control enabled, any *jobspec* was not found or specifies a job that was started without job control.

The fg command: This command usage is :

```
fg [jobspec]
```

Resume the job *jobspec* in the foreground and make it the current job. If *jobspec* is not supplied, the current job is used. The return status is that of the command placed into the foreground, or non-zero if run when job control is disabled or, when run with job control enabled, *jobspec* does not specify a valid job or *jobspec* specifies a job that was started without job control.

The jobs Command: This can be used in either of the following styles.

```
jobs [-lnprs] [jobspec]
jobs -x command [arguments]
```

The first form lists the active jobs. The options have the following meanings :

-l List process ids in addition to the normal information.

-n Display information only about jobs that have changed status since the user was last notified of their status.

-p List only the process id of the job's process group leader.

-r Restrict output to running jobs.

-s Restrict output to stopped jobs.

If *jobspec* is given, output is restricted to information about that job. If *jobspec* is not supplied, the status of all jobs is listed.

If the -x option is supplied, jobs replaces any *jobspec* found in *command* or *arguments* with the corresponding process group id, and executes *command*, passing it *argument*s, returning its exit status.

In all the above commands, we have to specify about the job with the jobspec. There are a number of ways to refer to a job in the shell. The character '%' introduces a job specification in bash.

Job number n may be referred to as '%n'. The symbols '%%' and '%+' refer to the shell's notion of the current job, which is the last job stopped while it was in the foreground or started in the background. A single '%' (with no accompanying job specification) also refers to the current job. The previous job may be referenced using '%-'. If there is only a single job, '%+' and '%-' can both be used to refer to that job. In output pertaining to jobs (e.g., the output of the jobs command), the current job is always flagged with a '+', and the previous job with a '-'.

A job may also be referred to using a prefix of the name used to start it, or using a substring that appears in its command line. For example, '%ce' refers to a stopped ce job. Using '%?ce', on the other hand, refers to any job containing the string 'ce' in its command line. If the prefix or substring matches more than one job, Bash reports an error.

Simply naming a job can be used to bring it into the foreground: '%1' is a synonym for 'fg %1', bringing job 1 from the background into the foreground. Similarly, '%1 &' resumes job 1 in the background, equivalent to 'bg %1'

The wait command: This commands format is :

wait [*jobspec* or *pid* ...]

Wait until the child process specified by each process id *pid* or job specification *jobspec* exits and return the exit status of the last command waited for. If a job spec is given, all processes in the job are waited for. If no arguments are given, all currently active child processes are waited for, and the return status is zero. If neither *jobspec* nor *pid* specifies an active child process of the shell, the return status is 127.

The disown command: This command usage style is :

disown [-ar] [-h] [*jobspec* ...]

Without options, each *jobspec* is removed from the table of active jobs. If the -h option is given, the job is not removed from the table, but is marked so that SIGHUP is not sent to the job if the shell receives a SIGHUP. If *jobspec* is not present, and neither the -a nor -r option is supplied, the current job is used. If no *jobspec* is supplied, the -a option means to remove or mark all jobs; the -r option without a *jobspec* argument restricts operation to running jobs.

The suspend command: This command usage syntax is :

suspend [-f]

Suspend the execution of this shell until it receives a SIGCONT signal. A login shell cannot be suspended; the -f option can be used to override this and force the suspension.

When job control is not active, the kill and wait built-ins do not accept *jobspec* arguments. They must be supplied process ids.

11.2.2 At command

UNIX also supports a facility known as at with the help of which we can instruct the UNIX to start a program at a specified time on a specified date. It needs a file having the commands to be executed on that date and time.

For example, the file "xxx" contains the following statements.

```
aa>pp
lpr pp
rm pp
```

To run the above commands on Nov 30 at 4pm the following command can be used.

at -f xxx 4pm Nov 30

To see what jobs are submitted to at command we can execute command "atq".

With the help of "atrm" command we can remove a submitted job from at commands queue.

11.2.3 The date command

This command displays current date.

```
$ date
Sun May 17 11:12:55 IST 2009
$ date –u   Displays Time Universal Time format
Sun May 17 05:42:57 UTC 2009
$ date +%m   Displays Month Only
05
$ date +%d%m%y   Displays Day Month Year
170509
$ date +%H   Displays Hour Only
11
$ date +%H%M%S   Displays Hour Minutes Seconds
111340
```

11.2.4 Time command

Sometimes, we may need to know how much time a program is taking. This can be known with the help of "time" command.

Example

```
time ls
```

This command displays three times to name, user time, system time and elapsed time.

- User time is the actual CPU time consumed by the users program.
- System time is the CPU time consumed by the OS on behalf of the users program while administering the system such as allocating memory, resources, CPU etc.
- Elapsed time is the time elapsed between the instant of starting a program and till we seen dollar prompt again.

User time is most important one. When we want to compare two programs we may use their user times only.

11.2.5 A Note on Identification Numbers Used in Linux Systems

Process UID and GID

In order for the operating system to know what a process is allowed to do it must store information about who owns the process (UID and GID). The UNIX operating system stores two types of UID and two types of GID.

Real UID and GID

A process' real UID and GID will be the same as the UID and GID of the user who ran the process. Therefore any process you execute will have your UID and GID.

The real UID and GID are used for accounting purposes.

Effective UID and GID

The effective UID and GID are used to determine what operations a process can perform. In most cases the effective UID and GID will be the same as the real UID and GID.

However using special file permissions it is possible to change the effective UID and GID. How and why you would want to do this is examined later in this chapter.

11.3 TERMINAL HANDLING

Since its development, UNIX systems are equipped with terminals which may be connected to machine via serial lines such as RS232. These terminals may use different control sequences while communicating with UNIX system via serial line driver. This serial driver which may perform some low-level conversions (handling ^c, ^d characters for flow control and translating DEL and ERASE characters) on what we type from the terminal before it is passed to the program which we are running. As there is no standard among the plethora of terminals, a large part of satisfactory terminal emulation is carried out at the host operating system.

Both "termcap" and "terminfo" contain some features for allowing programs to know what Escape sequences to expect from a terminal type, although not every UNIX program uses these features. For example /etc/termcap contains specifications about various terminals which users can use to log into the system. This information is used by terminal-emulation programs to adjust what the individual keys transmit.

The "term" and/or "TERM" environment variables are typically used to tell the system which records to look up in terminfo or termcap. The man page for your user shell should describe how these may be set.

You should be able to do at least

echo $TERM

to see the current setting.

To find out the serial-port/pseudo-terminal parameters, the "stty -a" command can be used, e.g.,

```
stty -a
speed 38400 baud;
rows = 25; columns = 80; ypixels = 0; xpixels = 0;
eucw 1:0:0:0, scrw 1:0:0:0
intr = ^c; quit = ^|; erase = ^?; kill = ^u;
eof = ^d; eol = <undef>; eol2 = <undef>; swtch = <undef>;
start = ^q; stop = ^s; susp = ^z; dsusp = ^y;
rprnt = ^r; flush = ^o; werase = ^w; lnext = ^v;
-parenb -parodd cs8 -cstopb -hupcl cread -clocal -loblk
-crtscts -crtsxoff -parext
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl -iuclc
ixon -ixany -ixoff imaxbel
isig icanon -xcase echo echoe echok -echonl -noflsh
-tostop echoctl -echoprt echoke -defecho -flusho -pendin iexten
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel tab3
```

We can also use either of the following commands to know the terminal information.

stty

stty -everything

In the above commands output last line contains - before some words are see indicating the respective terminal characters are set. Where as others are not set.

If we wanted to change terminal behaviors we can use command stty. For example, the best way to set the number of rows and columns displayed is by using the "stty" command :

stty rows 24 cols 80

Also we can change "termcap" entry for a given terminal to achieve the same effect.

stty –echo

Now whatever we type at dollar prompt will not appear on the screen. If we enter the following command then terminal characteristics will return to previous style.

stty echo

For example try the following also.

stty –echo; cat >destfile; stty echo

Now you can type whatever you want and press at the end ^d as usual. The destfile contains what we have typed.

Try the following and identify what happens.

stty –echo; cat >destfile

Also, If we execute reset command (or stty sane) at the dollar prompt then terminal behavior returns to previous style.

Run the following command sequences to know the effect of cbreak mode.

```
tty cbreak
cat
<type whatever you wanted>
^d
```

We may find when we enter enter key afresh line will be appearing.

For example when we execute the following command at the dollar prompt then end of file (eof) become ^a.

stty eof \^a

To see the effect try to create a file using cat command. By pressing ctrl + a we are able to stop giving input to cat command.

```
cat >filename
Adsdasds
Asdkjdsa
Asdkjds
Adsds
^a
```

Similarly, we can make ctrl + b as ctrl + c, we can run the following command.

stty intr \^b

11.3.1 Reading Verrrry Long Lines from the Terminal

Sometimes, you want a very long line of input to write a file. It might come from your personal computer, a device hooked to your terminal, or just an especially long set of characters that you have to type on the keyboard. Normally the UNIX terminal driver holds all characters you type until it sees a line terminator or interrupt character. Most buffers have room for 256 characters.

If you're typing the characters at the keyboard, there's an easy fix: Hit CTRL-d every 200 characters or so to flush the input buffer. You won't be able to backspace before that point, but the shell will read everything in.

Or, to make UNIX pass each character it reads without buffering, use stty to set your terminal to cbreak (or non-canonical) input mode.

For example :

```
% stty cbreak

% cat > file

enter the very long line.........

[CTRL-c]

% stty -cbreak
```

Run the following command sequence to know the effect of raw mode. You may find cat command not responding to ^d and ^c signals also!!

```
stty cbreak
cat
<type whatever you wanted>
^d
```

While you're in cbreak mode, special keys like BACKSPACE or DELETE won't be processed; they'll be stored in the file. Typing CTRL-d will not make cat quit. To quit, kill cat by pressing your normal interrupt key - say, CTRL-c.

who command

This displays details about the users such as user name, terminal on which working and since when they are working.

```
root    :0      Feb  9 00:22
root    pts/0   Feb  9 00:25 (:0.0)
root    pts/1   Feb  9 20:25 (:0.0)
```

w command

This displays details about the users in addition to what command they are working now.

w

Presents who users are and what they are doing in the following fashion.

```
USER   TTY     FROM    LOGIN@      IDLE    JCPU   PCPU    WHAT
root   :0      -       12:22am ?   0.00s   1.66s  /usr/bin/gnome-
root   pts/0   :0.0    12:25am     0.00s   0.81s  0.03s   w
```

w username

Displays what that user is doing.

w -i

displays details sorted by idle time

The last command

last	This shows who logged in and when
last -20	This shows only the last 20 logins
last -20 -a	This shows last 20 logins, with the hostname in the last field
last rao	This shows when user rao has logged in last time.

last -a > /root/lastlogins.tmp

This will print all the current login history to a file called lastlogins.tmp in /root/

11.4 ULIMIT COMMAND

UNIX system has resource limits such as limits on number of processes, maximum allowed file size, etc.

Example :

```
ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) 231122
file size               (blocks, -f) 231122
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 1016
virtual memory          (kbytes, -v) unlimited
```

ulimit

This command displays file size limit on the system currently.

ulimit -f 121212

This changes file size limit to 121212.

Similarly, we can change resource limits such as max data, text segment sizes etc.

11.5 CONCLUSIONS

This chapter explains about processes in Linux. How to make a process as background and foreground is explained. How to kill processes is explained by giving emphasis to Linux signals. At the end, commands such as at, and time are explained. A brief outline of terminal handling in Linux/UNIX is also included. Also, we have explained about system resource limits.