UNIT-8

The Process-The Meaning-Parent and Child Processes-Types of Processes-More about Foreground and Background processes-Internal and External Commands-Process Creation-The Trap Command-The Stty Command-The Kill Command-Job Control.

Because Unix is a multi-user as well as a multi-tasking system, number of programs can run simultaneously. All the programs that have been loaded into the memory for execution are referred to as processes.

1 THE MEANING

A process is defined as a program in execution. Unix being a multiuser and a multi-tasking system, there could be several programs belonging to different users or the same user running at the same time. All these programs share the same CPU. The kernel generates or spawns processes for every program under execution and allocates definite and equal CPU time slots to these various programs. Each of these processes have a unique identification number allocated to it by the kernel. Individual processes are identified by using these unique numbers, and are called process identification numbers or PIDs.

Mathematically, a process is represented by the tuple—

(*process id, code, data, register values, pc value*),

where process id (PID) is the unique identification number that is used to identify the process uniquely from other processes, code is the program code that is under execution, data is the data used during execution, register values are the values in CPU registers and PC value is the address in the program counter from where the execution of the program starts or continues. At present the maximum value of PID is 32767.

As soon as the system is booted, the kernel gets loaded into the memory and then gets executed. Immediately, a system process called the swapper is created. The PID of this process will be 0 (zero). This process 0 creates another process called init, meaning initialiser. This init is one of the first programs that is loaded which starts running immediately after the bootstrapping. The PID of init process is 1. This init process is responsible for setting up or initialising all subsequent processes on the system. init sets the user mode in either the single-user or the multi-user mode. Also init is responsible for generating processes on log-ins. It (process 1) exists as long as the system is running and it is the ancestor of all other processes on the system.

2 PARENT AND CHILD PROCESSES

In Unix, a process is responsible for generating another process. A process that generates another process is called the parent of the newly generated process, called the child. For example, when a command like $cat sample.lst is given, the shell creates a process for running the cat command. Thus, the shell sh (ksh or bash) being a process, generates another process (cat). Here the shell process is the parent process and the cat process is the child process. When a parent process creates or generates a child process, a process is said to have born. As long as a process is active, it is said to be alive. Once the job of a process is over it becomes inactive and is said to be dead.

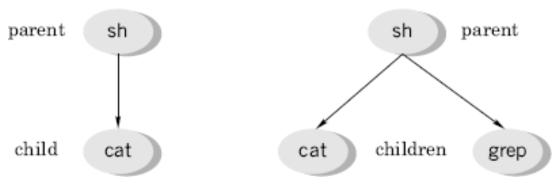


Fig 1 Parent and child processes

When a command like $cat sample.lst | grep lecturer is given to the shell, two processes, one for running the cat program and another for running the grep program are created simultaneously. Here, once again the shell process is the parent process and the cat and grep processes are its child processes. The cat and grep processes, which are the children of the same parent, will have different PIDs. All the child processes will inherit almost all the environmental parameters of their parent processes.

In general, a parent process waits for the complete execution of its child process—a parent waits for its child to die. However, sometimes a parent may die before its child. In such cases the child is said to be orphan. Generally these orphan processes are attached to the init process—the process with PID 1.

It should be noted that all the commands do not create processes. For example, running of the commands like cd, mkdir, pwd and others do not create processes.

2.1 Program and a Process

A careful observation of the parent and child processes and the relation between them (as discussed in the previous section) reveals that all processes get themselves arranged in the form of a hierarchical, inverted tree like structure. This is similar to file organization. The only difference between these two organizations is that file organization is *locational* whereas process organization is *temporal*.

A program exists in a single location in space and exists for any length of time. Thus a program is a static object that exists in a file. It contains just the entire set of instructions. But a process is a program in execution. Thus, it is a *dynamic object* and can never be in a file. It is a sequence of instructions under execution. Thus process has a definite life cycle.

3 TYPES OF PROCESSES

Processes within Unix are classified into three general categories—as *interactive processes, non-interactive processes* and *daemons*.

3.1 Interactive Processes—Foreground Processes

All the user processes, which are created by users with the shell, act upon the directions of the users and are normally attached to the terminal are called interactive processes. These types of processes are also called *foreground processes.*

3.2 Non-interactive Processes—Background Processes
Certain processes can be made to run independent of terminals. Such processes that run without any attachment to a terminal are called noninteractive processes. These types of processes are also called *background processes.*
3.3 Daemons
All processes that keep running always without holding up any terminals and keep waiting for certain instructions either from the system or the user and then immediately get into action are called daemons. swapper, init, cron, bdflush, vhandle are some examples of daemons. These daemons come into existence as soon as the system is booted and will be alive till the system is shut down. One cannot kill these processes prematurely.

4 MORE ABOUT FOREGROUND AND BACKGROUND PROCESSES

When a command is given, the shell parses, rebuilds and then hands it over to the kernel for execution. The shell then keeps on waiting for the kernel to complete the execution. During this shell-waiting period the user cannot issue any other command because the terminal is held up with the command under execution. As already mentioned, commands that hold up the terminal during their execution are called foreground processes. The chief disadvantage of foreground processing is that, no further commands can be given from the terminal as long as the older one is running. This disadvantage becomes significant when a currently running process is big and takes a lot of time for processing.

It is possible to make processes to run without using the terminal. Such processes take their input from some file and process it without holding up the terminal (non-interactively), and write their output on to another file are called background processes. Typical jobs that could be run in background are sorting of a large database file or locating a file in a big file system by using the find command and so on.

4.1 Running a Command in the BackgroundA command is made to run in the background (as a background process) by terminating the command line with an ampersand (&) character as shown in the following example.
```
 $sort –o student.lst student.lst &
567
$
```
The shell immediately returns the process identification (PID) number as well as the shell prompt $. In the above example, 567 is the PID of the just-submitted background job. As the shell prompt ($) re-appears immedietly, one can now readily work at the terminal. One should be careful in running background processes as the user may get into problems under certain situations. Some of these problems could be due to any one of the following.

1. The success or failure of the background processes are not reported. The user has to find it out. For this purpose the identification number is used.
2. The output has to be redirected to a file as otherwise the display on the monitor gets mixed up.
3. Too many processes running in the background degrades the overall efficiency of the system.

4. There is a danger of the user logging out when some processes are still running in the background.

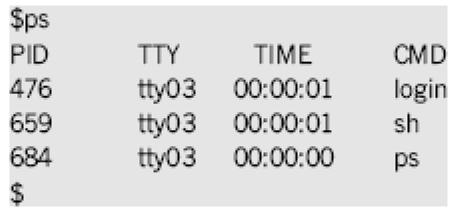
5 INTERNAL AND EXTERNAL COMMANDS

The classification of commands depending on whether they generate separate processes or not upon their running is discussed here. Most of the commands such as cat, who and others generate separate processes as soon as they are used. Commands that generate separate processes upon their running are called external commands. Some commands such as mkdir, rm, cd and others do not generate new processes when they are used; such commands are called internal commands.

6 THE ps COMMAND—KNOWING PROCESS ATTRIBUTES

The ps command is used to display the attributes of processes that are running currently. This is one of the commands that varies too much from one system to another. This comes with a number of options like –a (all users), –f (full list), –u (user), –t (terminal) and –e (every).

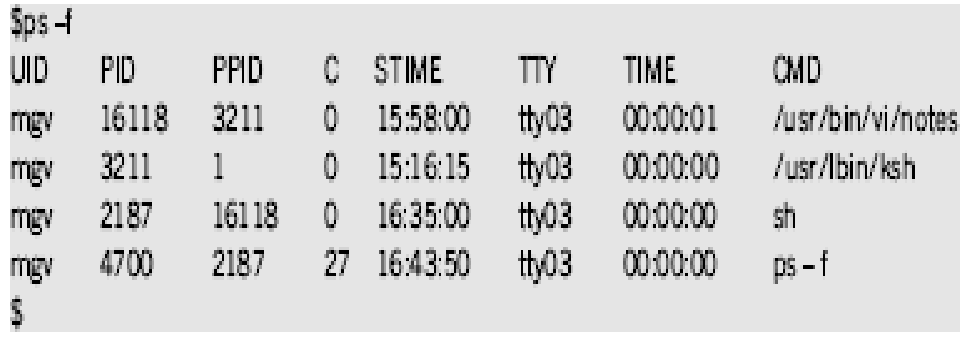
When used with *no option*, the ps command lists out certain attributes associated with the terminal as shown below.



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

where PID = process identification number
TTY = terminal type
TIME = cumulative time
CMD = command

A *full listing* of the processes can be obtained by using the –f option with the ps command, as shown below. As seen from the example on the next page, using this option, one can trace the ancestry of different processes also.



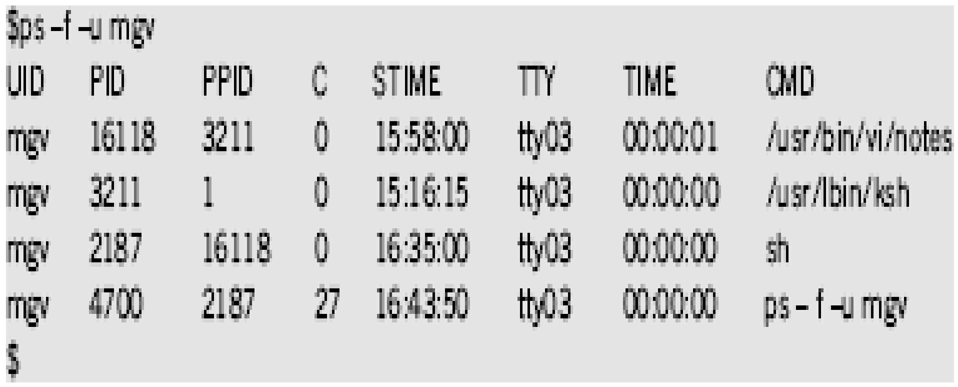
```
$ps –f
UID    PID      PPID     C    STIME      TTY      TIME       CMD
mgv    16118    3211     0    15:58:00   tty03    00:00:01   /usr/bin/vi/notes
mgv    3211     1        0    15:16:15   tty03    00:00:00   /usr/lbin/ksh
mgv    2187     16118    0    16:35:00   tty03    00:00:00   sh
mgv    4700     2187     27   16:43:50   tty03    00:00:00   ps – f
$
```

where UID = user ID

PPID	=	parent process ID
STIME	=	starting time
C	=	CPU time consumed

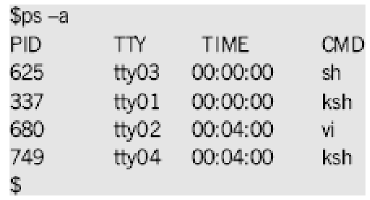
Here the user mgv uses vi to edit a file named notes. The ksh is the users' login shell since its parent process id is 1. ksh is the parent of vi/notes, and so on.

All the process of a particular user only can be listed by using the –u option along with the user-ID as an argument to the ps command, as shown below.



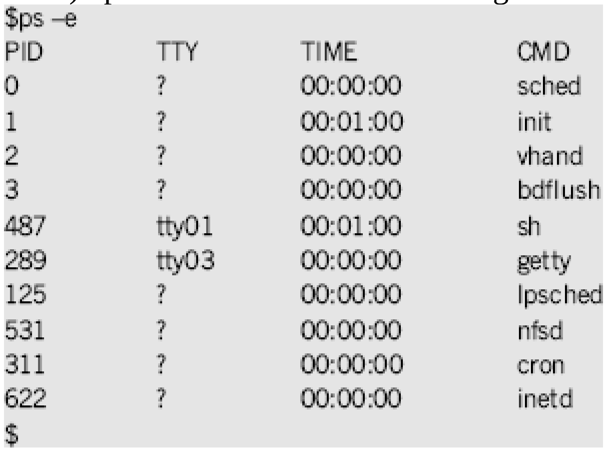
```
$ps –f –u mgv
UID   PID     PPID    C   STIME     TTY     TIME      CMD
mgv   16118   3211    0   15:58:00  tty03   00:00:01  /usr/bin/vi/notes
mgv   3211    1       0   15:16:15  tty03   00:00:00  /usr/lbin/ksh
mgv   2187    16118   0   16:35:00  tty03   00:00:00  sh
mgv   4700    2187    27  16:43:50  tty03   00:00:00  ps – f –u mgv
$
```

The process of all the users only (not the system processes) can be listed by using the –a option as shown in the following example.



```
$ps –a
PID     TTY     TIME      CMD
625     tty03   00:00:00  sh
337     tty01   00:00:00  ksh
680     tty02   00:04:00  vi
749     tty04   00:04:00  ksh
$
```

All the processes including the system processes are listed using the ps command along with the –e (every process) option as shown in the following illustration.



```
$ps –e
PID     TTY     TIME      CMD
0       ?       00:00:00  sched
1       ?       00:01:00  init
2       ?       00:00:00  vhand
3       ?       00:00:00  bdflush
487     tty01   00:01:00  sh
289     tty03   00:00:00  getty
125     ?       00:00:00  lpsched
531     ?       00:00:00  nfsd
311     ?       00:00:00  cron
622     ?       00:00:00  inetd
$
```

The appearance of a question mark (?) in the TTY column indicates that these are system processes. In the above listing, bdflush is the buffer to disk flushing activity support routine, nfsd is the network file system daemon, inetd is the internet daemon without which the TCP/IP does not work, vhand is the system routine that handles virtual memory management implementations and so on.

It should be noted that system processes support activities of the system and keep on doing their task, independent of what users are doing, as long as the system is on.

7 PROCESS CREATION

There are three distinct phases in the creation of a process. They are (1) forking, (2) overlaying and execution and (3) waiting. These three phases are taken care of by making calls to the system routines fork(), exec() and wait(), respectively.

Forking is the first phase in the creation of a process by a process. The calling process (parent) makes a call to the system routine fork() (the call here is referred to as a system call) which then makes an exact copy of itself. The copy will be of the memory of the calling process at the time of the fork() system call and not of the complete program the calling process was started with. Right after the fork() there will be two processes with identical memory images. Each one of these two processes has to return from the fork() system call. Thus there will be two return values. The fork of the parent process returns the PID of the new process, that is the child process just created, whereas the fork of the child returns a 0 (zero). Incase a new child process is not created a –1 is returned.

Immediately after forking, the parent makes a system call to one of the wait() functions. By doing so, the parent keeps waiting for the child process to complete its task. It awakens only when it receives a complete signal from the child, after which it will be free to continue with its other functions.

The child process inherits almost the entire environment of the calling process. In other words, the child process will have the same priority, same signal handling settings, same group and user ids, same current directory and so on. However, children will not inherit the local variables and will have different PID's.

In the second phase, the parent makes a system call to one of the exec() functions. This system call simply overwrites the text and data area of the child process by the text and data of the new program and then starts executing this new program. At the end of the overlaying and execution, a call is made to the exit() function that terminates the child and sends a signal back to the parent after which, the parent becomes free to continue with its other functions. The entire mechanism of process creation is pictorially shown in Fig. 7.2.

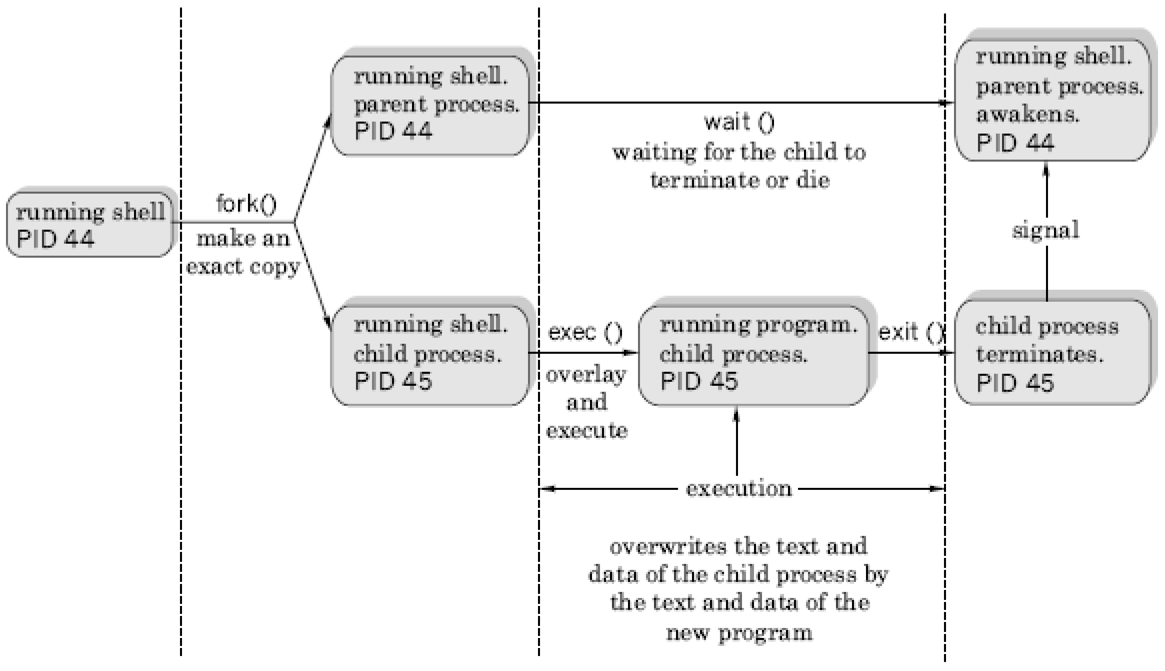

running shell
PID 44
fork()
make an exact copy
running shell. parent process. PID 44
wait ()
waiting for the child to terminate or die
running shell. parent process. awakens. PID 44
signal
running shell. child process. PID 45
exec ()
overlay and execute
running program. child process. PID 45
exit ()
child process terminates. PID 45
execution
overwrites the text and data of the child process by the text and data of the new program

Fig..2 Mechanism of process creation

8 SIGNALS

A signal is a message sent to a program under execution, that is a process, on one of the following two occasions.

1. Under some error conditions or the user interruption, the kernel generates signals.
2. During interprocess communication between two or more processes. The participating processes generate these signals. For example, a child process sends a signal to its parent process upon its termination.

In Unix, signals are identified by integers. They have names too. These names are in uppercase and start with SIG. There are about 30 such signals, numbered from 1. Some commercial implementations like AIX have more signals. The table below gives a list of exit or interrupt signals.

Signal Number	***Name***	***Function***
1	SIGHUP	Hangup; closes process communication links.
2	SIGINT	Interrupt; tells process to exit.(Ctrl–c)
3	SIGQUIT	Quit; forces the process to quit. (Ctrl–\)
9	SIGKILL	Sure kill; cannot be trapped or ignored.
15	SIGTERM	Software termination; default signal for the kill command.
24	SIGSTOP	Stop; (Ctrl–Z)

9 THE trap COMMAND

Normally signals are used to prematurely terminate the execution of a process either intentionally or unintentionally. The trap command is used to trap one or more signals and then decide about the further course of action. If no action is mentioned, then the signal or signals are just trapped and the execution of the program resumes from the point from where it had been left off. The general format of this command is given below.

```
 $trap [commands] signal_numbers
```

The commands part is optional. When it is present, all the commands present in this part are executed one by one as soon as the process receives one of the signals specified in the signal_numbers list. The commands used, must be enclosed using either single or double quotation marks. Multiple commands in the commands part are separated by the; (semicolon) character. Following are some examples.

```
 (i) $trap "echo killed by signal 15; exit" 15
```

When the process receives a kill command, causing signal 15, the above command first gives the message killed by signal 15 and then terminates the current process because of the execution of the exit command.

```
 (ii) $trap "ls –l" 1 2 3
```

When the process generates anyone of the signals 1, 2 or 3, a long listing of the current working directory is generated and then execution of the process resumes from the point where it had been left off.

```
 (iii) $trap " " 1 2 3 15
```

This command just traps the signal numbers 1, 2, 3 and 15.

Though majority of the signals can be trapped, certain signals like signal number 9 (the sure kill) cannot be trapped. Given below is a simple script that keeps on running till the user interrupts it by using the interrupt key.

```
 $cat –n sample.trap
1 trap "echo PROGRAM INTERRUPTED; exit 1" 2
2 while true
3 do
4  echo "program running."
5 done$
```

Resetting Traps Normally a trap command changes the default actions of the signals. Using the trap command, without the commands' part, changes the specified signals to their default actions. This behaviour of the trap command is useful under certain situations. For example, one might need to trap a certain signal in one part of a script and need the same signals not to be trapped in some other part. The command to trap the signal will be as shown below.

```
$trap "exit" 2 3 15
```

The effect of the signals 2, 3 and 15 are restored by using the trap command without the command part in it as shown in the following example.

```
 $trap 2 3 15
```

10THE stty COMMAND

One of the most widely used methods to communicate with a system is to use terminals, that is via keyboards. There are certain combination of keys, on these terminals, which control the behavior of any program in execution. For example, we have been using

1. <Ctrl–m>(^m), that is the <RETURN> key to end a command line and execute the command.
2. <Ctrl–c>(^c) to interrupt a current process and to come back to the shell.
3. <Ctrl–s>(^s) to pause display on the monitor.
4. <Ctrl–d>(^d) to indicate end of file and so on.

The stty command is used to see or verify the settings of different keys on the keyboard. The user can have a short listing of the settings by using this command without any arguments. In order to see all the settings, it has to be used with the –a (all) option, as shown in the following example.

```
 $stty –a
speed 9600 baud; ispeed 9600 baud; line = 0(tty);
erase = ^?; kill = ^U; eof = ^D; intr = ^C ; stop = ^S;
echo echoe – – – – – – – – – – – – – – – – – – – –
$
```

The output shown above is just illustrative. From the output one can see that the terminal speed is 9600 bauds, ^U is used for killing a line, ^D is used to indicate end of file, because of echo everything typed at the keyboard gets echoed on the display terminal, backspacing over a character retains its display, and so on.
This command can also be used to change the key settings as shown in the following examples.

```
 $stty –echo
$stty eof \^a
```

Execution of the former command, stops the display of characters that are typed at the keyboard. It may be noted that this is the setting used to handle passwords. After the execution of the latter command, the use of <Ctrl–a> terminates all standard input.
It is recommended not to play around with the terminal settings. This may lead to improper working of the terminal. However, if the user finds that the terminal is not working properly, he

or she may restore the sanity into terminal settings by using the word sane as a single argument to stty, as shown below. **$stty sane**

The execution of the above command sets the terminal settings with reasonable values.

<u>**11 THE kill COMMAND**</u> There are certain situations when one likes to terminate a process prematurely. Some of these situations are as follows.

- When the machine has hung.
- When a running program has gone into an endless loop.
- When a program is doing unintended things.
- When the system performance goes below acceptable levels because of too many background processes.

Terminating a process prematurely is called killing. Killing a foreground process is straightforward. This is done by using either the DEL key or the BREAK key. However, to kill a background process the kill command is used. This command is given with the PID of the process to be killed as its argument. If the PID is not known the ps command is used to know the same.

For example, a process having an identification number 555 can be killed using the kill command as shown in the following example.

```
$kill 555
```

More than one process can be terminated using a single kill command as shown in the following example.

```
$kill 330 333 375          # here 330,    333,    375 are process id's.
```

A kill command, when invoked, sends a termination signal to the process being killed. When used without any option, it sends 15 as its default signal number. This signal number 15 is known as the software termination signal and is ignored by many processes. For example, the shell process sh, ignores signal 15. In other words, signal 15 does not guarantee the killing of all processes. At such times, one can use signal number 9, the sure kill signal, to terminate a process forcibly as shown in the following example.

```
$kill –9 666            # 666 is the id number of the process
```

All the processes of a user (except his login shell) can be terminated by using a 0 (zero) as the argument of the kill command as shown in the following example.

```
$kill 0            # kill all the processes except the login shell
```

However using 9 as option and 0 (zero) as the argument, all processes including the shell can be killed as shown in the following example.

```
$kill –9 0          # kills all processes including the login shell
```

$! and $$ System Variables The special variable $! holds the PID value of the last background job, and the special variable $$ holds the PID value of the current shell. The last background job can be killed using the command $kill $!. The current shell can be killed using the sure kill command $kill –9 $$.

THE wait COMMAND

With some shells like the korn and bash, jobs can be run in the background as background processes. Sometimes it is necessary to wait for either all the background jobs or a specific job to be executed completely before any further action is initiated. Under such circumstances, the wait command is used for waiting background process(s) to be completely executed. Some examples are given here.

```
 $wait #waits till all the background processes are completely executed
$wait 227 #waits for the completion of the process with PID 227
```

12 JOB CONTROL Unix is a multi-tasking system and there will be many number of jobs or processes running simultaneously in a Unix environment. Quite often, it will be required to know

1. how many as well as which processes are running currently,
2. terminate either a misbehaving or a unwanted process,
3. modify the priority of a process,
4. to push a process into the background,
5. to bring up a required process to the foreground, and so on.

The above-listed type of activities is generally referred to as job-control activities. In Unix, there exist many commands by using which, one can perform any of the job-control activities. For example, the ps command is used to know details of currently running processes. The kill command is used to prematurely kill (or terminate) a process, the wait command is used to make a process to wait till its child is terminated, and so on. All these commands are available with all the shells including the Bourne Shell and have already been discussed. Other job-control commands such as jobs, fg and bg, which are available in Korn and some other recent shells (not with the Bourne Shell), have been discussed in the following sections.

.1 Job Control Commands—The jobs, fg and bgA command or a command line with a number of commands put together or a script is generally referred to as a job. In Unix, as one can run commands in the background, there could be a number of commands, that is, processes, running in the background. Also there could be a command—a process—running in the foreground.

The jobs **Command** A list of all the current jobs is obtained using the jobs commands as shown below.

```
[ksh]jobs        #The{ksh}prompt has been used intentionally
[1] + Running sort emp.data|grep `Bangalore`>address.lst &
[2] – Runningsleep 1000 &
```

```
[ksh]
```

In the above output, a + (plus) and – (minus) that appear after the job number mark the current and previous jobs, respectively. The word running indicates that the job is currently being executed. The alternate information that could appear in this position are stopped, suspended, terminated, done and exits. The output also displays the command name. After knowing the status of the jobs running in the background one may take any required action like bringing a job to the foreground, killing a job and so on.

The fg **Command** This command is used to bring a job that is being executed in the background currently to the foreground. This command can be either used without any argument or with a job number as its argument. Some simple illustrations are given here.

```
[ksh]fg           # Brings the most recent background process
                 # to the foreground
[ksh]fg %2        # Brings job number 2 to the foreground
[ksh]fg %sort     # Brings the job the name of which begins
                 # with sort to the foreground
```

As seen from the above examples, whenever a job number is used as an argument with a job-control command (not necessarily with fg only) it must be preceded by a percent sign (%). Here it may be noted that the current job may be referred to by using any one of the representations—%1 or %+ or %%. Also, it may be noted that first few characters of a command sequence can be used to refer to job as shown in the last example in the previous set of illustrations.

The bg **Command** A new job can be made to run in the background by using the & (ampersand) at the end of a command line as discussed in Section 7.4.1. The question here is how to make a currently running foreground process to run in the background? The answer is very simple. The currently running foreground process is first suspended, by using the <ctrl–z> keys, and then making it to run in the background by using the bg command. By assuming that the currently running process has been suspended right now, the following command line puts it in the background.

```
 [ksh]bg %1    # resumes job number 1 in background
```

<u>SCHEDULING JOBS' EXECUTION</u>

Normally, commands or programs are executed by using a suitable command line as and when required by typing them at the system prompt. In Unix, it is possible to get commands executed at any required time, whenever the system is relatively free and repeatedly according to certain requirement.

Commands such as at, batch and cron are used for scheduling execution of commands according to requirements.

The at Command—Running a Command at a Future Date and Time

This command is capable of executing Unix commands at a future date and time. The input to this command has to come from the standard input. In other words, commands may be typed in through the keyboard or may be provided through a file.

```
$at 17:00
clear > /dev/tty03
echo "It is 5 P.M. Back up your files and logout" > /dev/tty03
<ctrl–d>
job 801346789.a at Fri Jan 11 17:00:00 IST 2002
$
```

Once a job is submitted using the at command, details regarding the job id number, the date and time at which commands are to be executed are displayed. The job id number is based on number of seconds elapsed since the beginning of 1970. Note that neither the PID nor the filename of the process are displayed. One has to be extra careful in monitoring the jobs that are scheduled when using this command. It should be observed that the job id terminates with a .a. If the output of the at command is not redirected as shown in the above example, the output will arrive at the terminal as a mail at the scheduled time.

Once the command is submitted in the above-mentioned manner, the message will be displayed on the terminal at 5 pm sharp.

The time can use am and pm suffixes. If these suffixes are not given, the time will be taken in the 24-h format. Keywords like *now, noon, midnight, today, tomorrow, hours, days, weeks, months* and *years* can be used with this command. A list of some typical examples are given below.

```
 $at 1 pm today
$at noon
$at 15
$at 10 am tomorrow
$at now + 1 year
```

A file can be given as an argument to an at command using the –f option as shown in the following example. `$at –f scriptfile 7 am Monday`

The information regarding jobs that are scheduled using at will be available on a queue called the at queue. The details of this can be obtained using the –l option as shown below.

```
$at –l
889673410.a Wed Dec 31   15:08:00 2003
.. ... ...... .... ....
```

.. ..

A job scheduled with at command can be removed prematurely by using the –r option. For this, one has to remember and use the job id as shown in the following example.

```
$at –r 889673410.a
```

The batch Command

Jobs submitted by using this command are executed when the system is relatively free and the system load is light. Since the time at which the commands are executed is decided by the system, there is no need to specify the time. An example is given here.

```
$batch
sort emp.dat   |   grep `Bangalore` > address.lst
<ctrl–d>
job 6423 22445.b at Fri Jan 16 17:00:00 IST 2004
$
```

The extension .b attached to the job identification number indicates that it has been submitted by using the batch command. Jobs scheduled using this command also sit in the at queue.

The cron Daemon and the crontab Command

The term cron is derived from the word chronograph. Using this facility one can schedule required jobs to run periodically. Cron is a system daemon that keeps sleeping most of the time. It typically wakes up once every minute and checks its crontab file for any jobs to be executed during this minute. All users have a crontab file of his or her own. The name of this crontab file will be the user’s login name. Scheduled jobs will be present in the crontab file. crontab files will be present in the /var/spool/cron/crontabs directory.

A crontab file may contain one or more lines, each corresponding to a command that is to be executed periodically at a specified day, date and time. Figure 7.3 gives the basic syntax of a line on a crontab file. Every such line will be made up of six fields with each field separated by a blank.

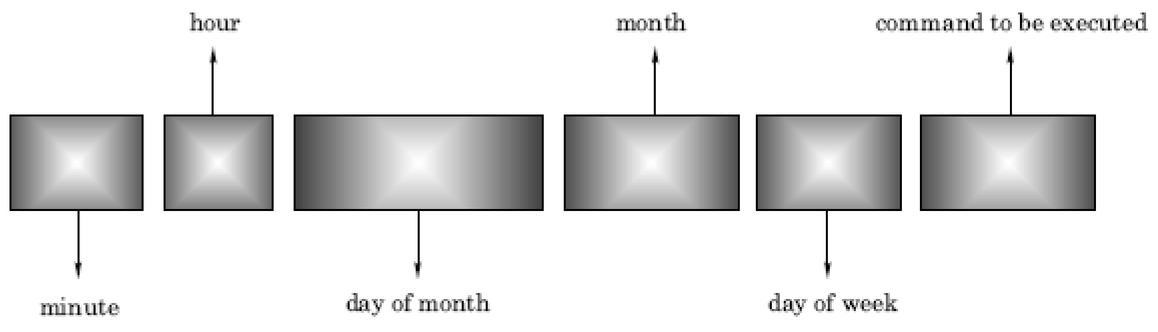


Fig. 7.3 Syntax of a crontab line

As seen from the figure above, the first field specifies the minute (0–59), the second field specifies the hour (24-h format), the third field specifies the day of the month (1–31), the fourth field specifies the month (1–12), the fifth field specifies the day of the week (0–6), 0 being Sunday, and the sixth field contains the command line to be executed. In a crontab line, an asterisk (*) represents all possible values. For example, if a * character appears in the fifth field, then the command mentioned in the line will be executed on all the days of the week at the specified time. If necessary groups of numbers can be specified within a single field by separating them with commas. No spaces are allowed within a field. Below are given two typical crontab lines.

1. 0 0 * * * backup.sh
   When executed, the above line runs the backup.sh script at midnight everyday.
2. 00,30 09–17 * * 1–5 mail.sh
   When executed, the above line runs the mail.sh script on all weekdays—Monday to Friday every half hour between 9 and 17 hours.

From the above examples, one can see that a crontab line not only contains commands to be repeatedly executed but also the details of date and time in a specific format.

When more than one command has to be periodically executed, every command has to be written in a separate line having the above format in a separate file. Then this file is submitted to the crontab command, as shown in the example below, where cmdfile is the name of the file that contains the command lines that are to be executed periodically.

```
$crontab cmdfile
```

When a file is submitted by using the crontab command, its contents are automatically transferred to the /var/spool/cron/crontabs directory. The crontab command when used without any argument accepts the input from the standard input—the keyboard. As usual, the input operation from the keyboard has to be terminated using <ctrl–d> keys. A careless use of this method removes all the entries on the existing crontab file. One has to be extra careful while entering crontab lines via the keyboard.

The contents of the crontab file can be seen using the $crontab –l command. A submitted file can be removed using the –r option as in $crontab –r command. It may be noted that here job name or job id is not required, as every user will have just one crontab file of his or her own.

THE nohup COMMAND

One of the dangers of running processes in the background is that, if a user logs out with one or more processes still running in the background, the running processes will have an unnatural death. Further, processes that need large processing time are run in the background. In such cases, rather than waiting, one can logout deliberately keeping the process running, come back and get the results. In either of the two cases, the user can keep running the process in the background till they are completed even when s/he logs out (not that the system is shut down). In other words, the user does not want the system to hang up the background process. This is accomplished by using a command called the nohup command, as shown in the following example.

```
$nohup sort –o students.lst students.lst &
79
$
```

Once a command is submitted with nohup one can logout without the process getting terminated on logging out. As shown in the above example, the output filename has to be mentioned. If it is not mentioned, the output will be stored in a file called nohup.out by default. Further, whenever commands are piped, each command should be qualified by the nohup command, as shown in the following example. This is because, every command in the pipeline spawns a process of its own.

```
 $nohup cat students.lst|nohup grep `murthy`|nohup sort > names.lst &
86
$
```

THE nice COMMAND Processes in Unix have equal priority. In Unix, the priority of a process is measured by using integer numbers ranging from 0 to 39 (in Linux this ranges is –19 to +20). A 0 (zero) indicates the highest priority whereas 39 indicates the lowest priority. The default value of the priority assigned to a process upon its creation is 20 in the Bourne shell (0 [zero] in the case of Linux). The priority of a process can be reduced to a lower-than-normal value by using the nice command as shown in the following example. Default value of reduction is 10 units.

```
 $nice big_program
```

This command runs big_program with a priority value reduced by 10 units, that is, with a priority value of 30. When the priority of a command is reduced it uses less CPU time and runs slower. It is possible to reduce the priority of a job by using a number option along with the nice command, as shown in the following example.

```
$nice –19 big_program
```

The above-mentioned command runs the file big_program with a priority value of 39. The following command lowers the priority and runs the command in the background.

```
$nice –19 big_program &
```

Users cannot increase the priority of a job. If such a facility is given, everyone likes to run his or her job with the highest priority. However, the system administrator or the supervisor can raise the priority of a process by using the nice command, with double minus option (– –), as shown in the following example.

```
#nice – –12 big_program
```

The above command runs the file big_program with a priority value of 8. The # character in this command line indicates that this command is issued by the supervisor.

THE time COMMAND This command is used to know the resource usage. It runs a program or command with given arguments, generates a timing statistics about the program run and directs this statistics report to the standard output. This statistics consists of the elapsed time between invocation and termination, the user CPU time and the system CPU time. By analyzing this, one can assess the efficiency of a program or command

```
 $find   /   –name   makefilex –print
real     0m14.509s
user     0m0.150s
```

```
sys     0m0.390s
$
```