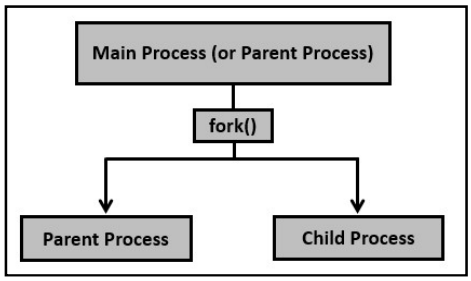
04 – Process creation and Management

Process

- A **process** is any active (running) instance of a program. In other words, process is a program in execution.
- A new process can be created **by the fork() system call**.
- The new process consists of a copy of the address space of the original process. fork() creates new process from existing process.
- Existing process is called the **parent process** and the process is created newly is called **child process**.



- Each process is given a unique process identification number (PID).
- PID is usually five-digit number.
    - This number is used to manage each process.
    - user can also use the process name to manage process.

Starting a Process

A process executes in two ways they are,

- Foreground Processes
- Background Processes

Foreground Processes

- Every process by default runs in Foreground (which means the output is printed on the screen.) The best example for the foreground process is the **ls** command which prints the output on the screen by listing the files and directories.
- When a program is running in the foreground, user cannot start another process without completing the previous process. [It is because of this reason foreground process is considered a time-consuming process.]

Background Processes

- When a process starts running and it is not visible on the screen it is called a background process.
- User can simultaneously run 'n' number of commands in the background process.
- To enable the background process, provide ampersand symbol (**&**) at the end of the command.
- **Example : ls &**

ps

The default output of ps is a simple list of the processes running in current terminal.

- **Example: ps**

  PID TTY TIME CMD

  23989 pts/0 00:00:00 bash

  24148 pts/0 00:00:00 ps

- Every running process (**-e**) and a full listing (**-f**) could be obtained with these options.

=== * ===

Demonstrate fork() system call

Example 1:

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int global;        // A global variable

int main(void)
{
   pid_t childPID;
   int local = 0;

   childPID = fork();

   if(childPID >= 0)   // fork was successful
   {
      if(childPID == 0)  // child process
      {
         local++;
         global++;
         printf("\n Child Process :: local = %d,
global %d\n", local, global);
      }
      else  //Parent process
      {

         local = 10;
         global = 20;

         printf("\n Parent process :: local = %d,
global %d\n", local, global);
      }
   }
   else  // fork failed
   {
      printf("\n Fork failed, quitting!!!!!!\n");
      return 1;
   }

   return 0;
}
```

Output:

Parent process :: local = 10, global 20

Child Process :: local = 1, global 1

=== * ===

Demonstrate exec( ) system call

The exec() family of functions replaces the current process image with a new process image. **exec** command in Linux is used to execute a command from the **bash** itself. This command does not create a new process it just replaces the bash with the command to be executed. If the exec command is successful, it does not return to the calling process.

exec > output.txt

- Redirect all output to the file **output.txt** for the current shell process.
- Redirections are a special case, and **exec** does not destroy the current shell process, but **bash** will no longer print output to the screen, writing it to the file instead.

Example 1: (try this command in sudo bash)

$ exec > output.txt
$ cal
$ echo i am trying command exec
$ date
$ pwd
$ echo no commands display output on screen
$ echo enjoy exec command
$ exit
exit
	Output:
	cat output.txt
	March 2022
	Su Mo Tu We Th Fr Sa
	1 2 3 4 5
	6 7 8 9 10 11 12
	13 14 15 16 17 18 19
	20 21 22 23 24 25 26
	27 28 29 30 31
	i am trying command exec
	Tue Mar 29 14:35:47 IST 2022
	/home/admincs
	no commands display output on screen
	enjoy exec command

=== * ===

Example 2: (try this command in sudo bash)

- **exec 3< output.txt** // no space between **3 and <**
- Open **output.txt** for reading ("<") on file descriptor **3**.
- The above command is an example of explicitly opening a file descriptor.
- After running the above command, user can read a line of **output.txt** by running the **read** command with the **–u** option:

- read –u 3 mydata
    - Here, "**-u 3**" tells **read** to get its data from file descriptor 3, which refers to **output.txt**. The contents are read, one line at a time, into the variable **mydata**. This would be useful if used as part of a **while** loop, for example.
- **exec 3< output.txt** // Assume contents of output.txt is **March 2022**
- **$ read –u 3 oneline**
- **$ echo $oneline**
    - **March 2022**

=== * ===

Example 3: (try this command in sudo bash)

- **exec 4> out.txt**
- The above command opens **out.txt** for writing (">") on file descriptor **4**.

=== * ===

Example 4: (try this command in sudo bash)

- Closing read and write descriptors
- **exec 4>&–** // closing write descriptor
- **exec 3<&–** // closing read descriptor

=== * ===

Example 5:

- **exec 6>> append.txt**
- Open append**.txt** for appending (">>") as file descriptor **6**.

$ exec 6>> append.txt
$ echo append line one >&6
$ echo append line Two >&6
$ echo append line Three >&6
	Output:
	$ cat append.txt
	append line one
	append line Two
	append line Three

=== * ===

bg: background

- **bg** used to place foreground jobs in background. [used to send a process running in the foreground to the background in the current shell.]

Syntax: bg [job_spec]

- Place the jobs identified by each *job_spec* in the background, as if they has been started with '&'.
- *Job_spec* may be
  - %n : Refer to job number n.
  - **%% or %+** : Refer to the current job.
    %- : Refer to the previous job.
- **Example: sleep 500** is a command which is used to create a dummy job which runs for 500 seconds.
  - Use jobs command to list all jobs
  - Create a process using sleep command, get job id as 1
  - Put that process in background using id
- **$ jobs** // Display current jobs running and displayed nothing
- **$ sleep 500** // Create one job, which sleeps for 500 seconds.
- ^z // Terminate the job by pressing Ctrl + z
  - [1]+ Stopped sleep 500
- **$ jobs** // Display current jobs running and displayed one job stopped
  - [1]+ Stopped sleep 500
- **$ bg %1** // Refer the process by job number **place it in background**
  - **[1]+ sleep 500 &**
- $ jobs // Display current jobs running and displayed one job which is running.
  - [1]+ **Running** sleep 500

=== * ===

fg: forground

- **bg** is a command that moves a background process on current Linux shell to the foreground.

Syntax: bg [job_spec]

- Place the jobs identified by each job_spec in the foreground, making it the current job.

Example: sleep 500 is a command which is used to create a dummy job which runs for 500 seconds.

- $ jobs
- $ sleep 500
- ^z

- [1]+ Stopped sleep 500
- $ jobs
  - [1]+ Stopped sleep 500
- $ bg %1
  - [1]+ sleep 500 &
- $ jobs
  - [1]+ **Running** sleep 500
- **$ fg %1** // Brings the background process (referred by its number) **to foreground**
- **sleep 500** // waiting for the process to terminate.

=== * ===

nohup

- **nohup**, short for **no hang up** is a command in Linux systems that **keep processes running even after exiting the shell or terminal**.
- nohup prevents the processes or jobs from receiving the SIGHUP (Signal Hang UP) signal.
- This is a signal that is sent to a process upon closing or exiting the terminal.
- Every command in Linux starts a process at the time of its execution, which automatically gets terminated upon exiting the terminal.
- Suppose, user executing programs over SSH and if the connection drops, the session will be terminated, all the executed processes will stop, and user may face a huge accidental crisis.
  - In such cases, running commands in the background can be very helpful to the user and this is where **nohup command** comes into the picture. **nohup (No Hang Up)** is a command in Linux systems that runs the process even after logging out from the shell/terminal.
- Usually, every process in Linux systems is sent a **SIGHUP (Signal Hang UP)** which is responsible for terminating the process after closing/exiting the terminal.
- *nohup* command prevents the process from receiving this signal upon closing or exiting the terminal/shell.
- Once a job is started or executed using the nohup command, **stdin** will not be available to the user and **nohup.out** file is used as the default file for **stdout** and **stderr**.

- If the output of the nohup command is redirected to some other file, **nohup.out** file is not generated.

Syntax: **nohup command [command-argument ...]**

- nohup command can be used to run multiple commands in the background.
- Syntax: nohup bash -c 'commands'

Example 1:

- **nohup bash -c 'cal && ls'**

```
yash9274@YASH-PC:~/GFG$ nohup bash -c 'cal && ls -l'
nohup: ignoring input and appending output to 'nohup.out'
yash9274@YASH-PC:~/GFG$ ls
nohup.out
yash9274@YASH-PC:~/GFG$ cat nohup.out
   November 2020
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

total 0
-rw------- 1 yash9274 yash9274 188 Nov 25 23:34 nohup.out
yash9274@YASH-PC:~/GFG$
```

- Here, the output will be by default stored in **nohup.out**. To redirect it, type:
- **nohup bash -c 'commands' > filename.txt**

Example 2:

- **nohup bash -c 'cal && ls' > output.txt**

```
yash9274@YASH-PC:~/GFG$ nohup bash -c 'cal && ls -l' > output.txt
nohup: ignoring input and redirecting stderr to stdout
yash9274@YASH-PC:~/GFG$ ls
output.txt
yash9274@YASH-PC:~/GFG$ cat output.txt
   November 2020
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

total 0
-rw-r--r-- 1 yash9274 yash9274 188 Nov 25 23:42 output.txt
yash9274@YASH-PC:~/GFG$
```

=== * ===

Stopping a process

kill

- kill is used to send a signal to a process. The most commonly used signal is "terminate" (SIGTERM) or "kill" (SIGKILL). The full list can be shown with **kill –L**
  - 1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
  - 6) SIGABRT 7) SIGBUS 8) SIGFPE 9) **SIGKILL** 10) SIGUSR1
  - 11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) **SIGTERM**

- Signal number nine is SIGKILL. The default signal is 15, which is SIGTERM.
  - Issue a command such as **kill -9 20896**.

pkill

- command used to send signal to kill process by process name.
- while issuing pkill, make sure that selected process name is the correct process to be stopped, verify it by its full path by issuing pgrep –f processname.
- Syntax: **pkill -f ProcessName**

=== * ===

nice

- **nice** command in Linux **helps in execution of a program/process with modified scheduling priority**.
- It launches a process with a user-defined scheduling priority.
- The **renice command** allows user to change and modify the scheduling priority of an already running process.
- Linux Kernel schedules the process and allocates CPU time accordingly for each of them.
- The kernel stores a great deal of information about processes including process priority which is simply the scheduling priority attached to a process.
- Processes with a higher priority will be executed before those with a lower priority, while processes with the same priority are scheduled one after the next, repeatedly.
- There are a total of **140** priorities and two distinct priority ranges implemented in Linux.
- The first one is a nice value (**niceness**) which ranges from **-**20 (highest priority value) to 19 (lowest priority value) and the default is 0.
- The other is the real-time priority, which ranges from **1** to **99** by default, then **100** to **139** are meant for user-space.

Check Nice Value of Linux Processes

ps -eo pid, ppid, ni, comm

- pid process-id, ppid parent process-id, ni niceness of process, comm command for that process
- Alternatively, user can use **top** or **htop** utilities to view Linux processes nice values
- in htop command output**, PRI** – is the process's actual priority, as seen by the Linux kernel.
- **rt** value in PRI fields, means the process is running under **real-time** scheduling priority

Important:

- If no value is provided, nice sets a priority of 10 by default.
- A command or program run without nice defaults to a priority of zero.
- Only root can run a command or program with increased or high priority.
- Normal users can only run a command or program with low priority.

Syntax:

$ nice -n niceness-value [command args]

OR

$ nice -niceness-value [command args] **#it's confusing for negative values**

OR

$ nice --adjustment=niceness-value [command args]

Example:

$ sudo nice -n 5 tar -czf backup.tar.gz ./Documents/*

OR

$ sudo nice --adjustment=5 tar -czf backup.tar.gz ./Documents/*

OR

$ sudo nice -5 tar -czf backup.tar.gz ./Documents/*

Change the Scheduling Priority of a Process

- Linux allows dynamic priority-based scheduling. Therefore, if a program is already running, user can change its priority with the **renice command** in this form:

- $ renice -n -12 -p 1055 //Mention process with id (-p)
- $ renice -n -2 -u apache // Mention user process with name (-u)
  // -g for group process priority
- **Output Verification**:
  - New priority for process 1055 is 20 – 12 = 8
  - New priority for process apache is 20 – 2 = 18
  - Observe new priority by issuing **top** or **htop** command.

=== * ===

top

- The top command has been around a long time and is very useful for viewing details of running processes and quickly identifying issues such as memory hogs.
- Its default view is shown below. **Example 1: top**

top - 11:56:28 up 1 day, 13:37, 1 user, load average: 0.09, 0.04, 0.03
Tasks: 292 total, 3 running, 225 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.1 us, 0.2 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 16387132 total, 10854648 free, 1859036 used, 3673448 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 14176540 avail Mem

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
17270	alan	20	0	3930764	247288	98992	R	0.7	1.5	5:58.22	gnome-shell
20496	alan	20	0	816144	45416	29844	S	0.5	0.3	0:22.16	gnome-terminal-
21110	alan	20	0	41940	3988	3188	R	0.1	0.0	0:00.17	top
1	root	20	0	225564	9416	6768	S	0.0	0.1	0:10.72	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.01	kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
7	root	20	0	0	0	0	S	0.0	0.0	0:00.08	ksoftirqd/0

- The update interval can be changed by typing the letter **s** followed by the number of seconds the user prefer for updates.
- To make it easier to monitor required processes, user can call top and pass the PID(s) using the **-p** option.

Example 2:

top -p20881 -p20882 -p20895 -p20896

```
Tasks:   4 total,   0 running,   4 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.8 us,  1.3 sy,  0.0 ni, 95.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16387132 total, 10856008 free,  1857648 used,  3673476 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 14177928 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
20881 alan      20   0   12016    348      0 S   0.0  0.0   0:00.00 nginx
20882 alan      20   0   12460   1644    932 S   0.0  0.0   0:00.00 nginx
20895 alan      20   0   12016    352      0 S   0.0  0.0   0:00.00 nginx
20896 alan      20   0   12460   1628    912 S   0.0  0.0   0:00.00 nginx
```

=== * ===

Commands to Schedule Tasks:

cron

- The **cron** is a software utility, offered by a Linux-like operating system that automates the scheduled task at a predetermined time.
- It is a **daemon process**, which runs as a background process and performs the specified operations at the predefined time when a certain event or condition is triggered without the intervention of a user.
- Dealing with a repeated task frequently is an intimidating task for the system administrator and thus he can schedule such processes to run automatically in the background at regular intervals of time by creating a list of those commands using cron.
- It enables the users to execute the scheduled task on a regular basis unobtrusively like doing the backup every day at midnight, scheduling updates on a weekly basis, synchronizing the files at some regular interval.
- Cron checks for the scheduled job recurrently and when the scheduled time fields match the current time fields, the scheduled commands are executed.
- It is started automatically from /etc/init.d on entering multi-user run levels.
- **Syntax:** **cron [-f] [-l] [-L loglevel]**

- The **crontab** (abbreviation for "cron table") is list of commands to execute the scheduled tasks at specific time. It allows the user to add, remove or modify the scheduled tasks.
- The crontab command syntax has **six fields** separated by space where the **first five** represent the **time to run the task** and **the last one is for the command**.
  - Minute (holds a value between 0-59)
  - Hour (holds value between 0-23)
  - Day of Month (holds value between 1-31)
  - Month of the year (holds a value between 1-12 or Jan-Dec, the first three letters of the month's name shall be used)
  - Day of the week (holds a value between 0-6 or Sun-Sat, here also first three letters of the day shall be used)
  - Command (**6th Field**)

The rules which govern the format of date and time field as follows:

- When any of the first five fields are set to an asterisk(*), it stands for all the values of the field. For instance, to execute a command daily, we can put an asterisk(*) in the week's field.
- One can also use a range of numbers, separated with a hyphen(-) in the time and date field to include more than one contiguous value but not all the values of the field. For example, we can use the 7-10 to run a command from July to October.
- The comma (,) operator is used to include a list of numbers which may or may not be consecutive. For example, "1, 3, 5" in the weeks' field signifies execution of a command every Monday, Wednesday, and Friday.
- A slash character(/) is included to skip given number of values. For instance, "*/4" in the hour's field specifies 'every 4 hours' which is equivalent to 0, 4, 8, 12, 16, 20.

Permitting users to run cron jobs:

- The user must be listed in this file to be able to run cron jobs if the file exists.
  - /etc/cron.allow
- If the cron.allow file doesn't exist but the cron.deny file exists, then a user must not be listed in this file to be able to run the cron job.
  - /etc/cron.deny

- **Note:** If neither of these files exists then only the superuser(system administrator) will be allowed to use a given command.

Example to Try:

- As cron processes will run in background, so to check whether scripts are executed or not, one way to check the log file i.e. /var/log/syslog
- Search for cron in syslog using command: **cat /var/log/syslog/ | grep cron**
- User can see all the entries, which shows, the scripted executed at a scheduled time mentioned in crontab file.

=== * ===

Simple Example:

File Name: task.sh path: **/home/admincs/**(say for example)

Contents:

```
echo  Welcome to Task Scheduler Demo
echo  Try date Command
date
echo  Try time Command
time
echo  List all file/folders in a home directory
ls
echo  End of Task Scheduler Demo
```

=== * ===

- Change the execution permission to task.sh

  chmod 764 task.sh

- Add the task to /etc/crontab (Task will be executed at 4.13pm, User can Change timings according to the requirements)

  13 16 * * * root sh /home/admincs/task.sh > /home/admincs/output.txt

- Save and Exit. And Wait till 4.13pm
- After that Check the output.txt by the command

  cat /home/admincs/output.txt

- Output of the task.sh is present in output.txt.
- Alternatively **/var/log/syslog** can be verified about the execution of the Task.

=== * ===

Example 1:

- Run /home/folder/gfg-code.sh every hour, from 9:00 AM to 6:00 PM, everyday.
  - **00 09-18 * * * /home/folder/gfg-code.sh**
- Run /usr/local/bin/backup at 11:30 PM, every weekday.
  - **30 23 * * Mon, Tue, Wed, Thu, Fri /usr/local/bin/backup**
- Run sample-command.sh at 07:30, 09:30, 13:30 and 15:30.
  - **30 07, 09, 13, 15 * * * sample-command.sh**

Example 2: Other Simple Example to Schedule user or System Tasks:

- **Note**: To schedule user task, the shell script path must be specified in place of **/path/to/script**
- Specify the interval at which user want to schedule the tasks using the cron job entries we specified earlier on.
- To reboot a system daily at 12:30 pm, use the syntax:
  - **30 12 * * * /sbin/reboot**
- To schedule the reboot at 4:00 am use the syntax:
  - **4 * * * /sbin/reboot**
- **Note:** The asterisk * is used to match all records
- To run a script twice every day, for example, 4:00 am and 4:00 pm, use the syntax.
  - **4,16 * * * /path/to/script**
- To schedule a cron job to run every Friday at 5:00 pm use the syntax:
  - **17 * * Fri /path/to/script** OR
  - **0 17 * * * 5 /path/to/script**
- If user wish to run cron job every 30 minutes then use:
  - ***/30 * * * * /path/to/script**
- To schedule cron to run after every 5 hours, run
  - ***/5 * * * /path/to/script**
- To run a script on selected days, for example, Wednesday and Friday at 6.00 pm execute:
  - **18 * * wed,fri /path/to/script**
- To schedule multiple tasks to use a single cron job, separate the tasks using a semicolon for example:
  - *** * * * /path/to/script1 ; /path/to/script2**

Example 3: Using special strings to save time on writing cron jobs

Some of the **cron** jobs can easily be configured using special strings that correspond to certain time intervals. For example,

- @hourly timestamp corresponds to 0 * * * *
    - It will execute a task in the first minute of every hour.
    - @hourly /path/to/script
- @daily timestamp is equivalent to 0 0 * * *
    - It executes a task in the first minute of every day (midnight). It comes in handy when executing daily jobs.
    - @daily /path/to/script
- @weekly timestamp is the equivalent to 0 0 1 * mon
    - It executes a cron job in the first minute of every week where a week whereby, a week starts on Monday.
    - @weekly /path/to/script
- @monthly is similar to the entry 0 0 1 * *
    - It carries out a task in the first minute of the first day of the month.
    - @monthly /path/to/script
- @yearly corresponds to 0 0 1 1 *
    - It executes a task in the first minute of every year and is useful in sending New year greetings
    - @monthly /path/to/script

=== * ===

at

- **at command** is a command-line utility that is used to schedule a command to be executed at a particular time in the future.
- Jobs created with **at command** are executed only once.
- The **at command** can be used to execute any program or mail at any time in the future.
- It executes commands at a particular time and accepts times of the form HH:MM to run a job at a specific time of day.
- In the command, expression like noon, midnight, teatime, tomorrow, next week, next Monday, etc. could be used with **at command** to schedule a job.

- Syntax: **at [OPTION...] runtime**

Working with at command

- Command to list the user's pending jobs:
  - **at -l**

    OR
  - **atq**

```
File  Edit  View  Search  Terminal  Help
cipers@cipers:~$ at -l
2       Wed Jun 24 03:57:00 2020 a cipers
cipers@cipers:~$ atq
2       Wed Jun 24 03:57:00 2020 a cipers
cipers@cipers:~$
```

- Schedule a job for the coming Monday at a time twenty minutes later than the current time:
  - **at Monday +20 minutes**
- Schedule a job to run at 1:45 Aug 12 2020:
  - **at 1:45 081220**
- Schedule a job to run at 3pm four days from now:
  - **at 3pm + 4 days**
- Schedule a job to shutdown the system at 4:30 today:
  - **echo "shutdown -h now" | at -m 4:30**
- Schedule a job to run five hour from now:
  - **at now +5 hours**

=== * ===

```
cipers@cipers:~$ at sunday +20 minutes
warning: commands will be executed using /bin/sh
at> echo "Hello World"
at> <EOT>
job 11 at Sun Jun 28 05:34:00 2020
cipers@cipers:~$ at -l
11      Sun Jun 28 05:34:00 2020 a cipers
cipers@cipers:~$ echo "Hello World" | at 1pm + 2 days
warning: commands will be executed using /bin/sh
job 12 at Fri Jun 26 13:00:00 2020
cipers@cipers:~$ at -l
11      Sun Jun 28 05:34:00 2020 a cipers
12      Fri Jun 26 13:00:00 2020 a cipers
```

```
cipers@cipers:~$ mv p2.c p4.c | at 12:30 102120
warning: commands will be executed using /bin/sh
job 13 at Wed Oct 21 12:30:00 2020
cipers@cipers:~$ atq
11      Sun Jun 28 05:34:00 2020 a cipers
12      Fri Jun 26 13:00:00 2020 a cipers
13      Wed Oct 21 12:30:00 2020 a cipers
cipers@cipers:~$ gcc p1.c -o output | at now +1 hours
warning: commands will be executed using /bin/sh
gcc: error: p1.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
job 14 at Wed Jun 24 06:16:00 2020
cipers@cipers:~$ gcc p4.c -o output | at now +1 hours
warning: commands will be executed using /bin/sh
job 15 at Wed Jun 24 06:17:00 2020
cipers@cipers:~$ at -l
14      Wed Jun 24 06:16:00 2020 a cipers
11      Sun Jun 28 05:34:00 2020 a cipers
12      Fri Jun 26 13:00:00 2020 a cipers
15      Wed Jun 24 06:17:00 2020 a cipers
13      Wed Oct 21 12:30:00 2020 a cipers
```

7. at -r or atrm command is used to deletes job , here used to deletes job 11.

at -r 11 OR atrm 11

```
cipers@cipers:~$ at -l
14      Wed Jun 24 06:16:00 2020 a cipers
11      Sun Jun 28 05:34:00 2020 a cipers
12      Fri Jun 26 13:00:00 2020 a cipers
15      Wed Jun 24 06:17:00 2020 a cipers
13      Wed Oct 21 12:30:00 2020 a cipers
cipers@cipers:~$ at -r 12
cipers@cipers:~$ at -l
14      Wed Jun 24 06:16:00 2020 a cipers
11      Sun Jun 28 05:34:00 2020 a cipers
15      Wed Jun 24 06:17:00 2020 a cipers
13      Wed Oct 21 12:30:00 2020 a cipers
```

```
cipers@cipers:~$ atrm 11
cipers@cipers:~$ at -l
14      Wed Jun 24 06:16:00 2020 a cipers
15      Wed Jun 24 06:17:00 2020 a cipers
13      Wed Oct 21 12:30:00 2020 a cipers
cipers@cipers:~$ atrm 14
cipers@cipers:~$ at -l
15      Wed Jun 24 06:17:00 2020 a cipers
13      Wed Oct 21 12:30:00 2020 a cipers
cipers@cipers:~$
```

- The /etc/at.deny and /etc/at.allow files allow user to control which users can create jobs with at or batch command. The files consist of a list of usernames, one user name per line.
- By default, only the /etc/at.deny file exists and is empty, which means that all users can use the at command. If user want to deny permission to a specific user, add the username to this file.

=== * ===

Work on Practice Session

Work on Practice Session