

# EXPERIMENT : 1

**AIM:** To learn basic Linux commands

## Commands:

### 1. Directory Related commands

- i. Current Directory: `pwd`

Shows path of current directory.

Options:

- L print the value of \$PWD if it names the current working directory
- P print the physical directory, without any symbolic links

- ii. List subdirectories: `ls`

Shows files and folders inside the current directory.

Options:

Mandatory arguments to long options are mandatory for short options too.

- a, --all do not ignore entries starting with .
- A, --almost-all do not list implied . and ..
- author with -l, print the author of each file
- b, --escape print C-style escapes for nongraphic characters
- block-size=SIZE with -l, scale sizes by SIZE when printing them; e.g., '--block-size=M'; see SIZE format below
- B, --ignore-backups do not list implied entries ending with ~
- c with -lt: sort by, and show, ctime (time of last modification of file status information); with -l: show ctime and sort by name; otherwise: sort by ctime, newest first
- C list entries by columns
- color[=WHEN] colorize the output; WHEN can be 'always' (default if omitted), 'auto', or 'never'; more info below
- d, --directory list directories themselves, not their contents
- D, --dired generate output designed for Emacs' dired mode
- f do not sort, enable -aU, disable -ls --color

-F,	--classify	append indicator (one of */=>@ ) to entries
	--file-type	likewise, except do not append '*'
	--format=WORD	across -x, commas -m, horizontal -x, long -l, single-column -1, verbose -l, vertical -C
	--full-time	like -l --time-style=full-iso
-g		like -l, but do not list owner
	--group-directories-first	group directories before files; can be augmented with a --sort option, but any use of --sort=none (-U) disables grouping
-G,	--no-group	in a long listing, don't print group names
-h,	--human-readable	with -l and -s, print sizes like 1K 234M 2G etc.
	--si	likewise, but use powers of 1000 not 1024
-H,	--dereference-command-line	follow symbolic links listed on the command line
	--dereference-command-line-symlink-to-dir	follow each command line symbolic link that points to a directory
	--hide=PATTERN	do not list implied entries matching shell PATTERN (overridden by -a or -A)
	--hyperlink[=WHEN]	hyperlink file names; WHEN can be 'always' (default if omitted), 'auto', or 'never'
	--indicator-style=WORD	append indicator with style WORD to entry names:
		none (default), slash (-p), file-type (--file-type), classify (-F)
-i,	--inode	print the index number of each file
-l,	--ignore=PATTERN	do not list implied entries matching shell PATTERN
-k,	--kibibytes	default to 1024-byte blocks for disk usage; used only with -s and per directory totals
-l		use a long listing format
-L,	--dereference	when showing file information for a symbolic link, show information for the file the link references rather than for the link itself
-m		fill width with a comma separated list of entries
-n,	--numeric-uid-gid	like -l, but list numeric user and group IDs
-N,	--literal	print entry names without quoting
-o		like -l, but do not list group information
-p,	--indicator-style=slash	append / indicator to directories
-q,	--hide-control-chars	print ? instead of nongraphic characters
	--show-control-chars	show nongraphic characters as-is (the default, unless program is 'ls' and output is a terminal)
-Q,	--quote-name	enclose entry names in double quotes

	--quoting-style=WORD	use quoting style WORD for entry names: literal, locale, shell, shell-always, shell-escape, shell-escape-always, c, escape (overrides QUOTING_STYLE environment variable)
-r,	--reverse	reverse order while sorting
-R,	--recursive	list subdirectories recursively
-s,	--size	print the allocated size of each file, in blocks
-S		sort by file size, largest first
	--sort=WORD	sort by WORD instead of name: none (-U), size (-S), time (-t), version (-v), extension (-X)
	--time=WORD	with -l, show time as WORD instead of default modification time: atime or access or use (-u); ctime or status (-c); also use specified time as sort key if --sort=time (newest first)
	--time-style=TIME_STYLE	time/date format with -l; see TIME_STYLE below
-t		sort by modification time, newest first
-T,	--tabsize=COLS	assume tab stops at each COLS instead of 8
-u		with -lt: sort by, and show, access time; with -l: show access time and sort by name; otherwise: sort by access time, newest first
-U		do not sort; list entries in directory order
-v		natural sort of (version) numbers within text
-w,	--width=COLS	set output width to COLS. 0 means no limit
-x		list entries by lines instead of by columns
-X		sort alphabetically by entry extension
-Z,	--context	print any security context of each file
-1		list one file per line. Avoid '\n' with -q or -b
	--help	display this help and exit
	--version	output version information and exit

### iii. Change Directory: `cd`

Changes current directory.

Usage: `cd [-L][-P [-e]] [-@]] [dir]`

Options:

-L	force symbolic links to be followed: resolve symbolic links in DIR after processing instances of '..'
-P	use the physical directory structure without following

	symbolic links: resolve symbolic links in DIR before processing instances of `..'
-e	if the -P option is supplied, and the current working directory cannot be determined successfully, exit with a non-zero status
-@	on systems that support it, present a file with extended attributes as a directory containing the file attributes

iv. Make Directory: `mkdir`

Usage: `mkdir [OPTION]... DIRECTORY...`

Makes a new folder in current directory

Options:

-m,	--mode=MODE	set file mode (as in <code>chmod</code> ), not <code>a=rwx - umask</code>
-p,	--parents	no error if existing, make parent directories as needed
-v,	--verbose	print a message for each created directory
-Z		set SELinux security context of each created directory to the default type
	--context[=CTX]	like -Z, or if CTX is specified then set the SELinux or SMACK security context to CTX
	--help	display this help and exit
	--version	output version information and exit

v. Remove Directory: `rmdir`

Usage: `rmdir [OPTION]... DIRECTORY...`

Remove the DIRECTORY(ies), if they are empty.

	--ignore-fail-on-non-empty	ignore each failure that is solely because a directory is non-empty
-p,	--parents	remove DIRECTORY and its ancestors; e.g., ' <code>rmdir -p a/b/c</code> ' is similar to ' <code>rmdir a/b/c a/b a</code> '
-v,	--verbose	output a diagnostic for every directory processed
	--help	display this help and exit
	--version	output version information and exit

## 2. File Related Commands:

i. Create a file: `touch`

Usage: `touch [OPTION]... FILE...`

Update the access and modification times of each FILE to the current time.

A FILE argument that does not exist is created empty, unless `-c` or `-h` is supplied.

A FILE argument string of `-` is handled specially and causes `touch` to change the times of the file associated with standard output.

Mandatory arguments to long options are mandatory for short options too.

-a		change only the access time
-c,	--no-create	do not create any files
-d,	--date=STRING	parse STRING and use it instead of current time
-f		(ignored)
-h,	--no-dereference	affect each symbolic link instead of any referenced file (useful only on systems that can change the timestamps of a symlink)
-m		change only the modification time
-r,	--reference=FILE	use this file's times instead of current time
-t STAMP		use [[CC]YY]MMDDhhmm[.ss] instead of current time
	--time=WORD	change the specified time: WORD is access, atime, or use: equivalent to -a WORD is modify or mtime: equivalent to -m
	--help	display this help and exit
	--version	output version information and exit

- ii. Concatenate/Display a file: `cat`  
 Usage: `cat [OPTION]... [FILE]...`  
 Concatenate FILE(s) to standard output.

With no FILE, or when FILE is -, read standard input.

-A,	--show-all	equivalent to -vET
-b,	--number-nonblank	number nonempty output lines, overrides -n
-e		equivalent to -vE
-E,	--show-ends	display \$ at end of each line
-n,	--number	number all output lines
-s,	--squeeze-blank	suppress repeated empty output lines
-t		equivalent to -vT
-T,	--show-tabs	display TAB characters as ^I
-u		(ignored)
-v,	--show-nonprinting	use ^ and M- notation, except for LFD and TAB
	--help	display this help and exit
	--version	output version information and exit

- iii. Copy command: `cp`  
 Usage: `cp [OPTION]... [-T] SOURCE DEST`  
 or: `cp [OPTION]... SOURCE... DIRECTORY`  
 or: `cp [OPTION]... -t DIRECTORY SOURCE...`  
 Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.

-a,	--archive	same as -dR --preserve=all
	--attributes-only	don't copy the file data, just the attributes
	--backup[=CONTROL]	make a backup of each existing destination file
-b		like --backup but does not accept an argument

	--copy-contents	copy contents of special files when recursive
-d		same as --no-dereference --preserve=links
-f,	--force	if an existing destination file cannot be opened, remove it and try again (this option is ignored when the -n option is also used)
-i,	--interactive	prompt before overwrite (overrides a previous -n option)
-H		follow command-line symbolic links in SOURCE
-l,	--link	hard link files instead of copying
-L,	--dereference	always follow symbolic links in SOURCE
-n,	--no-clobber	do not overwrite an existing file (overrides a previous -i option)
-P,	--no-dereference	never follow symbolic links in SOURCE
-p		same as --preserve=mode,ownership,timestamps
	--preserve[=ATTR_LIST]	preserve the specified attributes (default: mode,ownership,timestamps), if possible additional attributes: context, links, xattr, all
	--no-preserve=ATTR_LIST	don't preserve the specified attributes
	--parents	use full source file name under DIRECTORY
-R, -r,	--recursive	copy directories recursively
	--reflink[=WHEN]	control clone/CoW copies. See below
	--remove-destination	remove each existing destination file before attempting to open it (contrast with --force)
	--sparse=WHEN	control creation of sparse files. See below
	--strip-trailing-slashes	remove any trailing slashes from each SOURCE argument
-s,	--symbolic-link	make symbolic links instead of copying
-S,	--suffix=SUFFIX	override the usual backup suffix
-t,	--target-directory=DIRECTORY	copy all SOURCE arguments into DIRECTORY
-T,	--no-target-directory	treat DEST as a normal file
-u,	--update	copy only when the SOURCE file is newer than the destination file or when the destination file is missing
-v,	--verbose	explain what is being done
-x,	--one-file-system	stay on this file system
-Z		set SELinux security context of destination file to default type
	--context[=CTX]	like -Z, or if CTX is specified then set the SELinux or SMACK security context to CTX
	--help	display this help and exit
	--version	output version information and exit

iv. Move command: `mv`

Usage: `mv [OPTION]... [-T] SOURCE DEST`

or: mv [OPTION]... SOURCE... DIRECTORY  
 or: mv [OPTION]... -t DIRECTORY SOURCE...  
 Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.

	--backup[=CONTROL]	make a backup of each existing destination file
-b		like --backup but does not accept an argument
-f,	--force	do not prompt before overwriting
-i,	--interactive	prompt before overwrite
-n,	--no-clobber	do not overwrite an existing file

If you specify more than one of -i, -f, -n, only the final one takes effect.

	--strip-trailing-slashes	remove any trailing slashes from each SOURCE argument
-S,	--suffix=SUFFIX	override the usual backup suffix
-t,	--target-directory=DIRECTORY	move all SOURCE arguments into DIRECTORY
-T,	--no-target-directory	treat DEST as a normal file
-u,	--update	move only when the SOURCE file is newer than the destination file or when the destination file is missing
-v,	--verbose	explain what is being done
-Z,	--context	set SELinux security context of destination file to default type
	--help	display this help and exit
	--version	output version information and exit

v. Remove file: rm

Usage: rm [OPTION]... [FILE]...  
 Remove (unlink) the FILE(s).

-f,	--force	ignore nonexistent files and arguments, never prompt
-i		prompt before every removal
-l		prompt once before removing more than three files, or when removing recursively; less intrusive than -i, while still giving protection against most

mistakes

	--interactive[=WHEN]	prompt according to WHEN: never, once (-l), or always (-i); without WHEN, prompt always
	--one-file-system	when removing a hierarchy recursively, skip any directory that is on a file system different from that of the corresponding command line argument
	--no-preserve-root	do not treat '/' specially
	--preserve-root[=all]	do not remove '/' (default); with 'all', reject any command line argument on a separate device from its parent
-r, -R,	--recursive	remove directories and their contents recursively
-d,	--dir	remove empty directories
-v,	--verbose	explain what is being done

--help	display this help and exit
--version	output version information and exit

vi. Word command: wc

Usage: wc [OPTION]... [FILE]...

or: wc [OPTION]... --files0-from=F

Print newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified. A word is a non-zero-length sequence of characters delimited by white space.

With no FILE, or when FILE is -, read standard input.

The options below may be used to select which counts are printed, always in the following order: newline, word, character, byte, maximum line length.

-c,	--bytes	print the byte counts
-m,	--chars	print the character counts
-l,	--lines	print the newline counts
	--files0-from=F	read input from the files specified by NUL-terminated names in file F; If F is - then read names from standard input
-L,	--max-line-length	print the maximum display width
-w,	--words	print the word counts
	--help	display this help and exit
	--version	output version information and exit

### 3. Misc commands:

i. ECHO:

Repeats statements or returns values of environment variables

ii. set:

Sets shell attributes

iii. whoami:

Print the user name associated with the current effective user ID.

iv. uname

Usage: uname [OPTION]...

Print certain system information. With no OPTION, same as -s.

-a,	--all	print all information, in the following order, except omit -p and -i if unknown:
-s,	--kernel-name	print the kernel name
-n,	--nodename	print the network node hostname
-r,	--kernel-release	print the kernel release
-v,	--kernel-version	print the kernel version
-m,	--machine	print the machine hardware name
-p,	--processor	print the processor type (non-portable)
-i,	--hardware-platform	print the hardware platform (non-portable)
-o,	--operating-system	print the operating system
	--help	display this help and exit
	--version	output version information and exit



v. ps:

Displays system processes

Usage:

ps [options]

Basic options:

-A, -e	all processes
-a	all with tty, except session leaders
a	all with tty, including other users
-d	all except session leaders
-N, --deselect	negate selection
r	only running processes
T	all processes on this terminal
x	processes without controlling ttys

# EXPERIMENT : 2

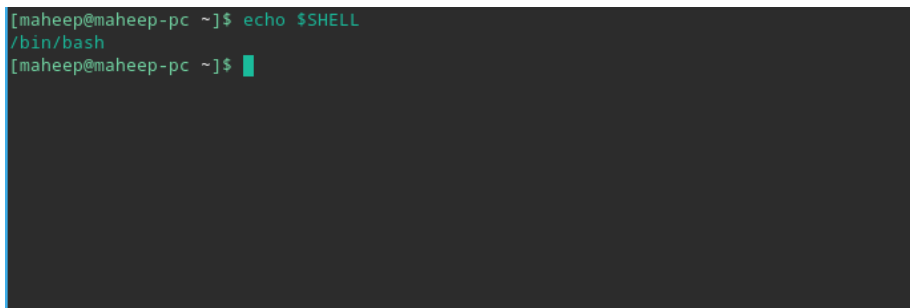
## Objective:

Shell Scripting basics: Making a script that reads the user's name and prints it back.

## Theory:

What is a shell?

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes



```
[maheep@maheep-pc ~]$ echo $SHELL
/bin/bash
[maheep@maheep-pc ~]$
```

programs based on that input. When a program finishes executing, it displays that program's output. Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Default Shell in Linux:

We can find this in the terminal by using `echo $SHELL`

output:

What is a shell script?

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by `#` sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

Shell scripts and functions are both interpreted. This means they are not compiled.

## Procedure:

1. Open terminal.

2. Go to the directory where you want to place the script.
3. Create a new file using any text editor. (here, we use vim)

The file can have any name, and it does not have a limit on extension, though for convenience purposes we give it .sh extension.

4. Write in the following code and save the file.

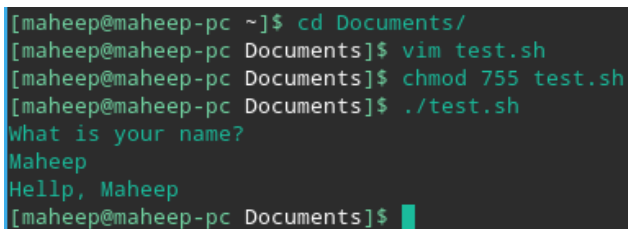
```
#!/bin/bash
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

here PERSON is a local variable. read assigns the user input to the variable.

echo prints any message in the quotes, but returns the values of variables instead of the variable name.

5. Change file permissions by using chmod 755 <filename>
6. Run script by using ./<filename>

### Output:



```
[maheep@maheep-pc ~]$ cd Documents/
[maheep@maheep-pc Documents]$ vim test.sh
[maheep@maheep-pc Documents]$ chmod 755 test.sh
[maheep@maheep-pc Documents]$ ./test.sh
What is your name?
Maheep
Hello, Maheep
[maheep@maheep-pc Documents]$
```

## EXPERIMENT : 3

**Aim :** To study and implement FCFS ( First-come-first-serve ) Scheduling algorithm

**Requiurements :** Windows/Linux based OS, Text editor, GCC compiler

### Theory :

The First Come First Served (FCFS) Scheduling Algorithm is the simplest one. In this algorithm the set of ready processes is managed as FIFO (first-in-first-out) Queue. The processes are serviced by the CPU until completion in order of their entering in the FIFO queue.

A process once allocated the CPU keeps it until releasing the CPU either by terminating or requesting I/O. For example, interrupted process is allowed to continujre running after interrupt handling is done with.

### Related Terminologies :

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time

### Procedure :

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm.

### Code :

```
#include <bits/stdc++.h>

using namespace std;

struct Schedule{
```

```
string pro_id;

int artime,bt,ct,ta,wt;

/*

artime = Arrival time,

bt = Burst time,

ct = Completion time,

ta = Turn around time,

wt = Waiting time

*/;
```

```
bool compare(Schedule p1,Schedule p2) { return p1.artime<p2.artime; }
```

```
int main()
```

```
{
```

```
    Schedule process[100];
```

```
    int cpunon=0;
```

```
    int n,i;
```

```
    cout<<"Enter the number of process: ";
```

```
    cin>>n;
```

```
    cout<<"Enter the (1)Process ID (2) Arrival time and (3) Brust time of "<n<<"
```

```
process\n";
```

```
    for(i=0;i<n;i++)
```

```
    {        cin>>process[i].pro_id;
```

```
            cin>>process[i].artime;
```

```
            cin>>process[i].bt;    }
```

```
sort(process,process+n,compare);

cpunon=process[0].artime-0;

process[0].ct=process[0].artime+process[0].bt;

process[0].ta=process[0].ct-process[0].artime;

process[0].wt=0;

for(i=1;i<n;i++)
{
    if(process[i].artime<=process[i-1].ct)
    {
        process[i].ct=process[i-1].ct+process[i].bt;

        process[i].ta=process[i].ct-process[i].artime;

        process[i].wt=process[i].ta-process[i].bt;
    }
    else
    {
        process[i].ct=process[i].bt+process[i].artime;

        process[i].ta=process[i].ct-process[i].artime;

        process[i].wt=process[i].ta-process[i].bt;

        cpunon+=process[i].artime-process[i-1].ct; }
}

cout<<"\nOUTPUT\n";  cout<<setw(10)<<"Process ID "<<setw(15)<<"Arrival
Time "<<setw(15)<<"Burst Time "<<setw(15)<<"Completion Time"<<setw(15)<<"Waiting
Time "<<setw(15)<<"TA Time \n";
```

```

    for(i=0;i<n;i++)

        {cout<<setw(10)<<process[i].pro_id <<setw(15)<<process[i].artime
        <<setw(15)<<process[i].bt <<setw(15)<<process[i].ct<<setw(15)<<process[i].wt
        <<setw(15)<<process[i].ta<<endl;

        }

    return 0;

}

```

## OUTPUT :

```

Enter the number of process: 4
Enter the (1)Process ID (2) Arrival time and (3) Brust time of 4 process
0 0 4
1 1 7
2 2 6
3 3 5

```

### OUTPUT

Process ID	Arrival Time	Burst Time	Completion Time	Waiting Time	TA Time
0	0	4	4	0	4
1	1	7	11	3	10
2	2	6	17	9	15
3	3	5	22	14	19

# EXPERIMENT : 4

**Aim :** To study and implement SJF ( Shortest Job First ) Scheduling algorithm (Non-preemptive type )

**Requiurements :** Windows/Linux based OS, Text editor, GCC compiler

## Theory :

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.

- Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging.
- It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

## Algorithm:

1. Sort all the process according to the arrival time.
2. Then select that process which has minimum arrival time and minimum Burst time.
3. After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

## How to compute below times in SJF using a program?

1. Completion Time: Time at which process completes its execution.



2. Turn Around Time: Time Difference between completion time and arrival time.  
 $\text{Turn Around Time} = \text{Completion Time} - \text{Arrival Time}$
3. Waiting Time(W.T): Time Difference between turn around time and burst time.  
 $\text{Waiting Time} = \text{Turn Around Time} - \text{Burst Time}$

**Procedure :**

```
#include <bits/stdc++.h>

using namespace std;

struct Schedule{

    string pro_id;

    int artime,bt,ct,ta,wt; };

bool compare(Schedule p1,Schedule p2) { return p1.bt<p2.bt; }

int main()

{

    Schedule process[100];

    int n,i;

    cout<<"Enter the number of process: ";

    cin>>n;

    cout<<"Enter the (1)Process ID (2) Arrival time (msec) and (3) Brust time (msec) of "<n<<" process\n";

    for(i=0;i<n;i++)

    {

        cin>>process[i].pro_id;

        cin>>process[i].artime;

        cin>>process[i].bt;

    }
```

```
sort(process,process+n,compare);

cpunon=process[0].artime-0;

process[0].ct=process[0].artime+process[0].bt;

process[0].ta=process[0].ct-process[0].artime;

process[0].wt=0;

for(i=1;i<n;i++)
{
    if(process[i].artime<=process[i-1].ct)
    {process[i].ct=process[i-1].ct+process[i].bt;

        process[i].ta=process[i].ct-process[i].artime;

        process[i].wt=process[i].ta-process[i].bt;

    }
    else
    {
        process[i].ct=process[i].bt+process[i].artime;

        process[i].ta=process[i].ct-process[i].artime;

        process[i].wt=process[i].ta-process[i].bt;

        cpunon+=process[i].artime-process[i-1].ct;

    }
}

cout<<"\nOUTPUT\n";

cout<<setw(10)<<"Process ID " <<setw(17)<<"Arrival Time(msec)
"<<setw(17)<<"Burst Time(msec) " <<setw(17)<<"Completion Time(msec)
"<<setw(17)<<"Waiting Time(msec) " <<setw(17)<<"TA Time(msec)\n";
```

```

    for(i=0;i<n;i++)

        {cout<<setw(10)<<process[i].pro_id <<setw(17)<<process[i].artime
<<setw(17)<<process[i].bt <<setw(17)<<process[i].ct<<setw(17)<<process[i].wt
<<setw(17)<<process[i].ta<<endl;

        }

    return 0;

}

```

## OUTPUT :

```

Enter the number of process: 4
Enter the (1)Process ID (2) Arrival time (msec) and (3) Brust time (msec) of 4 process
0 0 4
1 1 7
2 2 6
3 3 5

OUTPUT
Process ID Arrival Time(msec) Burst Time(msec) Completion Time(msec) Waiting Time(msec) TA Time(msec)
    0          0          4          4          0          4
    3          3          5          9          1          6
    2          2          6         15          7         13
    1          1          7         22         14         21

```

# EXPERIMENT : 5

**Aim :** To study and implement Round-Robin Scheduling algorithm (Non- preemptive type )

**Requiurements :** Windows/Linux based OS, Text editor, GCC compiler

## Theory :

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.

## Steps to find waiting times of all processes:

- 1- Create an array **rem\_bt[]** to keep track of remaining burst time of processes. This array is initially a copy of **bt[]** (burst times array)
- 2- Create another array **wt[]** to store waiting times of processes. Initialize this array as 0.
- 3- Initialize time :  $t = 0$
- 4- Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet.
  - a- If  $\text{rem\_bt}[i] > \text{quantum}$ 
    - (i)  $t = t + \text{quantum}$

```
(ii) bt_rem[i] -= quantum;  
c- Else // Last cycle for this process  
  (i) t = t + bt_rem[i];  
  (ii) wt[i] = t - bt[i]  
  (ii) bt_rem[i] = 0; // This process is over
```

### Procedure :

```
#include <bits/stdc++.h>  
  
#include <iomanip>  
  
using namespace std;  
  
struct process{  
    int id;  
  
    int arrivalTime,burstTime,remaining,completionTime;  
  
    int taTime,waitTime;  
};  
  
bool comp(process a,process b){  
    if(a.arrivalTime==b.arrivalTime)  
        return a.id<b.id;  
  
    return a.arrivalTime<=b.arrivalTime;  
}  
  
static int timeQuantum=2;  
  
int main(int argc, char const *argv[])
```

```
{  
    int n;  
  
    cout<<"\nEnter number of processes :";  
  
    cin>>n;  
  
    process p[n];  
  
    cout<<"\nEnter the arrival time and burst time for processes :";  
  
    for(int i=0;i<n;i++){  
        p[i].id=i+1;  
  
        cout<<"\nProcess "<<i+1<<" : ";  
  
        cin>>p[i].arrivalTime>>p[i].burstTime;  
  
        p[i].remaining=p[i].burstTime;  
    }  
  
    sort(p,p+n,comp);  
  
    int time=0;  
  
    list<process* >done;  
  
    deque<process* > q;  
  
    int gchart[100]={1};  
  
    int gcount=1;  
  
    while(true){  
        if(time==p[0].arrivalTime){  
            process* prun=p;  
  
            q.push_back(prun);  
        }  
    }  
}
```

```
        break;
    }
    else
        time++;
}

int i=1;
while(!q.empty()){
    process *ready=q.front();
    q.pop_front();
    gchart[gcount++]=ready->id;

    if(ready->remaining>=timeQuantum){
        ready->remaining-=2;
        time+=2;
    }
    else if(ready->remaining<timeQuantum){
        time+=ready->remaining;
        ready->remaining=0;
    }

    while(time >= p[i].arrivalTime){
        process *prun=p+i;
        q.push_back(prun);
```

```

        i++;
    }

    if(ready->remaining==0){
        ready->completionTime=time;
        ready->taTime=(ready->completionTime)-(ready->arrivalTime);
        ready->waitTime=(ready->taTime)-(ready->burstTime);
    }

    else

        q.push_back(ready);

}

cout<<"\n\nGaint chart:\n";

int x=0;

while((x++)<gcount-1)

    cout<<"P"<<gchart[x]<<" -> ";

cout<<"\nOUTPUT\n";

cout<<setw(10)<<"Process ID "<<setw(17)<<"Arrival Time(msec) "<<setw(17)<<"Burst
Time(msec) "<<setw(17)<<"Completion Time(msec) "<<setw(17)<<"Waiting Time(msec)
"<<setw(17)<<"TA Time(msec)\n";

for(int i=0;i<n;i++)

    { cout<<setw(10)<<p[i].id <<setw(17)<<p[i].arrivalTime
<<setw(17)<<p[i].burstTime <<setw(17)<<p[i].completionTime<<setw(17)<<p[i].waitTime
<<setw(17)<<p[i].taTime<<endl;

```



```

    }

return 0;

}

```

## OUTPUT :

```

Enter number of processes :5

Enter the arrival time and burst time for processes :
Process 1 : 0 5

Process 2 : 1 3

Process 3 : 2 1

Process 4 : 3 2

Process 5 : 4 3

Gaint chart:
P1 -> P2 -> P3 -> P1 -> P4 -> P5 -> P2 -> P1 -> P5 ->

```

Process ID	Arrival Time(msec)	Burst Time(msec)	Completion Time(msec)	Waiting Time(msec)	TA Time(msec)
1	0	5	13	8	13
2	1	3	12	8	11
3	2	1	5	2	3
4	3	2	9	4	6
5	4	3	14	7	10

# EXPERIMENT : 6

**Aim :** To study and implement SRTF ( Shortest Remaning Time First) Scheduling algorithm (Non- preemptive type )

**Requiurements :** Windows/Linux based OS, Text editor, GCC compiler

## Theory :

Shortest Remaining Time First (SRTF) is the preemptive version of Shortest Job Next (SJN) algorithm, where the processor is allocated to the job closest to completion. This algorithm requires advanced concept and knowledge of CPU time required to process the job in an interactive system, and hence can't be implemented there. But, in a batch system where it is desirable to give preference to short jobs, SRT algorithm is used.

However, SRT involves more overheads than SJN, as the OS is required to frequently monitor the CPU time of the jobs in the READY queue and perform context switching. for the same set of jobs, SRT algorithm is faster in execution than SJN algorithm. But, here the overhead charges, i.e., time required for context switching has been ignored. When a job is preempted, all of it's processing information must be saved in it's PCB for later when it is to be continued, and the contents of the PCB of the other job to which the OS is switching are loaded into the registers in the memory. This is known as **Context Switching**.

## Advantages:

SRTF algorithm makes the processing of the jobs faster than SJN algorithm, given it's overhead charges are not counted.

## Disadvantages:

The context switch is done a lot more times in SRTF than in SJN, and consumes CPU's valuable time for processing. This adds up to it's processing time and diminishes it's advantage of fast processing.

## Procedure :

```
#include <iostream>
using namespace std;

int main() {
    int n,i,j,temp=0,tat=0;
    int arrTime[10],execTime[10],nexecTime[10],pro[10],npro[10];
    cout<<"Enter number of processes:"<<endl;
    cin>>n;
    for(i=0;i<n;i++) {
        pro[i]=i;
        npro[i]=i;
    }

    for(i=0;i<n;i++) {
        cout<<"Enter execution time of process "<<i<<":";
        cin>>execTime[i];
        cout<<"Enter arrival time of process "<<i<<":";
        cin>>arrTime[i];
        cout<<endl;
    }

    for(i=0;i<n;i++) {
        tat = tat+execTime[i];
    }

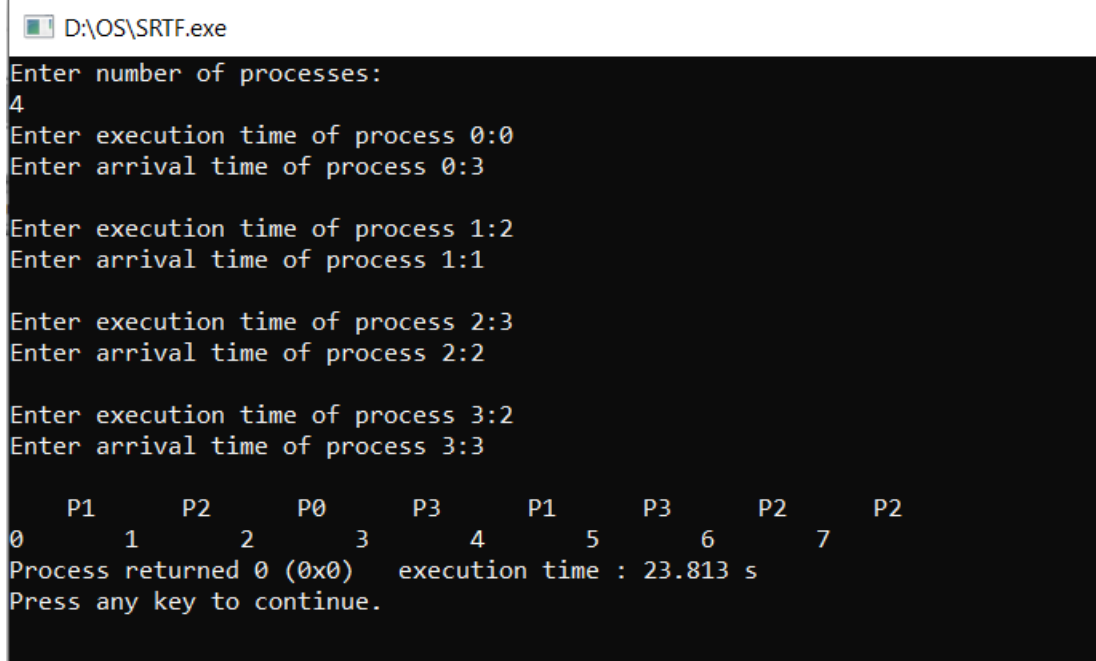
    for(i=0;i<n-1;i++) {
        for(j=0;j<n-i-1;j++) {
            if(arrTime[j]>arrTime[j+1]) {
                temp=arrTime[j+1];
                arrTime[j+1]=arrTime[j];
                arrTime[j]=temp;

                temp=pro[j+1];
                pro[j+1]=pro[j];
                pro[j]=temp;
            }
        }
    }

    for(i=0;i<n;i++) {
        cout<<"  P"<<pro[i]<<"  ";
```

```
        execTime[pro[i]]=execTime[pro[i]]-1;
    }
    for(i=0;i<n-1;i++) {
        for(j=0;j<n-i-1;j++) {
            if(execTime[j]>execTime[j+1]) {
                temp=execTime[j+1];
                execTime[j+1]=execTime[j];
                execTime[j]=temp;

                temp=npro[j+1];
                npro[j+1]=npro[j];
                npro[j]=temp;
            }
        }
    }
    for(i=0;i<n;i++) {
        if(execTime[i]!=0) {
            for(j=0;j<execTime[i];j++) {
                cout<<"  P"<<npro[i]<<" ";
            }
        }
    }
    cout<<endl;
    for(i=0;i<tat+1;i++) {
        cout<<i<<"\t";
    }
    return 0;
}
```

**OUTPUT :**

```
D:\OS\SRTF.exe
Enter number of processes:
4
Enter execution time of process 0:0
Enter arrival time of process 0:3

Enter execution time of process 1:2
Enter arrival time of process 1:1

Enter execution time of process 2:3
Enter arrival time of process 2:2

Enter execution time of process 3:2
Enter arrival time of process 3:3

      P1      P2      P0      P3      P1      P3      P2      P2
0      1      2      3      4      5      6      7
Process returned 0 (0x0)   execution time : 23.813 s
Press any key to continue.
```

# EXPERIMENT : 7

**Aim :** To implement and understand the solutions of producer consumer and reader writer problems using semaphore.

**Requirements :** Linux based OS, Text editor, C compiler

## (a) PRODUCER AND CONSUMER PROBLEM

### Theory :

**Problem Statement** – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

Initialization of semaphores –

mutex = 1

Full = 0 // Initially, all slots are empty. Thus full slots are 0

Empty = n // All slots are empty initially

### Solution for Producer –

```
do{
//produce an item
wait(empty);
wait(mutex);
//place in buffer
signal(mutex);
signal(full);
}while(true)
```

**Solution for Consumer –**

```
do{
wait(full);
wait(mutex);
// remove item from buffer
signal(mutex);
signal(empty);
// consumes item
}while(true)
```

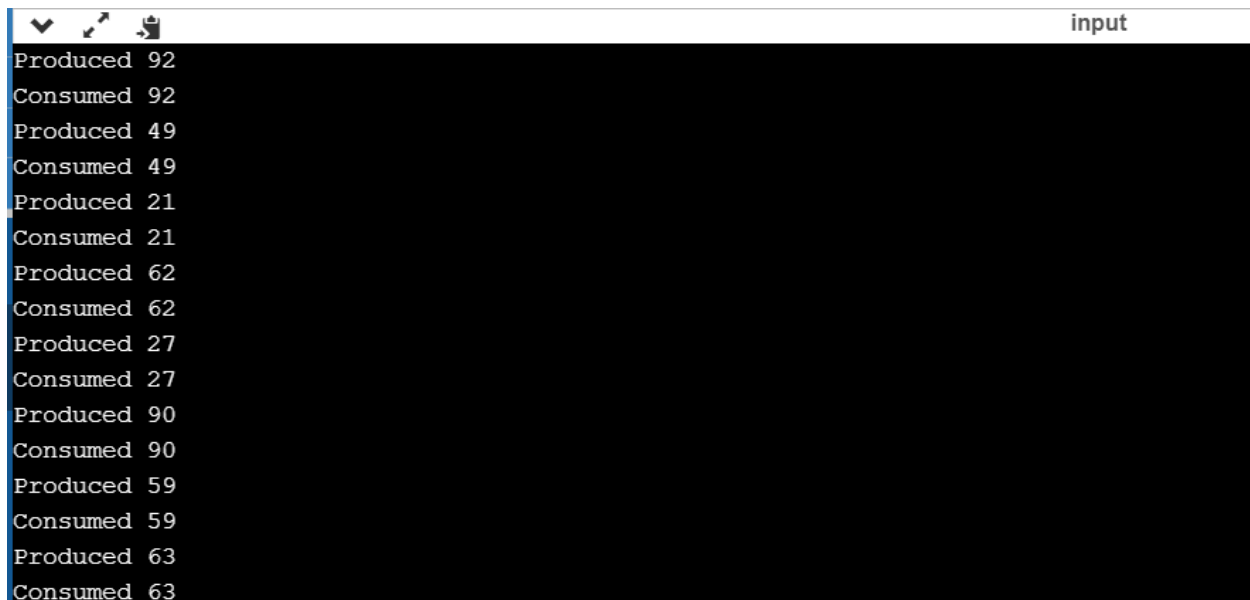
**Procedure :****C++ code Solution (only executable on linux based systems)**

```
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <random>
#include <unistd.h>
using namespace std;
#define BUFFER_SIZE 10
int buffer[BUFFER_SIZE];
int index=0;
sem_t full,empty;
pthread_mutex_t mutex;
void* produce(void* arg){
    while(1){
        sleep(1);
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        int item = rand()%100;
        buffer[index++] = item;
        cout<<"Produced "<<item<<endl;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}
void* consume(void* arg){
    while(1){
        sleep(1);
```

```
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[--index];
        cout<<"Consumed "<<item<<endl;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
}


int main(){
    pthread_t producer,consumer;
    sem_init(&empty,0,BUFFER_SIZE);
    sem_init(&full,0,0);
    pthread_mutex_init(&mutex,NULL);
    pthread_create(&producer,NULL,produce,NULL);
    pthread_create(&consumer,NULL,consume,NULL);
    pthread_exit(NULL);
}
```

## OUTPUT:



```
Produced 92
Consumed 92
Produced 49
Consumed 49
Produced 21
Consumed 21
Produced 62
Consumed 62
Produced 27
Consumed 27
Produced 90
Consumed 90
Produced 59
Consumed 59
Produced 63
Consumed 63
```





```
Produced 88
Consumed 88
Produced 74
Consumed 74
Produced 33
Consumed 33
Produced 57
Consumed 57
Produced 81
Consumed 81

...Program finished with exit code 9
Press ENTER to exit console.█
```

**(B )READER AND WRITER PROBLEM*****Problem parameters:***

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

***Solution when Reader has the Priority over Writer:***

(priority means, no reader should wait if the share is currently opened for reading.)

Three variables are used: mutex, wrt, readcnt to implement solution

semaphore mutex, wrt;

semaphore mutex is used to ensure mutual exclusion when readcnt is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers

int readcnt;

readcnt tells the number of processes performing read in the critical section, initially 0

**Functions for sempahore :**

- wait() : decrements the semaphore value.
- signal() : increments the semaphore value.

**Writer process:**

Writer requests the entry to critical section.

If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.

It exits the critical section.

```
do {  
    // writer requests for critical section  
    wait(wrt);  
  
    // performs the write  
  
    // leaves the critical section  
    signal(wrt);
```

```
} while(true);
```

**Reader process:**

Reader requests the entry to critical section.

If allowed:

it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.

It then, signals mutex as any other reader is allowed to enter while others are already reading.

After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.

If not allowed, it keeps on waiting.

do {

    // Reader wants to enter the critical section

    wait(mutex);

    // The number of readers has now increased by 1

    readcnt++;

    // there is atleast one reader in the critical section

    // this ensure no writer can enter if there is even one reader

    // thus we give preference to readers here

    if (readcnt==1)

        wait(wrt);

    // other readers can enter while this current reader is inside

    // the critical section

    signal(mutex);

    // current reader performs reading here

    wait(mutex); // a reader wants to leave

    readcnt--;

    // that is, no reader is left in the critical section,

    if (readcnt == 0)

        signal(wrt); // writers can enter

    signal(mutex); // reader leaves

} while(true);

