



Ramaiah Institute of Technology

(Autonomous Institute, Affiliated to VTU)

**Department of Artificial Intelligence and Machine
Learning**

Department of Artificial Intelligence and Data Science

OPERATING SYSTEM LAB MANUAL

Sub code: AI52/AD52

Oct 2024 -Jan 2025

INDEX

SL. NO	PROGRAM	PAGE NO.
i	Basic Unix Commands	
1	Simple Shell Programs a. even or odd b. leap year c. factorial d. swap two integers e. split and join a string	
2	Program for system calls of Unix operating systems (opendir, readdir, closedir)	
3	Program for system calls of Unix operating system (fork, getpid, exit)	
4	CPU Scheduling Algorithms - FCFS	
5	CPU Scheduling Algorithms - SJF	
6	CPU Scheduling Algorithms - SRTF	
7	CPU Scheduling Algorithms - PRIORITY	
8	CPU Scheduling Algorithms – ROUND ROBIN	
9	Producer Consumer Problem Using Semaphores	
10	IPC using Shared Memory	
11	Bankers Algorithm For Deadlock Avoidance	
12	Memory Allocation Methods For Fixed Partition – First Fit	
13	Memory Allocation Methods For Fixed Partition – Best Fit	

i	BASICS OF UNIX COMMANDS
	INTRODUCTION TO BASIC UNIX COMMANDS

AIM:

To study about the basics UNIX commands

UNIX:

It is a multi-user operating system. Developed at AT & T Bell Industries, USA in 1969.

Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed Information and Computing Service) OS.

By 1980, UNIX had been completely rewritten using C language.

LINUX:

It is similar to UNIX, which was created by Linus Torvalds. All UNIX commands work in Linux. Linux is an open source software. The main feature of Linux is coexisting with other OS such as windows and UNIX.

STRUCTURE OF A LINUX SYSTEM:

It consists of three parts.

- a) UNIX kernel
- b) Shells
- c) Tools and Applications

UNIX KERNEL:

Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS.

Decides when one program stops and another starts.

SHELL:

Shell is the command interpreter in the UNIX OS. It accepts command from the user and analyses and interprets them

a) date

–used to check the date and time

Syn:\$date

Format	Purpose	Example	Result
+%m	To display only month	\$date+%m	06
+%h	To display month name	\$date+%h	June
+%d	To display day of month	\$date+%d	01
+%y	To display last two digits of years	\$date+%y	09
+%H	To display hours	\$date+%H	10
+%M	To display minutes	\$date+%M	45
+%S	To display seconds	\$date+%S	55

b) cal

–used to display the calendar

Syn:\$cal 2 2009

c) echo

–used to print the message on the screen.

Syn:\$echo “text”

d) ls

–used to list the files. Your files are kept in a directory.

Syn:\$ls-ls

All files (include files with prefix)

ls-l Lodetai (provide file statistics)

ls-t Order by creation time

ls- u Sort by access time (or show when last accessed together with -l)

ls-s Order by size

ls-r Reverse order

ls-f Mark directories with /,executable with * , symbolic links with @, local sockets with =,
named pipes(FIFOs)with

ls-s Show file size

ls- h“ Human Readable”, show file size in Kilo Bytes & Mega Bytes (h can be used together with -l or)

ls[a-m]*List all the files whose name begin with alphabets From „a“ to „m“

ls[a]*List all the files whose name begins with „a“ or „A“

Eg:\$ls>my list Output of „ls“ command is stored to disk file named „my list“

e) lp

–used to take printouts

Syn:\$lp filename

f) man

–used to provide manual help on every UNIX commands.

Syn:\$man unix command

\$man cat

g) **who & whoami**

–it displays data about all users who have logged into the system currently. The next command displays about current user only.

Syn:\$who\$whoami

h) **uptime**

–tells you how long the computer has been running since its last reboot or power-off.

Syn:\$uptime

i) **uname**

–it displays the system information such as hardware platform, system name and processor, OS type.

Syn:\$uname-a

j) **hostname**

–displays and set system host name

Syn:\$ hostname

k) **bc**

–stands for „best calculator“

\$bc	\$ bc	\$ bc	\$ bc
10/2*3	scale =1	ibase=2	sqrt(196)
15	2.25+1	obase=16	14 quit
	3.35	11010011	
	quit	89275	
		1010	
		Ā	
		Quit	
\$bc	\$ bc-l		
for(i=1;i<3;i=i+1)I	scale=2		
1	s(3.14)		
2	0		
3 quit			

FILE MANIPULATION COMMANDS

a) **cat**—this create, view and concatenate files.

Creation:

Syn:\$cat>filename

Viewing:

Syn:\$cat filename

Add text to an existing file:

Syn:\$cat>>filename

Concatenate:

Syn:\$catfile1file2>file3

\$catfile1file2>>file3 (no over writing of file3)

b) **grep**—used to search a particular word or pattern related to that word from the file. Syn:\$grep search word filename
Eg:\$grep anu student

c) **rm**—deletes a file from the file system Syn:\$rm filename

d) **touch**—used to create a blank file.
Syn:\$touch file names

e) **cp**—copies the files or directories Syn:\$cpsource file destination file
Eg:\$cp student stud

f) **mv**—to rename the file or directory syn:\$mv old file new file
Eg:\$mv-i student student list(-i prompt when overwrite)

g) **cut**—it cuts or pickup a given number of character or fields of the file. Syn:\$cut<option><filename>
Eg: \$cut -c filename
\$cut-c1-10emp
\$cut-f 3,6emp
\$ cut -f 3-6 emp
-c cutting columns
-f cutting fields

h) **head**—displays10 lines from the head(top)of a given file Syn:\$head filename
Eg:\$head student

To display the top two lines:

Syn:\$head-2student

i) **tail**—displays last 10 lines of
the file Syn:\$tail filename

Eg:\$tail student

To display the bottom two lines;

Syn:\$ tail -2 student

j) **chmod**—used to change the permissions of a file or
directory. Syn:\$ch mod category operation
permission file Where, Category—is the user type

Operation—is used to assign or remove permission

Permission—is the type of permission

File—are used to assign or remove permission all

Examples:

\$chmodu-wx student

Removes write and execute permission for users

\$ch modu+rw,g+rwwstudent

Assigns read and write permission for users and groups

\$chmodg=rwx student

Assigns absolute permission for groups of all read, write and execute permissions

k) **wc**—it counts the number of lines, words, character in a specified
file(s) with the options as -l,-w,-c

Category	Operation	Permission
u— users	+assign	r— read
g—group	-remove	w— write
o— others	=assign absolutely	x-execut e

Syn: \$wc -l filename

\$wc -w filename

\$wc -c filename

```

Lab703@ThinkCentre-M70t:~$ date +%h
Jan
Lab703@ThinkCentre-M70t:~$ date +%d
02
Lab703@ThinkCentre-M70t:~$ date +%y
25
Lab703@ThinkCentre-M70t:~$ date +%H
14
Lab703@ThinkCentre-M70t:~$ date +%M
25
Lab703@ThinkCentre-M70t:~$ date +%S
19
Lab703@ThinkCentre-M70t:~$ cal
      January 2025
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

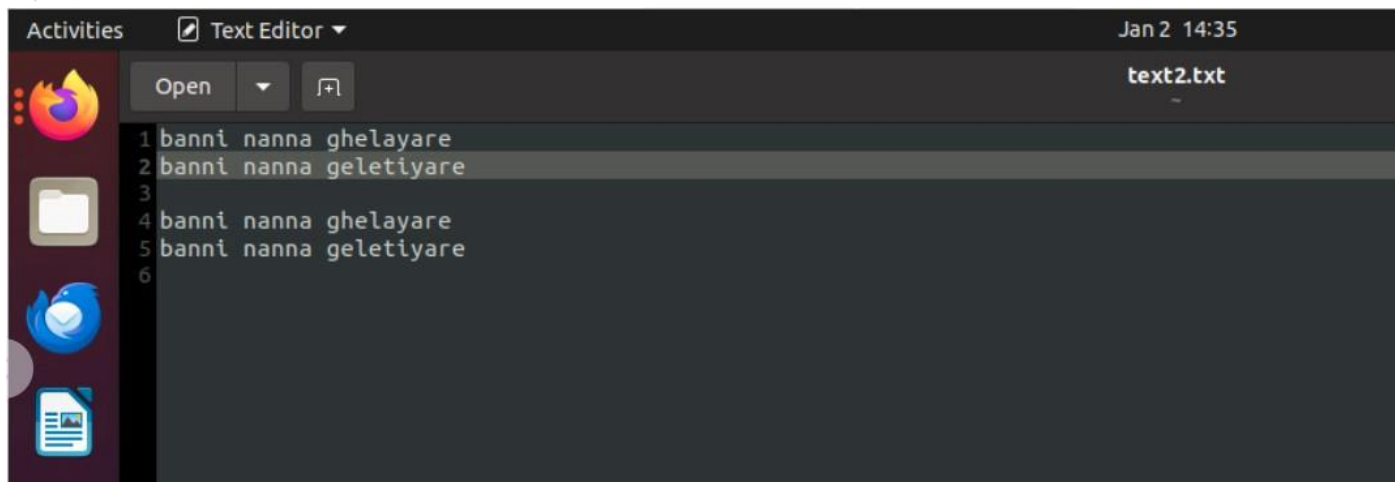
Lab703@ThinkCentre-M70t:~$ echo "hi"
hi
Lab703@ThinkCentre-M70t:~$ who
Lab703      :0                2025-01-02 14:08 (:0)
Lab703@ThinkCentre-M70t:~$ whoami
Lab703
Lab703@ThinkCentre-M70t:~$ uptime
 14:25:54 up 19 min,  1 user,  load average: 0.45, 0.29, 0.20
Lab703@ThinkCentre-M70t:~$ uname
Linux
Lab703@ThinkCentre-M70t:~$ hostname
ThinkCentre-M70t
Lab703@ThinkCentre-M70t:~$

```

```

Lab703@ThinkCentre-M70t:~$ gedit untitled.txt
Lab703@ThinkCentre-M70t:~$ gedit untitled.txt
Lab703@ThinkCentre-M70t:~$ cat untitled.txt
namsakara gelayare
ellaru hegi idira

```

```
Lab703@ThinkCentre-M70t:~$ grep banni text2.txt
banni nanna ghelayare
banni nanna geletiyare
banni nanna ghelayare
banni nanna geletiyare
Lab703@ThinkCentre-M70t:~$
```

```
Lab703@ThinkCentre-M70t:~$ head text2.txt
am going back to 505,
if its a 7 hours flight
or a 45 minutes drive.
in my imagination your
lying by your side
```

```
Lab703@ThinkCentre-M70t:~$ tail text2.txt
am going back to 505,
if its a 7 hours flight
or a 45 minutes drive.
in my imagination your
lying by your side
```

```
Lab703@ThinkCentre-M70t:~$ wc -w text2.txt
24 text2.txt
```

AIM:

To write simple shell programs by using conditional, branching and looping statements.

1.

a. Write a Shell program to check whether the given number is even or odd

ALGORITHM:

SEPT 1: Start the program.

STEP 2: Read the value of n.

STEP 3: Calculate „r=expr \$n%2“.

STEP 4: If the value of r equals 0 then print the number is even

STEP 5: If the value of r not equal to 0 then print the number is odd.

PROGRAM:

create file - gedit oddeven.sh

change the permissions chmod + oddeven.sh

execute or run - ./oddeven.sh

```
echo "Enter the Number"
```

```
read n
```

```
r=`expr $n % 2`
```

```
if [ $r -eq 0 ]
```

```
then
```

```
echo "$n is Even number"
```

```
else
```

```
echo "$n is Odd number"
```

```
fi
```

```
Lab703@ThinkCentre-M70t:~$ chmod + oddeven.sh
Lab703@ThinkCentre-M70t:~$ ./oddeven.sh
Enter Number
78
78 is even
```

1.

b. Write a Shell program to check whether the given year is a leap year or not

ALGORITHM:

SEPT 1: Start the program.

STEP 2: Read the value of year.

STEP 3: Calculate „b=expr \$y%4“.

STEP 4: If the value of b equals 0 then print the year is a leap year

STEP 5: If the value of r not equal to 0 then print the year is not a leap year

PROGRAM:

```
echo "Enter the year"
read y
b=`expr $y % 4`
if [ $b -eq 0 ]
then
echo "$y is a leap year"
else
echo "$y is not a leap year"
fi
```

```
Lab703@ThinkCentre-M70t:~$ ./program4.sh
Enter the year
2025
2025 is not a leap year
```

1. c. Write a Shell program to find the factorial of a number

ALGORITHM:

SEPT 1: Start the program.

STEP 2: Read the value of n.

STEP 3: Calculate „i=expr \$n-1“.

STEP 4: If the value of i is greater than 1 then calculate „n=expr \$n * \$i“ and „i=expr \$i - 1“

STEP 5: Print the factorial of the given number.

PROGRAM:

```
echo "Enter a Number"
read n
i=`expr $n - 1`
p=1
while [ $i -ge 1 ]
do
n=`expr $n \* $i`
i=`expr $i - 1`
done
echo "The Factorial of the given Number is $n"
```

```
Lab703@ThinkCentre-M70t:~$ ./program5.sh
Enter a Number
5
The Factorial of the given Number is 120
```

1. d. Write a Shell program to swap the two integers

ALGORITHM:

STEP 1: Start the program.


STEP 2: Read the value of a,b.

STEP 3: Calculate the swapping of two values by using a temporary variable temp.

STEP 4: Print the value of a and b.

PROGRAM:

```
echo "Enter Two Numbers"
read a b
temp=$a
a=$b
b=$temp
echo "after swapping"
echo $a $b
```



```
Lab703@ThinkCentre-M70t:~$ ./program6.sh
Enter Two Numbers
78 23
after swapping
23 78
```

1. e. Write a Shell program to split a string and join the string

ALGORITHM:

Start the program.

Declare a character array to hold the input string and an array to hold the split words.

Read the input string from the user.

Use strtok to split the string into tokens (words) based on spaces.

Print each split word.

Join the split words into a single string with a specified separator (e.g., a space or a hyphen).

Print the joined string.

End the program.

PROGRAM:

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main() {
    char input[256]; // To hold the input string
    char result[256] = ""; // To hold the joined string
    char *word; // Pointer for tokens
    char separator = '-'; // Separator for joining
```

```

// Step 3: Read the input string
printf("Enter a string: ");
fgets(input, sizeof(input), stdin);
input[strcspn(input, "\n")] = '\0'; // Remove the newline character

// Step 4: Split the string into words
printf("Split words:\n");
word = strtok(input, " ");
while (word != NULL) {
    printf("%s\n", word);

    // Step 6: Join the split words
    if (result[0] != '\0') {
        strncat(result, &separator, 1); // Add separator
    }
    strcat(result, word);

    word = strtok(NULL, " ");
}

// Step 7: Print the joined string
printf("Joined string: %s\n", result);

return 0;
}

```

2 & 3

Programs using the following system calls of UNIX operating system fork, exec, getpid, exit, wait, close, stat, opendir, readdir

AIM:

To write C Programs using the following system calls of UNIX operating system fork, exec, getpid, exit, wait, close, stat, opendir, readdir.

2. PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (OPENDIR, READDIR, CLOSEDIR)

ALGORITHM:

STEP 1: Start the program.
STEP 2: Create struct dirent.
STEP 3: declare the variable buff and pointer dptr.
STEP 4: Get the directory name.
STEP 5: Open the directory.
STEP 6: Read the contents in directory and print it.
STEP 7: Close the directory.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h> // For directory operations

int main() {
    char dirname[256];
    struct dirent *dptr; // Struct to hold directory entry information
    DIR *dir;           // Pointer to directory

    // Step 4: Get the directory name
    printf("ENTER DIRECTORY NAME: ");
    scanf("%s", dirname);

    // Step 5: Open the directory
    dir = opendir(dirname);
    if (dir == NULL) {
        perror("Error opening directory");
        return 1; // Exit with error
    }

    // Step 6: Read the contents of the directory and print them
    while ((dptr = readdir(dir)) != NULL) {
        printf("%s\n", dptr->d_name); // Print the name of each entry
    }

    // Step 7: Close the directory
    closedir(dir);

    return 0; // Successful execution
}
```

Compile and RUN:

```
gcc list_dir.c -o list_dir
./list_dir
```

```
Lab703@ThinkCentre-M70t:~$ ./a.out

ENTER DIRECTORY NAME: ai002
..
hello.java
```

3. PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEM (fork, getpid, exit)

ALGORITHM:

- STEP 1: Start the program.
- STEP 2: Declare the variables pid,pid1,pid2.
- STEP 3: Call fork() system call to create process.
- STEP 4: If pid==-1, exit.
- STEP 5: Ifpid!=-1 , get the process id using getpid().
- STEP 6: Print the process id.
- STEP 7: Stop the program

PROGRAM:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // Required for the exit() function

int main() {
    int pid, pid1, pid2;

    pid = fork(); // Create a new process
    if (pid == -1) {
        // If fork fails
        printf("ERROR IN PROCESS CREATION\n");
        exit(1); // Exit the program if process creation fails
    }

    if (pid != 0) {
        // Parent process
        pid1 = getpid(); // Get the process ID of the parent printf("\nThe parent
        process ID is %d\n", pid1);
    }
}
```

```
} else {  
    // Child process  
    pid2 = getpid(); // Get the process ID of the child printf("\nThe child  
    process ID is %d\n", pid2);  
}  
  
return 0; } // End of program
```

```
Lab703@ThinkCentre-M70t:~$ gedit program2.c  
Lab703@ThinkCentre-M70t:~$ gcc program2.c  
Lab703@ThinkCentre-M70t:~$ ./a.out
```

```
The parent process ID is 6924
```

```
The child process ID is 6925
```


4	CPU SCHEDULING ALGORITHMS
	First Come First Serve(FCFS)

AIM:

To write a C program for implementation of First Come First Serve(FCFS) scheduling algorithms

ALGORITHM:

Step 1 : Input Number of Processes

Step 2: Input the processes burst time (bt).

Step 3: Find waiting time (wt) for all processes.

- As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. $wt[0] = 0$.
- Find waiting time for all other processes i.e. for process $i \rightarrow$
 $wt[i] = bt[i-1] + wt[i-1]$.

Step 4: Find turnaround time = waiting_time + burst_time for all processes.

Step 5: Find average waiting time =
 $total_waiting_time / no_of_processes$.

Step 6: Similarly, find average turnaround time =
 $total_turn_around_time / no_of_processes$.

PROGRAM:

```
#include <stdio.h>
```

```
int main() {
    int pid[15]; // Process IDs
    int bt[15]; // Burst times
    int n;

    // Get the number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    // Get process IDs
    printf("Enter process ID of all the processes: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pid[i]);
    }
}
```

```

// Get burst times
printf("Enter burst time of all the processes: "); for (int i = 0; i < n; i++) {
    scanf("%d", &bt[i]);
}

int wt[n];          // Waiting times
wt[0] = 0;          // Waiting time for the first process is 0

// Calculate waiting time for each process for (int i = 1; i < n; i++) {
    wt[i] = bt[i - 1] + wt[i - 1];
}

// Print the process table
printf("Process ID   Burst Time   Waiting Time   Turnaround Time\n");

float twt = 0.0;      // Total waiting time float tat = 0.0;
                      // Total turnaround time

for (int i = 0; i < n; i++) {
    printf("%d\t", pid[i]);                // Print process ID
    printf("%d\t", bt[i]);                 // Print burst time
    printf("%d\t", wt[i]);                 // Print waiting time

    // Calculate and print turnaround time printf("%d\t", bt[i]
    + wt[i]); printf("\n");

    // Calculate total waiting time twt += wt[i];

    // Calculate total turnaround time tat += (wt[i] +
    bt[i]);
}

// Calculate and print average waiting time and turnaround time
float awt = twt / n;      // Average waiting time
float att = tat / n;      // Average turnaround time

printf("Avg. waiting time = %.2f\n", awt); printf("Avg. turnaround time = %.2f", att);

return 0;
}

```

```
Enter the number of processes: 3
Enter process id of all the processes: 1
2
3
Enter burst time of all the processes: 10
5
8
Process ID      Burst Time      Waiting Time      Turnaround Time
1               10              0                10
2               5              10              15
3               8              15              23
Avg. waiting time = 8.33
Avg. turnaround time = 16.00
```

5	CPU SCHEDULING ALGORITHMS
	Shortest Job First (SJF)

AIM:

To write a C program for implementation of Shortest Job First (SJF) scheduling algorithms

ALGORITHM:

- Step 1: Enter number of processes.
- Step 2: Enter the **burst time** of all the processes.
- Step 3: Sort all the processes according to their **burst time and its respective process number**.
- Step 4: Find waiting time, **WT** of all the processes.
- Step 5: For the smallest process, **WT = 0**.
- Step 6: For all the next processes **i**, find waiting time by adding burst time of all the previously completed process.
- Step 7: Calculate **Turnaround time = WT + BT** for all the processes.
- Step 8: Calculate **average waiting time = total waiting time / no. of processes**.
- Step 9: Calculate **average turnaround time= total turnaround time / no. of processes**.

PROGRAM:

```
#include <stdio.h>

int main() {
    int bt[20], p[20], wt[20], tat[20], i, j, n, total = 0, totalT = 0, pos, temp;
    float avg_wt, avg_tat;

    // Get the number of processes
    printf("Enter number of processes: ");
    scanf("%d", &n);

    // Get burst times for each process
    printf("\nEnter Burst Time:\n");
    for (i = 0; i < n; i++) {
        printf("p%d: ", i + 1);
        scanf("%d", &bt[i]);
        p[i] = i + 1;
    }

    // Sorting of burst times in ascending order
    for (i = 0; i < n; i++) {
        pos = i;
        for (j = i + 1; j < n; j++) {
```

```

        if (bt[j] < bt[pos]) {
            pos = j;
        }
    }

    // Swap burst times
    temp = bt[i];
    bt[i] = bt[pos];
    bt[pos] = temp;

    // Swap process IDs
    temp = p[i];
    p[i] = p[pos];
    p[pos] = temp;
}

wt[0] = 0; // Waiting time for the first process is 0

// Calculating waiting time for all processes
for (i = 1; i < n; i++) {
    wt[i] = 0;
    for (j = 0; j < i; j++) {
        // Individual WT by adding burst time of all previous completed processes
        wt[i] += bt[j];
    }
    // Total waiting time
    total += wt[i];
}

// Calculate average waiting time
avg_wt = (float)total / n;

// Printing Process Details
printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time");
for (i = 0; i < n; i++) {
    // Turnaround time for each process
    tat[i] = bt[i] + wt[i];

    // Total turnaround time
    totalT += tat[i];

    // Displaying process details
    printf("\np%d\t\t %d\t\t %d\t\t %d", p[i], bt[i], wt[i], tat[i]);
}

// Calculate average turnaround time
avg_tat = (float)totalT / n;

```

```
// Display averages
printf("\n\nAverage Waiting Time = %.2f", avg_wt);
printf("\n\nAverage Turnaround Time = %.2f", avg_tat);

return 0;
}
```

Enter number of processes: 4

Enter Burst Time:

p1: 6

p2: 8

p3: 7

p4: 3

Process	Burst Time	Waiting Time	Turnaround Time
p4	3	0	3
p1	6	3	9
p3	7	9	16
p2	8	16	24

Average Waiting Time = 7.00

6	CPU SCHEDULING ALGORITHMS
	Shortest Remaining Time First (SRTF)

AIM:

To write a C program for implementation of Shortest Remaining Time First (SRTF) scheduling algorithms

ALGORITHM:

- Step 1: Enter number of processes.
- Step 2: Enter the **burst time** and arrival time of all the processes.
- Step 3: Sort all the processes according to their **arrival time and its respective process burst time**.
- Step 4 Traverse until all process gets completely executed.
 - Find process with minimum remaining time at every single time lap.
 - Reduce its time by 1.
 - Check if its remaining time becomes 0
 - Increment the counter of process completion.
 - Completion time of current process = current_time + 1;
 - Calculate waiting time for each completed process.
 - wt[i]= Completion time – arrival_time-burst_time
 - Increment time lap by one.
- Step 5: Calculate **Turnaround time** = WT + BT for all the processes.
- Step 6: Calculate **average waiting time** = total waiting time / no. of processes.
- Step 7 Calculate **average turnaround time**= total turnaround time / no. of processes.

PROGRAM:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 100

// Declaring the structure for process
struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
};

int main() {
    int n;
    struct process processes[MAX_PROCESSES];
    bool completed[MAX_PROCESSES] = {false};
    int current_time = 0;
```

```

int shortest_time = 0;
int shortest_index = 0;

// Get the number of processes
printf("Enter the number of processes: ");
scanf("%d", &n);

// Get arrival time and burst time for each process
for (int i = 0; i < n; i++) {
    printf("Enter arrival time and burst time for process %d: ", i + 1);
    scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
    processes[i].pid = i + 1;
    processes[i].remaining_time = processes[i].burst_time;
}
printf("\n");

// Process scheduling loop
while (true) {
    bool completed_all = true;
    shortest_index = -1; // Reset the shortest index each time

    for (int i = 0; i < n; i++) {
        if (!completed[i]) {
            completed_all = false;

            // Select the process with the shortest remaining time and arrival time less than or equal to
            // current time
            if (processes[i].arrival_time <= current_time &&
                (shortest_index == -1 || processes[i].remaining_time <
                 processes[shortest_index].remaining_time)) {
                shortest_index = i;
            }
        }
    }

    if (completed_all) {
        break; // Exit loop when all processes are completed
    }

    // Process the selected shortest job
    if (shortest_index != -1) {
        processes[shortest_index].remaining_time--;
        if (processes[shortest_index].remaining_time == 0) {
            completed[shortest_index] = true; // Mark process as completed
        }
    }
}

```



```

    current_time++; // Increment the current time
}

// Display the results
printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

int total_waiting_time = 0;
int total_turnaround_time = 0;

// Calculate waiting time and turnaround time for each process
for (int i = 0; i < n; i++) {
    int waiting_time = current_time - processes[i].burst_time - processes[i].arrival_time;
    int turnaround_time = current_time - processes[i].arrival_time;

    total_waiting_time += waiting_time;
    total_turnaround_time += turnaround_time;

    printf("%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].arrival_time,
    processes[i].burst_time, waiting_time, turnaround_time);
}

// Calculate average waiting time and turnaround time
float avg_waiting_time = (float) total_waiting_time / n;
float avg_turnaround_time = (float) total_turnaround_time / n;

printf("The Average Waiting Time: %.2f\n", avg_waiting_time);
printf("The Average Turnaround Time: %.2f\n", avg_turnaround_time);

return 0;
}

```

```

Enter the number of processes: 4
Enter arrival time and burst time for process 1: 2
3
Enter arrival time and burst time for process 2: 4
5
Enter arrival time and burst time for process 3: 3
5
Enter arrival time and burst time for process 4: 4
5

```

Process	Arrival Time	Burst Time	Waiting Time	Turnaround Time
1	2	3	-5	-2
2	4	5	-9	-4
3	3	5	-8	-3
4	4	5	-9	-4

```

The Average Waiting Time: -7.75
The Average Turnaround Time: -3.25

```

7	CPU SCHEDULING ALGORITHMS
	PRIORITY

AIM:

To write a C program for the implementation of Priority scheduling algorithms.

ALGORITHM:

Step 1: Take input for number of process, Burst time and priority of each process.

Step 2: Sort the processes in ascending order based on priority of each process.

Step 3: Find the waiting time for each process

Step 4: Calculate the turnaround time of each process.

Step 5: Calculate average waiting time and average turnaround time.

PROGRAM:

```
#include
<stdio.h> int
main() {
    int bt[20], p[20], pri[20], wt[20], tat[20], i, j, n, total = 0, totalT = 0, pos,
    temp; float avg_wt, avg_tat;

    // Get the number of processes
    printf("Enter number of processes:
    "); scanf("%d", &n);

    // Get burst time and priority for each
    process printf("\nEnter Burst Time and
    Priority:\n"); for (i = 0; i < n; i++) {
        printf("Enter Burst time p%d: ", i + 1);
        scanf("%d", &bt[i]);
        printf("Enter Priority p%d: ", i + 1);
        scanf("%d", &pri[i]);
        p[i] = i + 1; // Process ID
    }

    // Sorting of processes according to
    priority for (i = 0; i < n; i++) {
        pos = i;
        for (j = i + 1; j < n; j++)
            { if (pri[j] < pri[pos])
              {
                pos = j; // Find process with higher priority
              }
            }
    }
}
```

```

// Swap priorities, burst times, and process
IDs temp = pri[i];
pri[i] = pri[pos]; pri[pos] = temp;
temp = bt[i];
bt[i] = bt[pos];
bt[pos] = temp;

temp = p[i];
p[i] = p[pos];
p[pos] = temp;
}

wt[0] = 0; // Waiting time for the first process is 0

// Calculate waiting time for each process for (i = 1; i < n; i++) {
    wt[i] = 0;
    for (j = 0; j < i; j++) {
        // Calculate individual waiting time by adding burst time of all previous processes
        wt[i] += bt[j];
    }
    // Total waiting time
    total += wt[i];
}

// Calculate the average waiting time

avg_wt = (float)total / n;

// Print Process Table
printf("\nProcess\tBurst Time \tPriority \tWaiting Time\tTurnaround
Time\n"); for (i = 0; i < n; i++) {
    // Turnaround time of individual
    processes tat[i] = bt[i] + wt[i];
    // Total turnaround time
    totalT += tat[i];

    // Print process details
    printf("p%d\t%d\t%d\t%d\t%d\n", p[i], bt[i], pri[i], wt[i], tat[i]);
}

// Calculate average turnaround
time avg_tat = (float)totalT / n;

// Print average times
printf("\nAverage Waiting Time = %.2f", avg_wt);
printf("\nAverage Turnaround Time = %.2f\n", avg_tat);

return 0; }

```

Enter number of processes: 4

Enter Burst Time and Priority for each process:

Enter Burst time for p1: 5

Enter Priority for p1: 3

Enter Burst time for p2: 3

Enter Priority for p2: 1

Enter Burst time for p3: 8

Enter Priority for p3: 2

Enter Burst time for p4: 6

Enter Priority for p4: 4

Process	Burst Time	Priority	Waiting Time	Turnaround Time
p2	3	1	0	3
p3	8	2	3	11
p1	5	3	11	16
p4	6	4	16	22

Average Waiting Time = 7.500000

8	CPU SCHEDULING ALGORITHMS
	ROUND ROBIN SCHEDULING

AIM:

To write a C program for implementation of Round Robin scheduling algorithms.

ALGORITHM:

Step 1: Take input for number of process and Burst time of each process.

Step 2: Take input for time quantum

Step 3: Find the waiting time for each process

Step: 4: Calculate the turnaround time of each process.

Step 5: Calculate average waiting time and average turnaround time.

- Create an array **temp_burst_time[]** to keep track of remaining burst time of processes. This array is initially a copy of **bt[]** (burst times array)
- Create another array **wait_time[]** to store waiting times of processes. Initialize this array as 0.
- Initialize time : $t = 0$
- Keep traversing all the processes while they are not done. Do following for **i'th** process if it is not done yet.
 - If **temp_burst_time[i] > time_slot**
 - $t = t + \text{quantum}$
 - **temp_burst_time[i] -= time_slot;**
 - Else // Last cycle for this process
 - $t = t + \text{temp_burst_time}[i];$
 - $\text{wait_time}[i] = t - \text{bt}[i]$
 - **temp_burst_time[i] = 0;** // This process is over

PROGRAM:

```
#include<stdio.h>
```

```
int main() {
    // Input number of processes
    int n;
    printf("Enter Total Number of Processes: ");
    scanf("%d", &n);

    // Declare necessary arrays and variables
    int wait_time = 0, ta_time = 0;
    int arr_time[n], burst_time[n], temp_burst_time[n];
    int x = n; // Process count

    // Input process details (arrival time and burst time)
    for (int i = 0; i < n; i++) {
```

```

    printf("Enter Details of Process %d \n", i + 1);
    printf("Arrival Time: ");
    scanf("%d", &arr_time[i]);
    printf("Burst Time: ");
    scanf("%d", &burst_time[i]);
    temp_burst_time[i] = burst_time[i]; // Save the burst time for processing
}

// Input time slot for round robin scheduling
int time_slot;
printf("Enter Time Slot: ");
scanf("%d", &time_slot);

// Initialize counters
int total = 0, counter = 0, i = 0;
printf("Process ID    Burst Time    Turnaround Time    Waiting Time\n");

// Round Robin Scheduling
for (total = 0; x != 0; ) {
    // If the process can be completed within the time slot
    if (temp_burst_time[i] <= time_slot && temp_burst_time[i] > 0) {
        total += temp_burst_time[i];
        temp_burst_time[i] = 0; // Process completed
        counter = 1; // Mark process as completed
    }
    // If the process needs more time
    else if (temp_burst_time[i] > 0) {
        temp_burst_time[i] -= time_slot;
        total += time_slot;
    }
}

// When a process is completed
if (temp_burst_time[i] == 0 && counter == 1) {
    x--; // Decrease process count
    printf("\nProcess No %d \t\t %d\t\t\t %d\t\t\t %d", i + 1, burst_time[i],
        total - arr_time[i], total - arr_time[i] - burst_time[i]);

    // Calculate waiting and turnaround times
    wait_time += total - arr_time[i] - burst_time[i];
    ta_time += total - arr_time[i];
    counter = 0; // Reset counter for the next process
}

// Check if we have reached the end of the list of processes
if (i == n - 1) {
    i = 0; // Reset to the first process
}

```

```

// Move to the next process if it has arrived
else if (arr_time[i + 1] <= total) {
    i++;
}
// Otherwise, reset to the first process
else {
    i = 0;
}
}

// Calculate and print the average waiting and turnaround times
float average_wait_time = wait_time * 1.0 / n;
float average_turnaround_time = ta_time * 1.0 / n;

printf("\nAverage Waiting Time: %.2f", average_wait_time);
printf("\nAverage Turnaround Time: %.2f", average_turnaround_time);

return 0;
}

```

```

Enter Total Number of Processes: 3
Enter Details of Process 1
Arrival Time: 1
Burst Time: 5
Enter Details of Process 2
Arrival Time: 1
Burst Time: 3
Enter Details of Process 3
Arrival Time: 1
Burst Time: 3
Enter Time Slot: 2

```

Process ID	Burst Time	Turnaround Time	Waiting Time
Process No 2	3	8	5
Process No 3	3	9	6
Process No 1	5	10	5

```

Average Waiting Time: 5.333333

```

AIM:

To write a C-program to implement the producer – consumer problem using semaphores.

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the required variables.

Step 3: Initialize the buffer size and get maximum item you want to produce.

Step 4: Get the option, which you want to do either producer, consumer or exit from the operation.

Step 5: If you select the producer, check the buffer size if it is full the producer should not produce the item or otherwise produce the item and increase the value buffer size.

Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not consume the item or otherwise consume the item and decrease the value of buffer size.

Step 7: If you select exit come out of the program.

Step 8: Stop the program.

PROGRAM:

```
#include<stdio.h>
```

```
int mutex = 1, full = 0, empty = 3, x = 0;
```

```
void producer();
```

```
void consumer();
```

```
int wait(int);
```

```
int signal(int);
```

```
int main() {
```

```
    int n;
```

```
    printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
```

```
    while (1) {
```

```
        printf("\nENTER YOUR CHOICE\n");
```

```
        scanf("%d", &n);
```

```
        switch (n) {
```

```
            case 1:
```

```
                if ((mutex == 1) && (empty != 0)) {
```

```
                    producer();
```



```

        } else {
            printf("BUFFER IS FULL\n");
        }
        break;

    case 2:
        if ((mutex == 1) && (full != 0)) {
            consumer();
        } else {
            printf("BUFFER IS EMPTY\n");
        }
        break;

    case 3:
        exit(0);
        break;
    }
}
return 0;
}

int wait(int s) {
    return (--s);
}

int signal(int s) {
    return (++s);
}

void producer() {
    mutex = wait(mutex);
    full = signal(full);
    empty = wait(empty);
    x++;
    printf("\nProducer produces the item %d", x);
    mutex = signal(mutex);
}

void consumer() {
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    printf("\nConsumer consumes item %d", x);
    x--;
    mutex = signal(mutex);
}

```

```
1.PRODUCER  
2.CONSUMER  
3.EXIT
```

```
ENTER YOUR CHOICE: 1
```

```
Producer produces item 1
```

```
ENTER YOUR CHOICE: 1
```

```
Producer produces item 2
```

```
ENTER YOUR CHOICE: 2
```

```
Consumer consumes item 2
```

```
ENTER YOUR CHOICE: 2
```

```
Consumer consumes item 1
```

```
ENTER YOUR CHOICE: 2
```

```
BUFFER IS EMPTY
```

```
ENTER YOUR CHOICE: 1
```

```
Producer produces item 1
```

```
ENTER YOUR CHOICE: 3
```

AIM:

To write a c program to implement IPC using shared memory.

ALGORITHM:

- Step 1: Start the process
- Step 2: Declare the segment size
- Step 3: Create the shared memory
- Step 4: Read the data from the shared memory
- Step 5: Write the data to the shared memory
- Step 6: Edit the data
- Step 7: Stop the process

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>

#define SEGSIZE 100

int main(int argc, char *argv[])
{
    int shmid, cntr;
    key_t key;
    char *segptr;
    char buff[] = "poooda";

    // Generate a unique key for shared memory segment
    key = ftok(".", 's');

    // Try to get the shared memory segment, create if not exists
    if((shmid = shmget(key, SEGSIZE, IPC_CREAT | IPC_EXCL | 0666)) == -1)
    {
        // If shared memory already exists, just get it
        if((shmid = shmget(key, SEGSIZE, 0)) == -1)
        {
```

```

        perror("shmget");
        exit(1);
    }
}
else
{
    printf("Creating a new shared memory segment \n");
    printf("SHMID: %d\n", shmid);
}

// Display all shared memory segments
system("ipcs -m");

// Attach the shared memory segment to process's address space
if((segptr = (char*)shmat(shmid, 0, 0)) == (char*) -1)
{
    perror("shmat");
    exit(1);
}

// Writing data to shared memory
printf("Writing data to shared memory...\n");
strcpy(segptr, buff);
printf("DONE\n");

// Reading data from shared memory
printf("Reading data from shared memory...\n");
printf("DATA: %s\n", segptr);
printf("DONE\n");

// Removing shared memory segment
printf("Removing shared memory segment...\n");
if(shmctl(shmid, IPC_RMID, 0) == -1)
    printf("Can't remove shared memory segment...\n");
else
    printf("Removed successfully\n");

return 0;
}

```

Creating a new shared memory seg

----- Message Queues -----

key	msqid	owner	perms	used-bytes	messages
-----	-------	-------	-------	------------	----------

----- Shared Memory Segments -----

key	shmid	owner	perms	bytes	nattch	status
0x00000000	6	Lab703	600	524288	2	dest
0x00000000	27	Lab703	600	524288	2	dest
0x00000000	31	Lab703	600	524288	2	dest
0x00000000	34	Lab703	600	524288	2	dest
0x00000000	35	Lab703	600	4194304	2	dest
0x00000000	36	Lab703	600	33554432	2	dest
0x00000000	43	Lab703	600	524288	2	dest
0x7306032d	45	Lab703	666	100	0	

----- Semaphore Arrays -----

key	semid	owner	perms	nsems
-----	-------	-------	-------	-------

SHMID:45Writing data to shared memory...

DONE

Reading data from shared memory...

DATA:-poooda

DONE

Removing shared memory Segment...

AIM:

To write a C program to implement banker's algorithm for deadlock avoidance.

ALGORITHM:

Step-1: Start the program.

Step-2: Declare the memory for the process.

Step-3: Read the number of process, resources, allocation matrix and available matrix.

Step-4: Compare each and every process using the banker's algorithm.

Step-5: If the process is in safe state then it is not a deadlock process otherwise it is a deadlock process

Step-6: produce the result of state of process

Step-7: Stop the program

PROGRAM:

```
#include<stdio.h>
```

```
int main() {
```

```
    /* array will store at most 5 processes with 3 resources.
```

```
    If your processes or resources are greater than 5 and 3,
    then increase the size of the arrays */
```

```
    int p, c, count = 0, i, j, alc[5][3], max[5][3], need[5][3], safe[5], available[3], done[5], terminate = 0;
```

```
    // Input number of processes and resources
```

```
    printf("Enter the number of processes and resources: ");
```

```
    scanf("%d %d", &p, &c);
```

```
    // Input allocation matrix
```

```
    printf("Enter the allocation matrix for resources of all processes (%dx%d matrix):\n", p, c);
```

```
    for (i = 0; i < p; i++) {
```

```
        for (j = 0; j < c; j++) {
```

```
            scanf("%d", &alc[i][j]);
```

```
        }
```

```
    }
```

```
    // Input max resources required for each process
```

```
    printf("Enter the max resource requirements for each process (%dx%d matrix):\n", p, c);
```

```

for (i = 0; i < p; i++) {
    for (j = 0; j < c; j++) {
        scanf("%d", &max[i][j]);
    }
}

// Input available resources
printf("Enter the available resources: ");
for (i = 0; i < c; i++) {
    scanf("%d", &available[i]);
}

// Compute the need matrix (Need = Max - Allocation)
printf("\nNeed resources matrix is:\n");
for (i = 0; i < p; i++) {
    for (j = 0; j < c; j++) {
        need[i][j] = max[i][j] - alc[i][j];
        printf("%d\t", need[i][j]);
    }
    printf("\n");
}

// Initialize the done array to track if a process has completed
for (i = 0; i < p; i++) {
    done[i] = 0;
}

// Safety algorithm to find the safe sequence
while (count < p) {
    for (i = 0; i < p; i++) {
        if (done[i] == 0) {
            for (j = 0; j < c; j++) {
                if (need[i][j] > available[j]) {
                    break; // If need is greater than available, skip the process
                }
            }
        }
    }

    // If the process can proceed (i.e., need <= available), then it will execute
    if (j == c) {
        safe[count] = i; // Add the process to the safe sequence
        done[i] = 1; // Mark the process as done

        // Release resources held by the process and update available resources
    }
}

```

```

        for (j = 0; j < c; j++) {
            available[j] += alc[i][j];
        }

        count++;
        terminate = 0;
    } else {
        terminate++; // Process cannot be executed at this moment
    }
}

// If no process could be executed, it means no safe sequence exists
if (terminate == p) {
    printf("Safe sequence does not exist\n");
    break;
}

// If we found a safe sequence, print the results
if (terminate != p) {
    printf("\nAvailable resources after completion:\n");
    for (i = 0; i < c; i++) {
        printf("%d\t", available[i]);
    }

    printf("\nSafe sequence is:\n");
    for (i = 0; i < p; i++) {
        printf("p%d\t", safe[i]);
    }
    printf("\n");
}

return 0;
}

```



```
Enter the number of processes and resources: 3
3
Enter the allocation of resources for all processes (3x3 matrix):
0
1
0
2
0
0
3
0
2
Enter the max resource required by each process (3x3 matrix):
7
5
3
3
2
2
9
0
2
Enter the available resources: 3
3
2

Need resources matrix is:
7      4      3
1      2      2
6      0      0
Safe sequence does not exist.
```

12	MEMORY ALLOCATION METHODS FOR FIXED PARTITION
	FIRST FIT

AIM:

To write a C program for implementation memory allocation methods for fixed partition using first fit.

ALGORITHM:

Step 1: Input memory blocks with size and processes with size.

Step 2: Initialize all memory blocks as free.

Step 3: Start by picking each process and check if it can be assigned to current block.

Step 4 ; If size-of-process \leq size-of-block if yes then assign and check for next process.

Step 5: If not then keep checking the further blocks.

PROGRAM:

```
#include<stdio.h>
```

```
void main() {
    int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

    // Initialize flags and allocation arrays
    for(i = 0; i < 10; i++) {
        flags[i] = 0;    // 0 means not allocated
        allocation[i] = -1; // -1 means no process allocated to this block
    }

    // Get the number of blocks and their sizes
    printf("Enter number of blocks: ");
    scanf("%d", &bno);

    printf("\nEnter size of each block: ");
    for(i = 0; i < bno; i++)
        scanf("%d", &bsize[i]);

    // Get the number of processes and their sizes
    printf("\nEnter number of processes: ");
    scanf("%d", &pno);

    printf("\nEnter size of each process: ");
    for(i = 0; i < pno; i++)
        scanf("%d", &psize[i]);

    // First-fit allocation algorithm
```

```

for(i = 0; i < pno; i++) { // Loop through each process
    for(j = 0; j < bno; j++) { // Loop through each block
        if(flags[j] == 0 && bsize[j] >= psize[i]) { // If block is free and large enough
            allocation[j] = i; // Allocate process to block
            flags[j] = 1; // Mark the block as allocated
            break; // Move to the next process
        }
    }
}

// Display the allocation details
printf("\nBlock no.\tsize\tprocess no.\tsize");
for(i = 0; i < bno; i++) {
    printf("\n%d\t%d\t", i + 1, bsize[i]); // Block details
    if(flags[i] == 1) { // If the block is allocated
        printf("%d\t\t", allocation[i] + 1, psize[allocation[i]]);
    } else { // If the block is not allocated
        printf("Not allocated");
    }
}
}

```

Enter no. of blocks: 4

Enter size of each block: 100

200

300

400

Enter no. of processes: 3

Enter size of each process: 150

250

350

Block no.	Block size	Process no.	Process size
1	100	Not allocated	
2	200	1	150
3	300	2	250

13	MEMORY ALLOCATION METHODS FOR FIXED PARTITION
	BEST FIT

AIM:

To write a C program for implementation of FCFS and SJF scheduling algorithms.

ALGORITHM:

Step 1: Input memory blocks and processes with sizes.

Step 2: Initialize all memory blocks as free.

Step 2: Start by picking each process and find the minimum block size that can be assigned to current process i.e., find $\min(\text{blockSize}[1], \text{blockSize}[2], \dots, \text{blockSize}[n]) > \text{processSize}[\text{current}]$, if found then assign It to the current process.

Step 4: If not then leave that process and keep checking the further processes.

PROGRAM:

```
#include<stdio.h>

void main() {
    int fragment[20], b[20], p[20], i, j, nb, np, temp, lowest = 9999;
    static int barray[20], parray[20];

    printf("\n\t\t\tMemory Management Scheme - Best Fit");
    printf("\nEnter the number of blocks:");
    scanf("%d", &nb); // number of blocks

    printf("Enter the number of processes:");
    scanf("%d", &np); // number of processes

    printf("\nEnter the size of the blocks:-\n");
    for(i = 1; i <= nb; i++) { // input sizes of blocks
        printf("Block no.%d:", i);
        scanf("%d", &b[i]);
    }

    printf("\nEnter the size of the processes :-\n");
    for(i = 1; i <= np; i++) { // input sizes of processes
        printf("Process no.%d:", i);
        scanf("%d", &p[i]);
    }

    // Best Fit Allocation
```

```

for(i = 1; i <= np; i++) { // loop through processes
    for(j = 1; j <= nb; j++) { // loop through blocks
        if(barray[j] != 1) { // block not yet allocated
            temp = b[j] - p[i]; // calculate fragment (remaining space)
            if(temp >= 0 && lowest > temp) { // find the best fitting block
                parray[i] = j;
                lowest = temp;
            }
        }
    }
    fragment[i] = lowest; // store the fragment size
    barray[parray[i]] = 1; // mark block as allocated
    lowest = 10000; // reset the lowest fragment for the next process
}

// Display the results
printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
for(i = 1; i <= np && parray[i] != 0; i++) {
    printf("\n%d\t%d\t%d\t%d\t%d", i, p[i], parray[i], b[parray[i]], fragment[i]);
}
}

```

```

                                Memory Management Scheme - Best Fit
Enter the number of blocks: 3
Enter the number of processes: 3

Enter the size of the blocks:
Block no.1: 100
Block no.2: 500
Block no.3: 200

Enter the size of the processes:
Process no.1: 212
Process no.2: 417
Process no.3: 112

Process_no      Process_size    Block_no      Block_size     Fragment
1               212            2             500            288

```