

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**CSB4242 - OPERATING SYSTEMS LABORATORY**

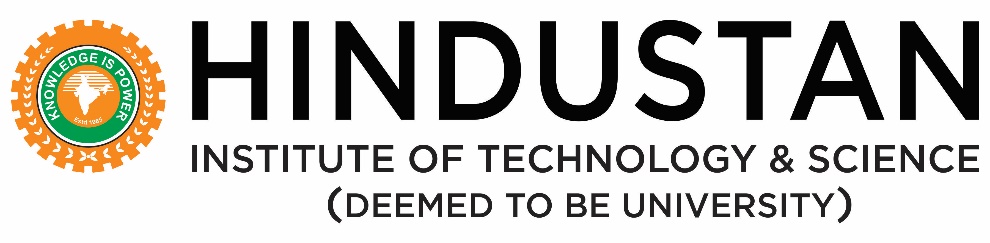
NAME : KONDREDDY SUMANTH KUMAR REDDY

REGISTER NUMBER : 19113105

SEMESTER : **IV**

YEAR : **II**

BRANCH/SECTION : **CSE - B**



**LABORATORY RECORD**

19113105

**REG NO:**

Name of the lab  **Operating Systems**  in the Department of **CSE**

Certified that this is a bonafide record of the work done by

**K. Sumanth Kumar Reddy** of **CSE 4B** Class in the

**virtual** laboratory during the year **2020 - 21**

……………………………

Signature of

Staff-in-charge

………………………….... …………………………...

Internal Examiner External Examiner

Name of the Examination: **University Practical Lab Examination**

Register No. : **19113105**

Date of the Examination : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**INDEX**

|  |  |  |
| --- | --- | --- |
| **E.NO** | **NAME OF THE EXPERIMENTS** | **PAGE NO.** |
| 1 | Program to report the behavior of the OS to get the CPU type and model, kernal version. | 04-06 |
| 2 | Shell programming – I | 07-12 |
| 3 | Shell programming – II | 13-15 |
| 4 | Program to get the amount of memory configured into the computer, amount of memory currently available | 16-17 |
| 5 | Process Scheduling Algorithms - FCFS, SJF, Priority, Round – robin. | 18-26 |
| 6 | Reader – writer’s problem. | 27-29 |
| 7 | Dining philosopher’s problem. | 30-32 |
| 8 | Banker’s algorithm. | 33-36 |
| 9 | First fit; best fit and worst fit file allocation strategy. | 37-42 |
| 10 | Write a program to create processes and threads. | 43-44 |

**CPU INFORMATION**

**Aim:** Program for CPU INFORMATION

**Program:**

#include<stdio.h>

#include<stdlib.h>

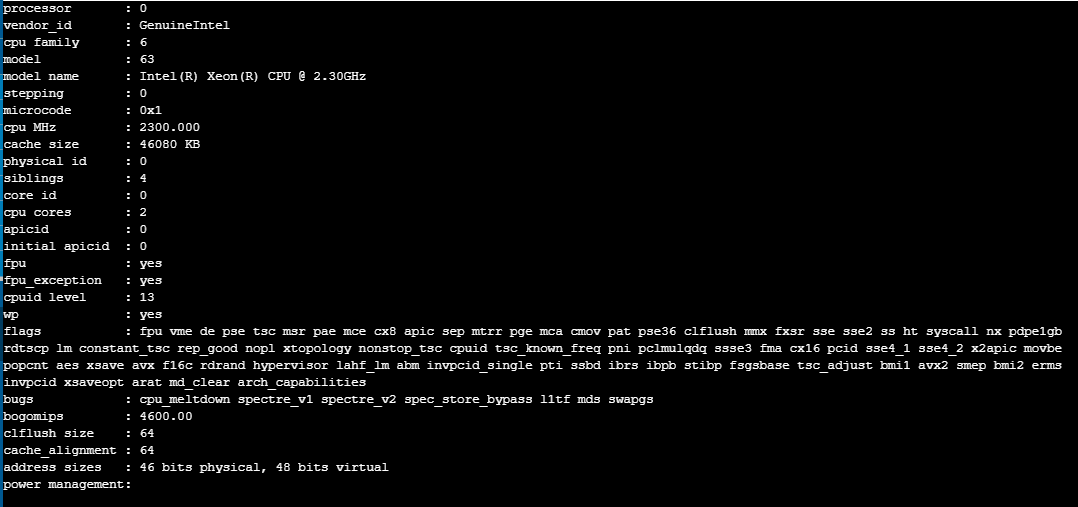
int main()

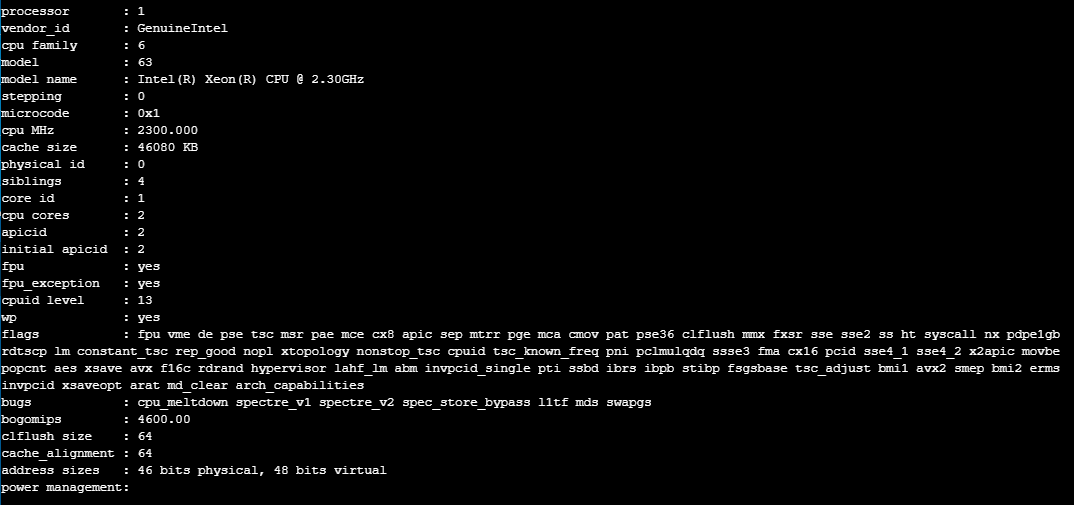
{

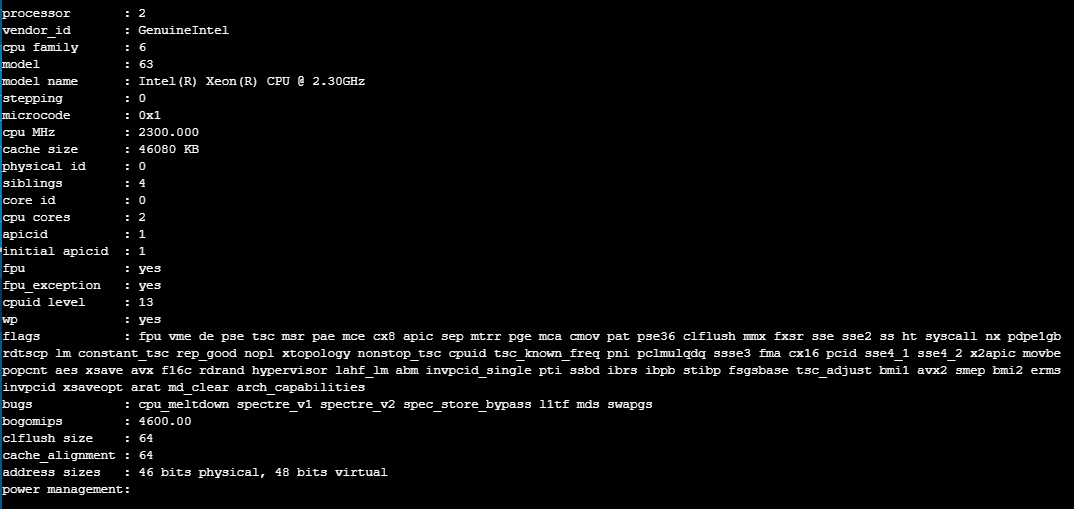
int r=system("cat /proc/cpuinfo");

}

**Output:**

****

****

****



**Result:**

Thus, implementation of program for CPU INFORMATION has been successfully.

**EX.NO.2 Shell programming – I**

**BASIC LIXUX COMMANDS**

**Basic Linux Commands**

AIM: To implement the basic Linux commands

**1.Who command**

Synbtax: $who

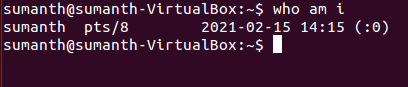
Description: It displays the information about all the users currently working on the system

os1

**2. Who iam I command**

Syntax: $who am i

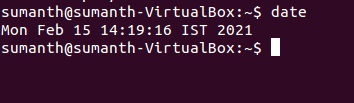
Description: It gives the details of the user currently logged in, also used to know in which terminal the user is currently logged on . It also informs the login name, the date and time of log in to the system.



**3. Date Command**

Syntax: $date

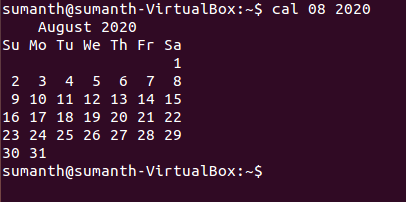
Description: It is used to print the system date and time



**4. cal command**

Syntax: cal[month][year]

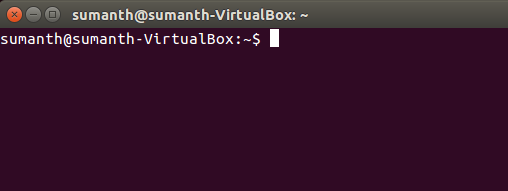
Description: It prints a calendar for the specified year on the standard output.



**5. clear command**

Syntax: $clear

Description: It is used to clear the screen

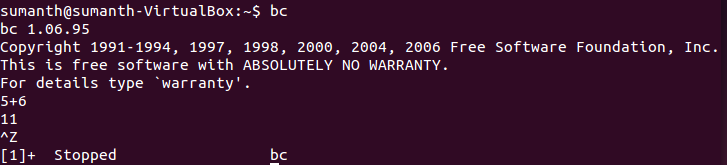


**6. Bc command**

Syntax: $bc 3+4

Description : This command is used to open binary calculator to perform simple mathematical

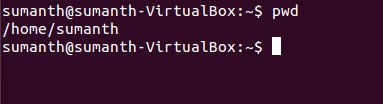
Calculations



**7. pwd command**

Syntax: $ pwd

Description: The pwd command is used to display the absolute pathname of the current working directory.

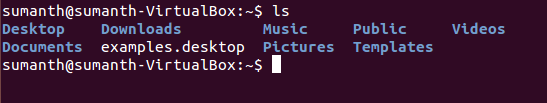


**8. ls command**

Syntax: $ ls option(s) argument(s)

Description: The list command is used to list out the names of all the files in the current working directory

Eg; $ ls-a , $ls-s

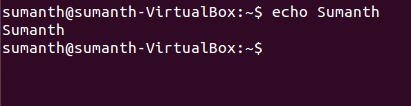


**9. Echo Commnad**

Syntax: $Echo String

Description: It simply echos each of the arguments on the standard output of the terminal

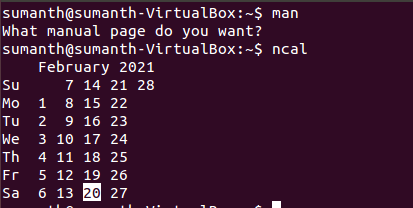
Eg; $echo shiva OUTPUT: shiva



**10. man command**

Syntax: $man

Description: It gives complete access to unix commands

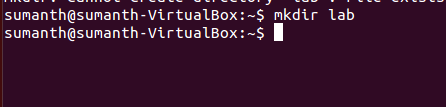


**Directory Manipulation commands**

**1. mkdir command**

Syntax: $mkdir [directory name]

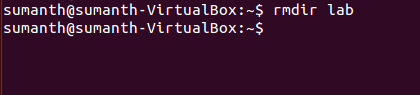
Description: This is used to create a new directory



**2. rmdir command**

Syntax: $ rmdir [directory name]

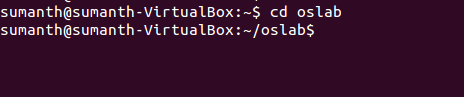
Description: Used to remove a directory



**3. cd command**

Syntax: $cd [directory name]

Description: Used to change from one working directory to another



**File Manupulation Command**

**1. cat command**

Syntax: $cat>filename

Description: It is used to create a file

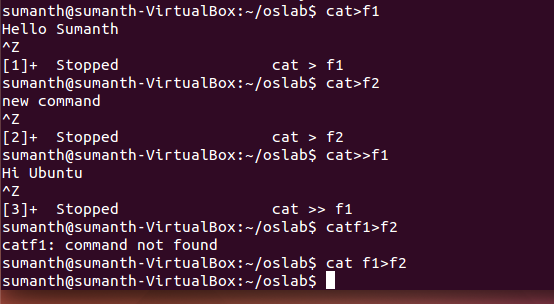
Eg: $ cat Remo

To append the file

$ cat>>Remo

To copy the f1 to f2

$cat f1>f2



**2. wc command**

Syntax: $wc options(s) file(s)

Description: The word count command is used to count the no of words, lines characters in a file or a group of file.

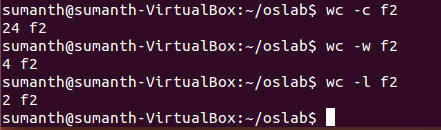
Option Description

-c counts the no of characters in the file

-w counts no of words

-l Counts no of lines in the file

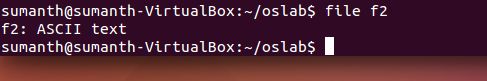
Eg $wc Remo where a is the file name



**3. File command**

Syntax: $file filenames(s)

Description: It is used to determine the type of file

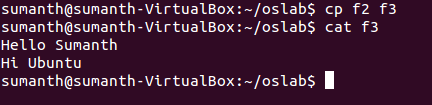


**4. cp command**

Syntax: $cp source\_file destination\_file

Description: It is used to copy one file to another

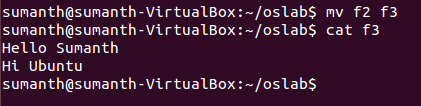
Eg: $cp file1 file2



**5. mv command**

Syntax: $mv source\_file destination\_file

Description: The move command is used to move a file within a directory with different names and also used to move a file within a directory with its original names. It is nothing but renaming of the file. Eg: mv file1 file2

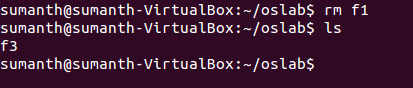


**6. rmcommand**

Syntax: $ rm option(s) file(s)

Description: The remove command is used for removing one or more files permanently from the disk

Eg: $rm file1



**7. Sort command**

Syntax: $sort filename

Description: Sort command is used to sort the file in ascending or descending order

Eg $sort softwares ASP, Borland c++, Java, Oracle.

$sort-r used to print in descending order



**EX.NO.3 SHELL PROGRAMMING -- II**

**2a) Sum and average of two numbers**

#A program to add two numbers

echo "Enter the two numbers";

read a b ;

c=$((a+b));

d=$((c/2))

echo "The sum of $a and $b is $c";

echo “The average of $a and $b is $d”;

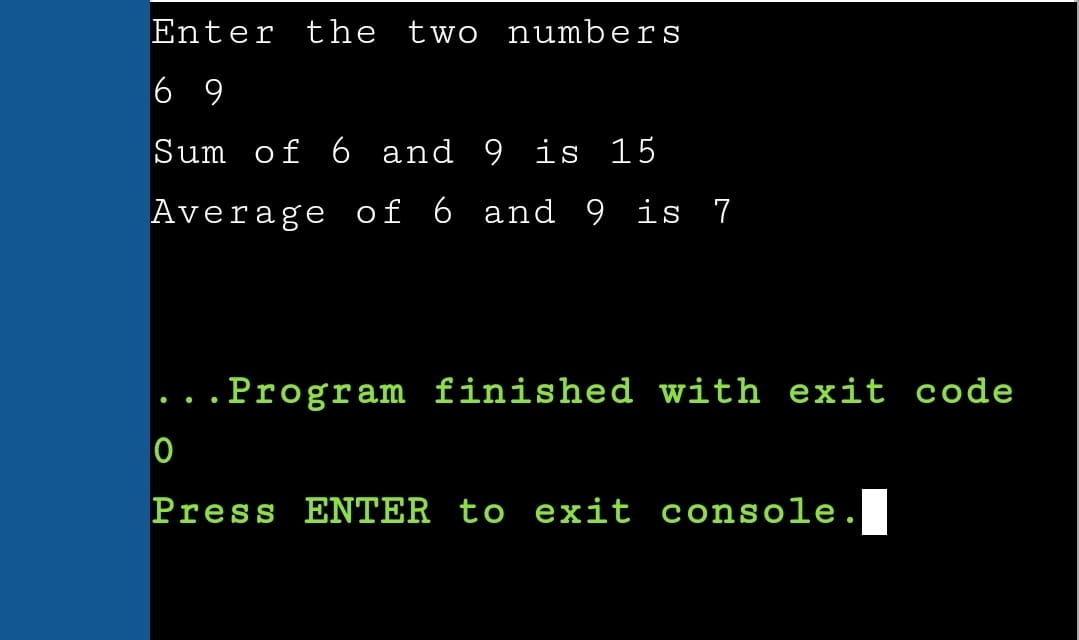
**Output**

Enter the two number

4 2

The sum of 4 and 2 is 6

The average of 4 and 2 is 3



**2b) Swapping two values**

#A program to swap two values

echo "Enter values for a and b"

read a b

c=$a

d=$b

temp=$b

b=$a

a=$temp

echo "Values of a and b before swapping is $c and $d"

echo "Values of a and b after swapping is $a and $b"

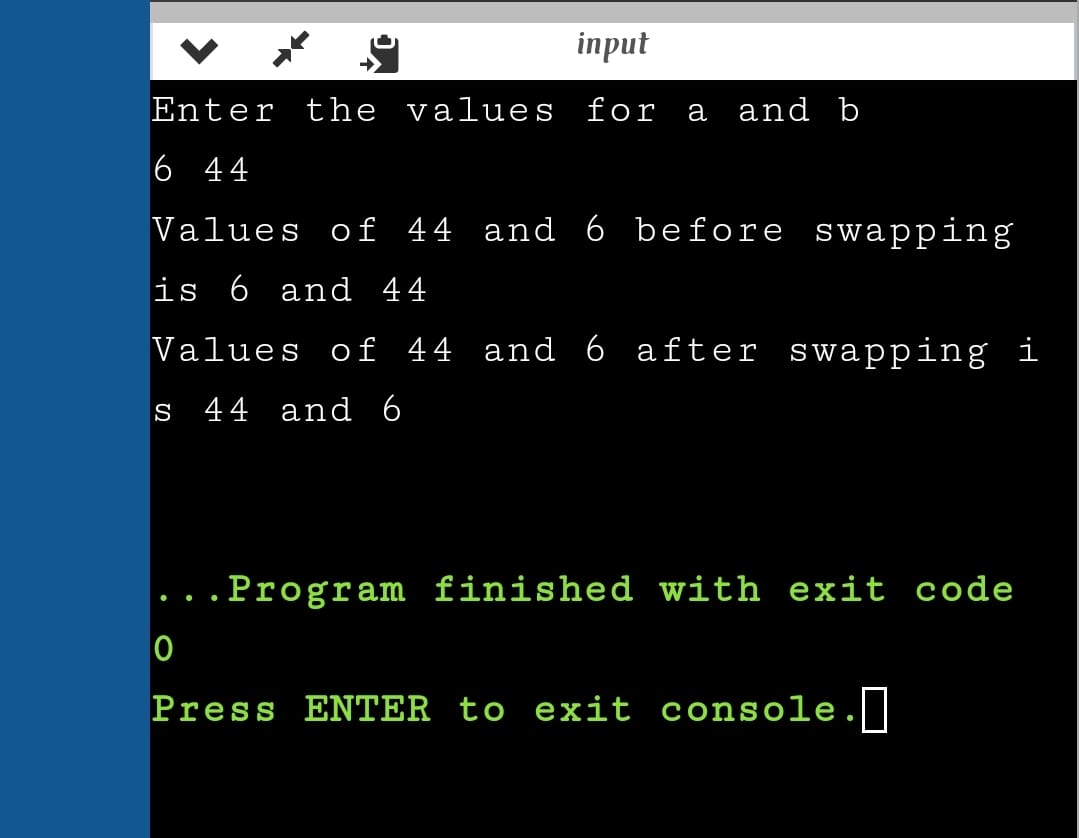
**Output**

Enter values for a and b

10 20

Values of a and b before swapping is 10 and 20

Values of a and b after swapping is 20 and 10



**2c) Maximum of three numbers**

#A program to find the greatest of three numbers

echo "Enter the three numbers"

read a b c

if(test $a -gt $b -a $a -gt $c)

then

echo "$a is greater"

elif(test $b -gt $c)

then

echo "$b is greater"

else

echo "$c is greater"

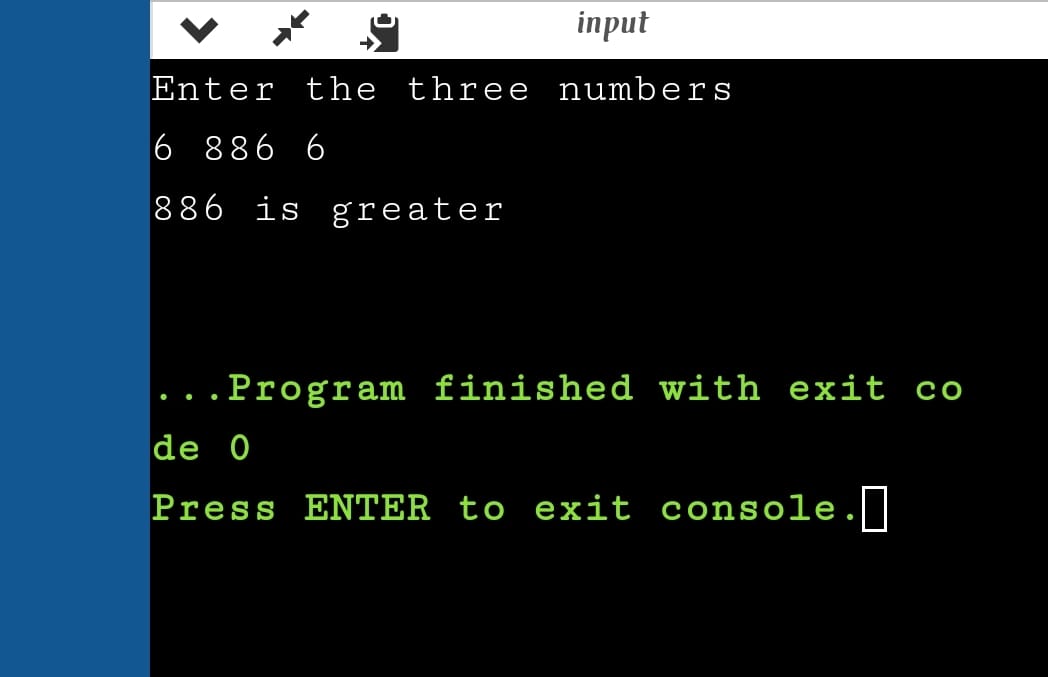
fi

**Output**

Enter the three numbers

7 8 3

8 is greater



**2d) Reversing a string**

#A program to reverse a string

echo "Enter the string"

read s

n=`echo $s | wc -c`

a=" "

while(test $n -gt 0)

do

x=`echo $s | cut -c $n`

a=`echo $a$x`

n=$((n-1))

done

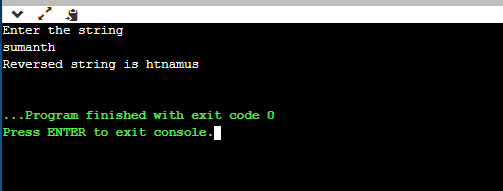
echo "Reversed string is $a"

**Output**

Enter the string

Aswin

Reversed string is niwsA



**EX.NO: 4 MEMORY INFORMATION**

**Aim:** Program for MEMORY INFORMATION

**Program:**

#include<stdio.h>

#include<stdlib.h>

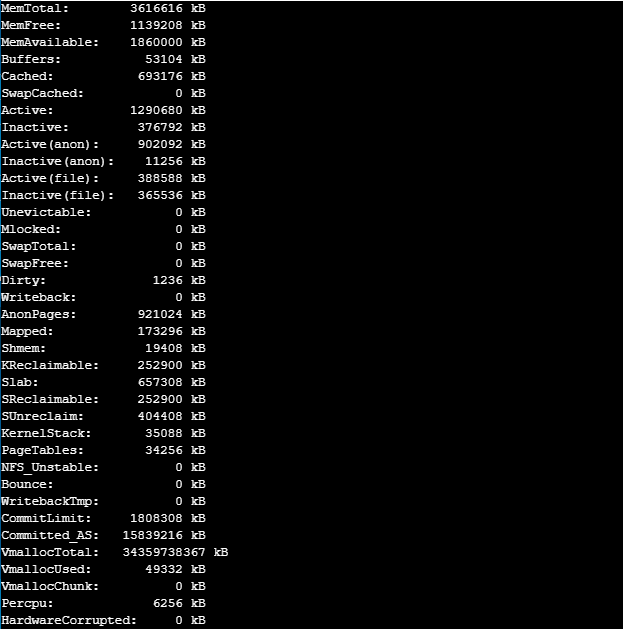
int main()

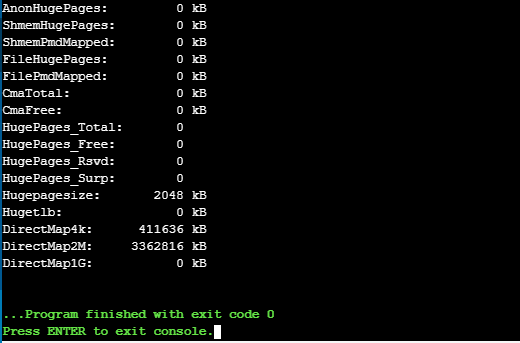
{

int r=system("cat /proc/meminfo");

}

**Output:**





**Result:**

Thus, the implementation of program for Memory Information has been executed successfully.

**EX.NO.5 Process Scheduling Algorithms**

**FIRST COME FIRST SERVE Scheduling**

**Aim:** To implement a C program for FCFS Scheduling

**Algorithm:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and give the values for CPU burst time.

Step 4: Start the Ready Q values which are stored in the order that they arrive in the ready queue.

Step 5: Set the waiting time of the first process as ’0’ and its turnaround time as its burst time.

Step 6: For each process in the ready queue, calculate Waiting and Turnaround Time.

Step 7: Calculate Average Waiting Time and Turnaround Time.

Step 8: Stop the process.

**Program:**

#include<stdio.h>

int main()

{

int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;

printf("Enter total number of processes:");

scanf("%d",&n);

printf("\nEnter Process Burst Time\n");

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

{

printf("P[%d]:",i+1);

scanf("%d",&bt[i]);

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=0;

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

wt[i]+=bt[j];

}

printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

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

{

tat[i]=bt[i]+wt[i];

avwt+=wt[i];

avtat+=tat[i];

printf("\nP[%d]\t\t%d\t\t%d\t\t\t%d",i+1,bt[i],wt[i],tat[i]);

}

avwt/=i;

avtat/=i;

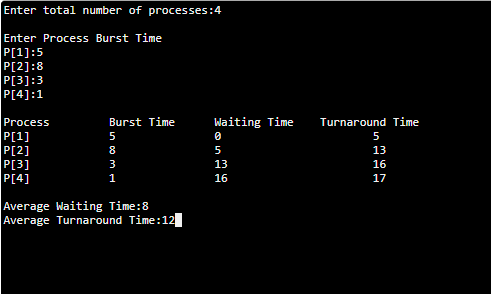
printf("\n\nAverage Waiting Time:%d",avwt);

printf("\nAverage Turnaround Time:%d",avtat);

return 0;

}

**Output:**



**Result:**

Thus C program for FCFS Scheduling has been executed successfully.

**SJF Scheduling**

**Aim:** To implement a C program for SJF Scheduling

**Algorithm:**

Step 1: Start the process  
Step 2: Accept the number of processes in the ready Queue  
Step 3: For each process in the ready Q, assign the process id and give the values for CPU burst time   
Step 4: Start the Ready Q according the shortest Burst time by sorting the burst time in ascending order.

Step 5: Set the waiting time of the first process as ’0‘ and its turnaround time as its burst time  
Step 6: Sort the processes names based on their Burt time  
Step 7: For each process in the ready queue, calculate Waiting and Turnaround Time

Step 8: Calculate Average Waiting Time and Turnaround Time

Step 9: Stop the process

**Program:**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int p[20], bt[20], wt[20], tat[20], i, k, n, temp;

float wtavg, tatavg;

printf("\nEnter the number of processes: ");

scanf("%d", &n);

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

{

p[i]=i+1;

printf("Enter Burst Time for P %d : ", i+1);

scanf("%d", &bt[i]);

}

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

for(k=i+1;k<n;k++)

if(bt[i]>bt[k])

{

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=p[i];

p[i]=p[k];

p[k]=temp;

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

for(i=1;i<n;i++)

{

wt[i] = wt[i-1] +bt[i-1];

tat[i] = tat[i-1] +bt[i];

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);

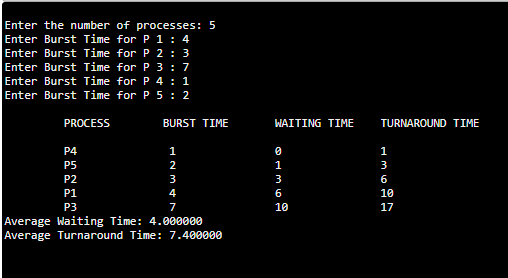
printf("\nAverage Waiting Time: %f ", wtavg/n);

printf("\nAverage Turnaround Time: %f", tatavg/n);

return 0;

}

**Output:**



**Result:**

Thus C program for SJF Scheduling has been executed successfully.

**Priority Scheduling**

**Aim:** To implement a C program for Priority Scheduling

**Algorithm:**

Step 1: Start the process  
Step 2: Accept the number of processes in the ready Queue  
Step 3: For each process in the ready Q, assign the process id, give the values for CPU burst time and priority.  
Step 4: Start the Ready Q values are stored in based on its priority so that process with higher priority are dealt with first.

Step 5: Set the waiting time of the first process as ’0‘ and its turnaround time as its burst time  
Step 6: Sort the processes names based on their Priority.  
Step 7: For each process in the ready queue, calculate Waiting and Turnaround Time

Step 8: Calculate Average Waiting Time and Turnaround Time

Step 9: Stop the process

**Program:**

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg\_wt,avg\_tat;

printf("Enter Total Number of Process:");

scanf("%d",&n);

printf("\nEnter Burst Time and Priority\n");

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

{

printf("\nP[%d]\n",i+1);

printf("Burst Time:");

scanf("%d",&bt[i]);

printf("Priority:");

scanf("%d",&pr[i]);

p[i]=i+1; //contains process number

}

//sorting burst time, priority and process number in ascending order using selection sort

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

{

pos=i;

for(j=i+1;j<n;j++)

{

if(pr[j]<pr[pos])

pos=j;

}

temp=pr[i];

pr[i]=pr[pos];

pr[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 first process is zero

//calculate waiting time

for(i=1;i<n;i++)

{

wt[i]=0;

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

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=total/n; //average waiting time

total=0;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

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

{

tat[i]=bt[i]+wt[i]; //calculate turnaround time

total+=tat[i];

printf("\nP[%d]\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=total/n; //average turnaround time

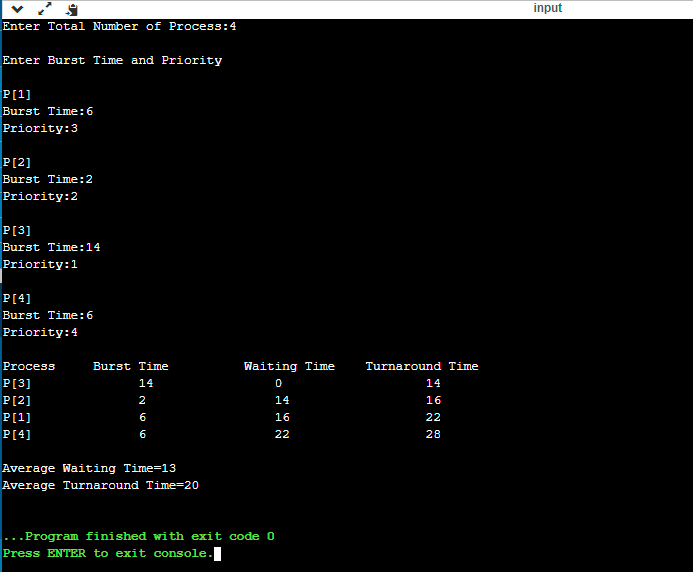
printf("\n\nAverage Waiting Time=%d",avg\_wt);

printf("\nAverage Turnaround Time=%d\n",avg\_tat);

return 0;

}

**Output:**



**Result:**

Thus C program for Priority Scheduling has been executed successfully.

**Round Robin Scheduling**

**Aim:** To implement a C program for Round Robin Scheduling

**Algorithm:**

Step 1: Start the process

Step 2: Enter the number of processes in the ready Queue and Time Quantum.

Step 3: In the ready Queue processes are arranged in first come first serve order.

Step 4: Quantum value is allocated to execute each process for a particular time.

Step 5: The first process is executed until the end of the quantum value. After this, an interrupt is generated and the state is saved.

Step 6: The CPU then moves to the next process and the same method is followed.

Step 7: Same steps are repeated till all the processes are over.

Step 8: Stop the process.

**Program:**

#include<stdio.h>

int main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;

float awt=0,att=0,temp=0;

printf("Enter the no of processes: ");

scanf("%d",&n);

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

{

printf("\nEnter Burst Time for process %d: ", i+1);

scanf("%d",&bu[i]);

ct[i]=bu[i];

}

printf("\nEnter the size of time slice: ");

scanf("%d",&t);

max=bu[0];

for(i=1;i<n;i++)

if(max<bu[i])

max=bu[i];

for(j=0;j<(max/t)+1;j++)

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

if(bu[i]!=0)

if(bu[i]<=t)

{

tat[i]=temp+bu[i];

temp=temp+bu[i];

bu[i]=0;

}

else

{

bu[i]=bu[i]-t;

temp=temp+t;

}

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

{

wa[i]=tat[i]-ct[i];

att+=tat[i];

awt+=wa[i];

}

printf("\nThe Average Turnaround time is %f",att/n);

printf("\nThe Average Waiting time is %f ",awt/n);

printf("\n\n\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");

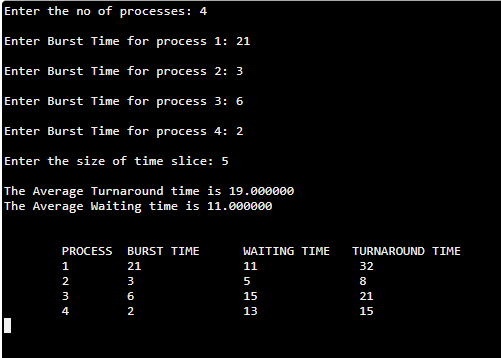
for(i=0;i<n;i++)

printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);

return 0;

}

**Output:**



**Result:**

Thus C program for Round Robin Scheduling has been executed successfully.

**EX.NO.6 Reader - Writer’s Problem**

**Aim:** A C program to solve Reader Writer Problem

**Algorithm:**

**Writer process:**

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

**Reader process:**

1. Reader requests the entry to critical section.
2. 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.
3. If not allowed, it keeps on waiting.

**Program:**

#include<semaphore.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

sem\_t x,y;

pthread\_t tid;

pthread\_t writerthreads[100],readerthreads[100];

int readercount = 0;

void \*reader(void\* param)

{

sem\_wait(&x);

readercount++;

if(readercount==1)

sem\_wait(&y);

sem\_post(&x);

printf("%d reader is inside\n",readercount);

usleep(3);

sem\_wait(&x);

readercount--;

if(readercount==0)

{

sem\_post(&y);

}

sem\_post(&x);

printf("%d Reader is leaving\n",readercount+1);

return NULL;

}

void \*writer(void\* param)

{

printf("Writer is trying to enter\n");

sem\_wait(&y);

printf("Writer has entered\n");

sem\_post(&y);

printf("Writer is leaving\n");

return NULL;

}

int main()

{

int n2,i;

printf("Enter the number of readers:");

scanf("%d",&n2);

printf("\n");

int n1[n2];

sem\_init(&x,0,1);

sem\_init(&y,0,1);

for(i=0;i<n2;i++)

{

pthread\_create(&writerthreads[i],NULL,reader,NULL);

pthread\_create(&readerthreads[i],NULL,writer,NULL);

}

for(i=0;i<n2;i++)

{

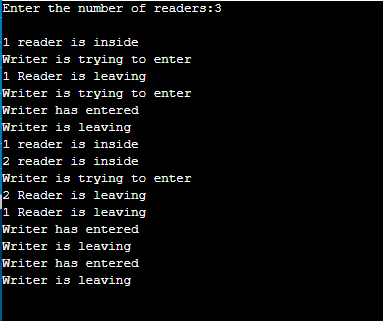
pthread\_join(writerthreads[i],NULL);

pthread\_join(readerthreads[i],NULL);

}

}

**Output:**



**Result:**

Thus, the Reader and Writer Problem has been executed successfully.

**EX.NO.7 Dining philosopher’s problem**

**Aim:** To implement a C program for Dining Philosopher Problem.

**Algorithm:**

Step 1: Start the process.

Step 2: Last philosopher first try to take right side fork and then left side fork.

Step 3: 5th person tries to take 4th Fork instead of 5th one.

Step 4: Since 4th Fork already taken by 4th the person, he gets nothing.

Step 5: But he left 5th Fork. Now the first person will take this 5th Fork and complete dinner and make 1st and 5th available for remaining people.

Step 6: Next 2nd person takes 1st fork and completes and releases 1st and 2nd.

Step 7: This continuous until all finishes dinner.

Step 8: Stop the process.

**Program:**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };

sem\_t mutex;

sem\_t S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY

&& state[LEFT] != EATING

&& state[RIGHT] != EATING) {

// state that eating

state[phnum] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n",

phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is Eating\n", phnum + 1);

// sem\_post(&S[phnum]) has no effect

// during takefork

// used to wake up hungry philosophers

// during putfork

sem\_post(&S[phnum]);

}

}

// take up chopsticks

void take\_fork(int phnum)

{

sem\_wait(&mutex);

// state that hungry

state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

// eat if neighbours are not eating

test(phnum);

sem\_post(&mutex);

// if unable to eat wait to be signalled

sem\_wait(&S[phnum]);

sleep(1);

}

// put down chopsticks

void put\_fork(int phnum)

{

sem\_wait(&mutex);

// state that thinking

state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n",

phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philospher(void\* num)

{

while (1) {

int\* i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

int main()

{

int i;

pthread\_t thread\_id[N];

// initialize the semaphores

sem\_init(&mutex, 0, 1);

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

sem\_init(&S[i], 0, 0);

for (i = 0; i < N; i++) {

// create philosopher processes

pthread\_create(&thread\_id[i], NULL,

philospher, &phil[i]);

printf("Philosopher %d is thinking\n", i + 1);

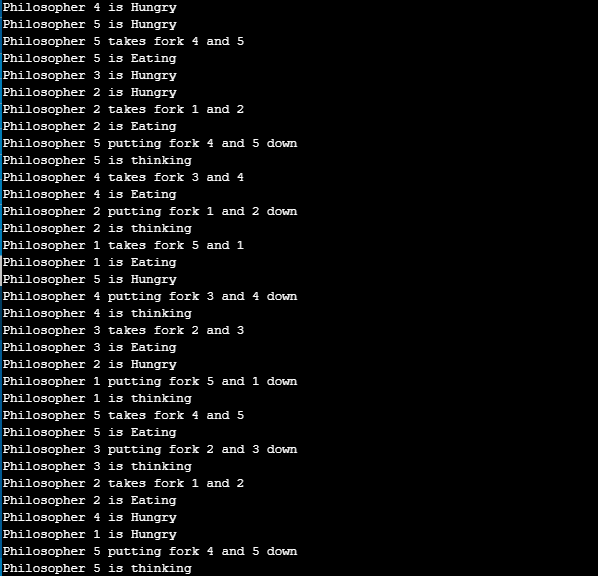
}

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

pthread\_join(thread\_id[i], NULL);

}

**Output:**



**Result:**

Thus C program for Dining Philosopher Problem has been executed successfully.

**EX.NO.8 Banker’s algorithm**

**Aim:** To implement Bankers algorithm using C.

**Algorithm:**

1. Enter the number of processes and resources. Create a resource request array R[i] for each process P[i]. If the Resource Requesti [j] equal to 'K', which means the process P[i] requires 'k' instances of Resources type R[j] in the system.
2. When the number of **requested resources** of each type is less than the **Need** resources, go to step 3 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:

If Request(i) <= Need  
Go to step 3;

1. And when the number of requested resources of each type is less than the available resource for each process, go to step (4). As the expression suggests:

If Request(i) <= Available  
Else Process P[i] must wait for the resource since it is not available for use.

1. When the requested resource is allocated to the process by changing state:

Available = Available - Request  
Allocation(i) = Allocation(i) + Request (i)  
Needi= Needi - Requesti

1. When the resource allocation state is safe, its resources are allocated to the process P(i). And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore the old resource-allocation state.

**Program:**

#include <stdio.h>

#include <conio.h>

int main()

{

int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10], safeSequence[10];

int p, r, i, j, process, count;

count = 0;

printf("Enter the no of processes : ");

scanf("%d", &p);

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

completed[i] = 0;

printf("\n\nEnter the no of resources : ");

scanf("%d", &r);

printf("\n\nEnter the Max Matrix for each process : ");

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

{

printf("\nFor process %d : ", i + 1);

for(j = 0; j < r; j++)

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

}

printf("\n\nEnter the allocation for each process : ");

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

{

printf("\nFor process %d : ",i + 1);

for(j = 0; j < r; j++)

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

}

printf("\n\nEnter the Available Resources : ");

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

scanf("%d", &avail[i]);

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

for(j = 0; j < r; j++)

need[i][j] = Max[i][j] - alloc[i][j];

do

{

printf("\n Max matrix:\tAllocation matrix:\n");

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

{

for( j = 0; j < r; j++)

printf("%d ", Max[i][j]);

printf("\t\t");

for( j = 0; j < r; j++)

printf("%d ", alloc[i][j]);

printf("\n");

}

process = -1;

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

{

if(completed[i] == 0)//if not completed

{

process = i ;

for(j = 0; j < r; j++)

{

if(avail[j] < need[i][j])

{

process = -1;

break;

}

}

}

if(process != -1)

break;

}

if(process != -1)

{

printf("\nProcess %d runs to completion!", process + 1);

safeSequence[count] = process + 1;

count++;

for(j = 0; j < r; j++)

{

avail[j] += alloc[process][j];

alloc[process][j] = 0;

Max[process][j] = 0;

completed[process] = 1;

}

}

}while(count != p && process != -1);

if(count == p)

{

printf("\nThe system is in a safe state!!\n");

printf("Safe Sequence : < ");

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

printf("%d ", safeSequence[i]);

printf(">\n");

}

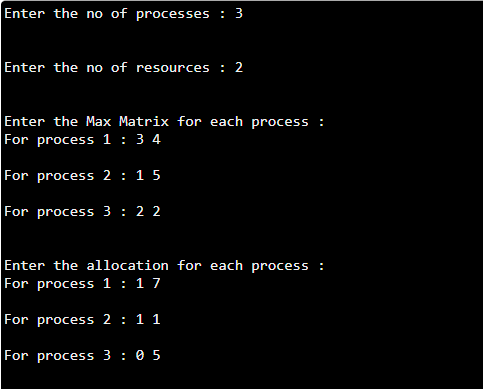
else

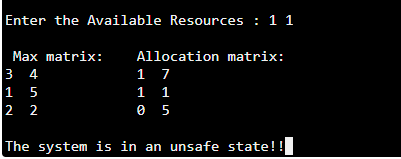
printf("\nThe system is in an unsafe state!!");

getch();

}

**Output:**

****



**Result:**

Thus, the implementation of Banker’s algorithm has been executed successfully.

**EX.NO.9 First fit; best fit and worst fit file allocation strategy**

**FIRST FIT MEMORY ALLOCATION**

**Aim:** To implement first fit memory allocation using C.

**Algorithm:**

1. Start the process.
2. In the first fit, the partition is allocated which is first sufficient from the top of Main Memory.
3. Input memory blocks with size and processes with size.
4. Initialize all memory blocks as free.
5. Start by picking each process and check if it can be assigned to current block.
6. If size-of-process <= size-of-block if yes then assign and check for next process.
7. If not then keep checking the further blocks.
8. Stop the process.

**Program:**

#include<stdio.h>

#define max 25

int main()

{

int frag[max],b[max],f[max],I,j,nb,nf,temp;

static int bf[max],ff[max];

printf(“\nEnter the number of blocks:”);

scanf(“%d”,&nb);

printf(“Enter the number of files:”);

scanf(“%d”,&nf);

printf(“\nEnter the size of the blocks:-\n”);

for(i=1;i<=nb;i++)

{

printf(“Block %d:”,i);

scanf(“%d”,&b[i]);

}

printf(“Enter the size of the files:-\n”);

for(i=1;i<=nf;i++)

{

printf(“File %d:”,i);

scanf(“%d”,&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i];

if(temp>=0)

{

ff[i]=j;

break;

}

}

}

frag[i]=temp;

bf[ff[i]]=1;

}

printf(“\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragment”);

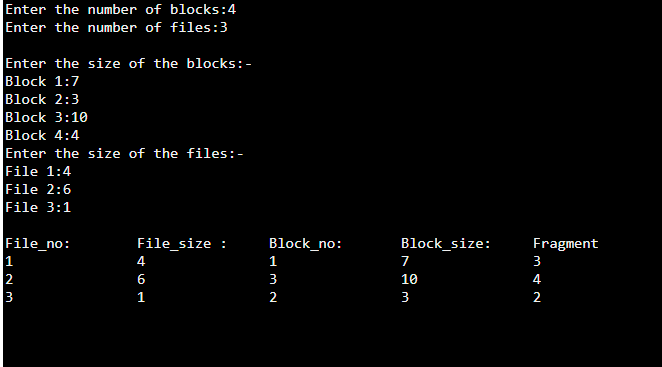
for(i=1;i<=nf;i++)

printf(“\n%d\t\t%d\t\t%d\t\t%d\t\t%d”,I,f[i],ff[i],b[ff[i]],frag[i]);

return 0;

}

**Output:**

****

**Result:**

Thus, the implementation First fit file allocation strategy has been executed successfully.

**BEST FIT MEMORY ALLOCATION**

**Aim:** To implement best fit memory allocation using C.

**Algorithm:**

1. Start the process.
2. Input memory blocks and processes with sizes.
3. Initialize all memory blocks as free.
4. Start by picking each process and find the minimum block size that can be assigned to current process i.e., find min(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
5. If not then leave that process and keep checking the further processes.
6. Stop the process.

**Program:**

#include<stdio.h>

#define max 25

int main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;

static int bf[max],ff[max];

printf("\nEnter the number of blocks:");

scanf("%d",&nb);

printf("Enter the number of files:");

scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n");

for(i=1;i<=nb;i++)

{

printf("Block %d:",i);

scanf("%d",&b[i]);

}

printf("Enter the size of the files:-\n");

for(i=1;i<=nf;i++)

{

printf("File %d:",i);

scanf("%d",&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i];

if(temp>=0)

if(lowest>temp)

{

ff[i]=j;

lowest=temp;

}

}

}

frag[i]=lowest;

bf[ff[i]]=1;

lowest=10000;

}

printf("\nFile\_no \tFile\_size \tBlock\_no \tBlock\_size \tFragment");

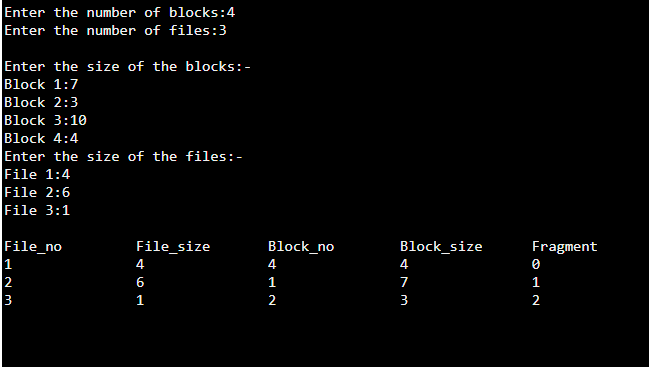
for(i=1;i<=nf && ff[i]!=0;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

return 0;

}

**Output:**



**Result:**

Thus, the implementation Best fit file allocation strategy has been executed successfully.

**WORST FIT MEMORY ALLOCATION**

**Aim:** To implement Worst fit memory allocation using C.

**Algorithm:**

1. Start the process.
2. Input memory blocks and processes with sizes.
3. Initialize all memory blocks as free.
4. Start by picking each process and find the maximum block size that can be assigned to current process i.e., find max(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
5. If not then leave that process and keep checking the further processes.
6. Stop the process.

**Program:**

#include<stdio.h>

#define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;

static int bf[max],ff[max];

printf("\nEnter the number of blocks:");

scanf("%d",&nb);

printf("Enter the number of files:");

scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n");

for(i=1;i<=nb;i++)

{

printf("Block %d:",i);

scanf("%d",&b[i]);

}

printf("Enter the size of the files:-\n");

for(i=1;i<=nf;i++)

{

printf("File %d:",i);

scanf("%d",&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1) //if bf[j] is not allocated

{

temp=b[j]-f[i];

if(temp>=0)

if(highest<temp)

{

ff[i]=j;

highest=temp;

}

}

}

frag[i]=highest;

bf[ff[i]]=1;

highest=0;

}

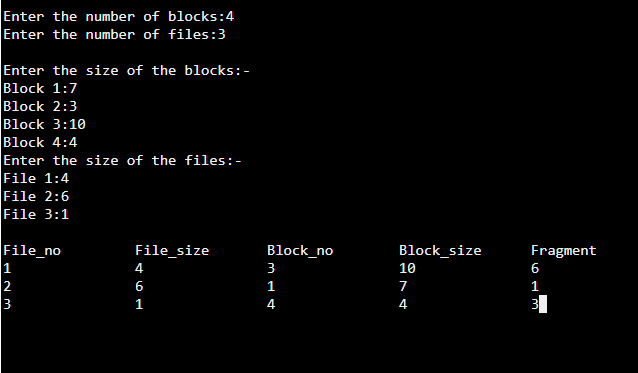
printf("\nFile\_no \tFile\_size \tBlock\_no \tBlock\_size \tFragment");

for(i=1;i<=nf;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

}

**Output:**



**Result:**

Thus, the implementation Worst fit file allocation strategy has been executed successfully.

**EX.NO.10 Create Processes and Threads**

**Aim:** Program for Process and Thread Creation.

**Algorithm:**

Step 1: Start the process.

Step 2: Create the child process. After this call, there are two processes, the existing one is called the parent process and the newly created one is called the child process.

Step 3: The fork() system call returns either of the three values −

* Negative value to indicate an error, i.e., unsuccessful in creating the child process.
* Returns a zero for child process.
* Returns a positive value for the parent process. This value is the process ID of the newly created child process.

Step 4: Stop the process.

**Program:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

pid\_t pid, mypid, myppid;

pid = getpid();

printf("Before fork: Process id is %d\n", pid);

pid = fork();

if (pid < 0) {

perror("fork() failure\n");

return 1;

}

// Child process

if (pid == 0) {

printf("This is child process\n");

mypid = getpid();

myppid = getppid();

printf("Process id is %d and PPID is %d\n", mypid, myppid);

} else { // Parent process

sleep(2);

printf("This is parent process\n");

mypid = getpid();

myppid = getppid();

printf("Process id is %d and PPID is %d\n", mypid, myppid);

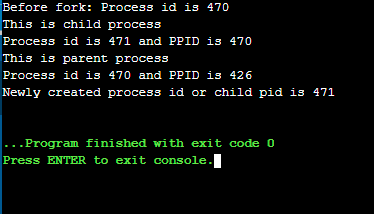
printf("Newly created process id or child pid is %d\n", pid);

}

return 0;

}

**Output:**



**Result:**

Thus, the implementation of Process and Thread Creation has been executed successfully.