PROGRAM 1

Aim: Study of hardware and software requirements of different operating systems (UNIX, LINUX, Windows XP, Windows 7/8).

Operating System (OS) :-

An operating system is an interface between a computer user and computer hardware. An operating system is a software which perform all the basic tasks like file management, memory management, process management, handeling input and output, and controlling peripheral devices such as disk driver and printers.

An operating system is software that enables application to interact with a compiler's hardware. The primary purpose of An operating system are to enables applications to interact with a computer's hardware and to manage a system's hardware and software resource. Some popular Os include Linux, Windows, VMS, OS/400, Z/OS, AIX etc.

Functionality and features of Operating System :-

Features of an operating system include,

Resources Management :- it refers to allocating, controlling and optimizing the utilization of various hardware And software resources Available within a computer system these include

Any entities that can be dynamically assigned And used by program when they execute on A computer.

2. Processor Management :- In multiprogramming environment, the OS decides which process gets the processor when and for how much time this function is called process scheduling. The OS allocates the processor to a process And deallocation processor when a process is no longer required.

3. Memory Management :- Memory management refers to management of primary memory or main memory. Main memory provides a fast storage that can be accessed directly by the CPU. The OS decides which process will get the memory And how much.

4. File Management :- A file system is normally organized into directories for easy navigation and usage. These directories may contain files And other directions.
An OS helps track of information, location, uses, status etc. the collective facilities are often known as file system.

5. Security :- Operating system security functions protect Against unauthorized access, data breaches, And malicious attacks by implementing security policies, access control, Authentication And cryptography.

6. Backup And Recovery :- Operating System functions related to Backup And recovery involve creating copies of data (Backup) and restoring them in case of loss or damage (recovery), ensuring data integrity And business continuity.

Types of Operating System :-

1. UNIX :- UNIX operating System also referred to as UNICS or UNIplexed information computing System is known for its features of multitasking, flexibility, stability etc. The Hierarchical file structure in the UNIX OS. helps in indexed storage And easy retrival this OS power lies in its kernal. it is An open source And promotes multitasking.

Hardware Requirement :-

Processor : Minimum of 1 GHz processor.
Memory (RAM) : Minimum of 1 GB RAM.
Hard Disk Drive : Minimum of 10 GB free disk space

Software Requirement :-

- UNIX - compatible operating system, such as Sun Solaries, IBM, AIX, HP-UX etc.
- Compiler And development tools (optional).
- X Windows System for graphical user interface (optional).

Networking tools for network communication. (optional).

2. LINUX :- Linux - based operating systems make use of what is known as the Linux kernal to manage device hardware resources And the software packages the power the remainder of the OS. Linux offers a high degree of capability And has a wide range of Application developers can use

Hardware Requirment :-

Processor : Minimum of 1 GHz processor.
Memory (RAM) : Minimum of 1 GB RAM (2 GB or more recomended for better performance).
Hard Disk Space: Minimum of 10 GB free disk space (20 GB or more recommended for better performance).

Software Requirment :-

Linux distribution, such as Ubuntu, fedoura, Centos. etc
Graphical users interface (optional).
Compiler And development tools (optional).
Networking tools for network comm. (optional).

3. Windows XP :- windows XP, released in 2001, was a popular Microsoft windows OS known for its user-friendly interface And improvements in performance And security, eventually

Succeeded by Windows Vista.

Hardware Requirements :-

Processor : Minimum of Pentium 233 MHz processor 1300 MH or high recommended.

Memory (RAM) :- Minimum of 64 MB RAM (128 MB or high recommended).

Hard Disk space : Minimum 15 GB free disk space.

Software Requirement :-

Windows XP operating system.

DirectX 9 graphical device with WDDM driver (optimal for graphical user interface)

Networking tools for network communication.

Windows 7/8 :- Windows 7 And windows 8 were both personal computer operating system. developed by Microsoft.

with windows 7 serving as the successor to windows Vista And windows 8 introducing a new "Modern" title based user interface.

Hardware requirement :-

Processor : Minimum of 1 GHz processor (1GHz or high recommended).

Memory (RAM) : Minimum of 1 GB RAM (2 GB or higher recommended).

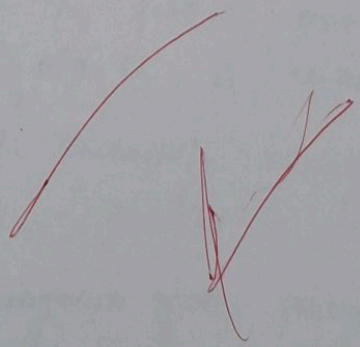
Hard Disk Space :- Minimum of 16 GB free disk (20 GB or higher recommended).

Software Requirement :-

Windows 7 or windows 8 operating system.

DirectX 9 graphical device with WDDM 1.0 or higher driver (optimal for graphical user interface).

Networking tools for network communication. (optional).



ITM-2SK
13

PROGRAM-2

Objective :- Execute various UNIX system calls for,

i) Processor management
ii) File Management
iii) input/output System calls.

1. **Process Management - System Calls -**

Process management uses certain system calls, They are explained below -

a. To create a new process - fork() is used.

b. To run a new program - exec() is used.

c. To make the process to wait - wait() is used. Sleep()

d. To determinate the process - exit() is used.

e. To find the unique process id - getpid() is used.

f. To find the parent process id - getpid() is used.

g. To bias the currently runing process property - nice() is used.

i) Example program for example of fork().

```
#include <stdio.h>
#include <unistd.h> --> for fork() and sleep() function.
int main(){
    int i, pid;
    pid = fork();
```

ITM-2SK
12

```
if (pid == 0){
        for(i=0; i<20; i++){
            sleep(2)
            printf("from child process %d\n", i);
        }
}
else{
    for(i=0; i<20; i=i+2){
        sleep(2);
        printf(" from parent process %d\n",i);
    }
}
return 0;
}
```

ii) Example program for example of execv

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
        int i;
    char *p[] = {"./hello", NULL};
    int pid;
    pid = fork();
    if (pid == 0){
            for(i=0; i<20; i++){
                sleep(2);
            printf("from child process %d\n",i);
    }   }
```

```
else { for(i=0; i<10; i=i+2){
        Sleep(2)
        printf("from parent process %d\n", i);
        execv(p[0], p);
      }
  }
  return 0;
}
```

2. File Management - System calls -

There are four system calls for file management-

a. open(): System call is used to know the file description to user - created file. Since read and write use file description as their 1st parameter so to know the file descriptor open() system call is used.

b. read(): System call is used to read the content from the file. It can also be used to read the input file from the keyboard by specifying the 0 as file descriptor.

c. write(): System call is used to write the content to the file.

d. Close(): System call is used to close the opened file. It tell the operating system that you are done with file and close the file.

```
#include <unistd.h>   // for read and write function
                      for input from keyboard.
#include <fcntl.h>    // for open function and
                      O_CREAT and O_RDWR
#include <stdio.h>    flags.
int main(){
    int n, fd;
    char buff[1024];
    printf("Enter text to write in the file:\n");
    n = read(0, buff, 1024);
    fd = open("Amit", O_CREAT | O_RDWR, 0777);
    write(fd, buff, n);
    write(1, buff, n);
    int close(int fd);
    return 0;
}
```

3. Input / Output System Calls :-

Basically, there are total 4 types of I/O system calls:

a. create : Used to create a new empty file.
b. open : Used to open the file for reading, writting both.
c. close : Tells the operating system you are done with a file descriptor and close the file which pointed by fd.
d. read : From the file indicated by the file descriptor fd. the read() function reads cnt bytes of input into the memory area indicated by buf.

A successful read() updates the access time for the file.

e. write : writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). if cnt is zero, write() simply return 0 without attempting any other action.

```
#include <unistd.h> // for read and write function from input from keyboard.
#include <stdio.h>
Void main(){
    char c;
    int a=1, i;
    while(a!=0){
        read(0, &a, 3);
        i=a;
        write(1, &i, 3);
        write(1, "\n", 1);
    }
}
```

Experiment No - 3

Objective : Implement CPU Scheduling Policies.
FCFS.

Code :

```
#include<stdio.h>
Void calcCT (int p[], int n, int bt[], int at[],
int ct[]){
        ct[0] = at[0] + bt[0];
        for(int i=1; i<n; i++){
ct[i] = (ct[i-1] > at[i]) ? ct[i-1] + bt[i] : at[i] + bt[i];
    }   }
Void calcWT (int p[], int n, int bt[], int wt[], int
    at[], int ct[]){
        for(int i=0; i<n; i++){
            wt[i] = ct[i] - at[i] - bt[i]};
    }   }
Void calcTAT (int p[], int n; int bt[], int wt[],
    int tat[]){
        for(int i=0; i<n; i++){
            tat[i] = bt[i] + wt[i];
    }   }
Void calcRT( int p[], int n, int bt[], int wt[]){
        int wt[]
    for (int i=0; i<n; i++){
        wt[i] = wt[i];
}   }
```

```
Void calcAvgTime(int P[], int n, int bt[], int dt[]){
    int ct[n], wt[n], tat[n], rt[n];
    int totalWT=0, totalTAT=0, totalRT=0;
    calcCT(p, n, bt, at, ct);
    calcWT(p, n, bt, wt, at, ct);
    calcTAT(p, n, bt, wt, tat);
    calcRT(p, n, bt, rt, wt);
  printf("Process ID  Arrival Time  Burst Time
Completion time,  Waiting time  Twin Around
  time  Reponce time \n");
for(int i=0; i<n; i++){
    totalWT= totalWT + wt[i];
    totalTAT= totalTAT + tat[i];
    totalRT = totalRT + rt[i];
  printf("%d  %d  %d  %d  %d  %d  %d\n",
      p[i], at[i], bt[i], ct[i], wt[i], tat[i], rt[i]);
}
float avgWT = %.2f\n", avgWT)/(float)totalWT/n;
float avgTAT = (float)totalTAT/n;
float avgRT = (float)totalRT/n;
printf("Avg WT = %.2f\n", avgWT);
printf("Avg TAT = %.2f\n", avgTAT);
printf("Avg RT = %.2f\n", avgRT);
int main(){
  int p[] = {1,2,3}
  int n = 3;
  int bt[] = {10,5,8}
  int at[] = {0,2,4};
calcAvgTime(p,n, bt, at);
}
```

Output :

Process ID	Arrival time	Burst time	Comple. time	waiting time
1	0	10	10	0
2	2	5	15	8
3	4	8	23	11

Turn Around time	Response time
10	0
13	8
19	11

Avg WT = 6.33
Avg TAT = 14.00
Avg RT = 6.33

ITM-2SK
12

Experiment No- 4

⇒ Implementing CPU Scheduling Policy:
Shortest Job first (SJF):-

Code:-

```
#include <stdio.h>
typedef struct {
        int at;
        int bt;
        int ct;
        int tt;
        int wt;
        int pid;
} process;

Void calculateTimes (Process proc[], int n){
        int currentTime = 0;
    for( int i=0; i<n ; i++){
        if (proc[i].at > currentTime){
                currentTime = proc[i].at;
            }
        proc[i].ct = currentTime + proc[i].bt;
        proc[i].tt = proc[i].ct - proc[i].at;
        proc[i].wt = proc[i].tt - proc[i].bt;
        currentTime = proc[i].ct;
    }
}

Void displayResults (Process proc[], int n){
        float total_wt = 0;
        float total_tt = 0;
```

```
printf("PID\tAT\tBT\tCT\tTT\tWT\n");
for(int i=0;i<n;i++){
    total_wt += proc[i].wt;
    total_wt += proc[i].tt;
    printf("%d\t%d\t%d\t%d\t%d\t%d\n",
proc[i].pid, proc[i].at, proc[i].bt, proc[i].ct,
  proc[i].tt, proc[i].wt);
}
Void sortProccessByArrival(Process proc[], int n){
  for(int i=0; i<n-1; i++){
    for(int j=i+1; j<n; j++){
      if(proc[i].at > proc[j].at || (proc[i].at ==
      proc[j].at && proc[i].pid > proc[j].pid)){
          process temp = proc[i];
          proc[i] = proc[j];
          proc[j] = temp;
      }
    }
  }
int main(){
  int n;
  printf("Enter the number of processes :");
  Scanf("%d", &n);
  Process proc[n];
```

```
for (int i=0; i<n; i++) {
    proc[i].pid = i+1;
    printf("Enter arrival time and burst time for
    process %d", i+1);
    }
    SortProcessByArrival(proc, n);
    calculateTimes(proc, n);
    displayResults(proc, n);
    return 0;
}
```

Output :

```
Enter the number of processes : 4
Enter arrival time and burst time for process 1 : 0 2
Enter arrival time and burst time for processes 2 : 1 2
Enter arrival time and burst time for process 3 : 5 3
Enter arrival time and burst time for process 4 : 6 4
```

PID	AT	BT	CT	TAT	WT
1	0	2	2	2	0
2	1	2	4	3	1
3	5	3	8	3	0
4	6	4	12	6	2

Average waiting Time = 0.75

Average Turnaround Time = 3.50

Experiment - 5

Implementing Preemptive Priority CPU Scheduling policy.

```
# include <stdio.h>
# define MIN -9999
struct proc {
    int no, at, bt, rt, ct, wt, tat, pri, temp;
};
struct proc read(int i) {
    struct proc p;
    printf("\n Process No : %d\n", i);
    p.no = i;
    printf(" Enter Arrival time : ");
    scanf("%d", &p.at);
    printf(" Enter burst time : ");
    scanf("%d", &p.bt);
    p.rt = p.bt;
    printf(" Enter priority : ");
    scanf("%d ", &p.pri);
    p.temp = p.pri;
    return p;
}
Void main() {
    int i, n, c, remaining, max_val, max_index;
    struct proc p[10], temp;
    float avgtat = 0, avgwt = 0;
```

```
printf("<--Highest Priority First Scheduling
      Algorithm (preemptive) -->\n");
printf("Enter Number of process :");
scanf("%d", &n);
for(i=0;i<n;i++){
      P[i] = read(i+1);
}
remaining = n;
for(i=0; i<n-1;i++){
    for(int j=0; j<n-i-1; j++){
        if(P[j].at > P[j+1].at){
              temp = P[j];
              P[j] = P[j+1];
              P[j+1] = temp;
        }
    max_val = P[0].temp;
    max_index = 0;
    for(int j=0; j<n && P[j].at <= P[0].at; j++){
        if(P[j].temp > max_val){
            max_val = P[j].temp; max_index = j;
        }
    i = max_index;
    c = P[i].ct = P[i].at+1;
    P[i].rt--;
    if(P[i].rt==0){
        P[i].temp = MIN;
    }  remaining--;
```

ITM-2SK
12

```
while(remaining > 0){
    max_val = MIN;
    max_index = -1;
for(int j=0; j<n && P[j].at <= c; j++){
    if( P[j].temp > max_val &&
            P[j].rt > 0){
        max_val = P[j].temp;
        max_index = j;
    }
}
if( max_index == -1){
    c++;
    continue;
}
i= max_index;
P[i].ct = c+ c+1;
P[i].rt --;
if(P[i].rt == 0){
    P[i].temp = MIN;
    remaining --;
}
}
printf("\n Process No \t AT \t BT \t Pri \t CT \t
TAT \t WT \t RT \n");
for(i=0; i<n; i++){
    P[i].tat = P[i].ct - P[i].at;
    avgtat += P[i].tat;
    P[i].wt = P[i].tat - P[i].bt;
    avgwt += P[i].wt;
```

ITM-2SK

```
printf("P %d \t %d \t %d \t %d \t %d \t %d \t %d \t %d \n",
        P[i].no, P[i].at, P[i].bt,
P[i].pri, P[i].ct, P[i].tat, P[i].wt, P[i].rt);
}
avg tat /= n;
avgwt /= n;
printf("\n Average TAT = %.2f", avgtat);
printf("\n Average WT = %.2f", avgwt);
}
```

o/p:

<= Highest Priority First Scheduling Algorithm (Precemptive->

Enter Number of processes : 3

Process No : 1
Enter Arrival Time :0
Enter burst Time :5
Enter Priority :10

Process No : 2
Enter Arrival Time : 1
Enter burst time : 4
Enter priority : 20

Process No : 3
Enter Arrival time : 2
Enter burst time : 2
Enter priority : 30

Process No	AT	BT	Pri	CT	TAT	WT	RT
P1	0	5	10	11	11	6	0
P2	1	4	20	7	6	2	0
P3	2	2	30	4	2	0	0

Average ~~Turn~~ TAT = 6.33

Average WT = 2.67

ITM-2SK

12

Experiment - 6

Implementing Producer / Consumer problem

Implement the solution of Bounded Buffer (producer - consumer) problem using inter process communication techniques - semaphores.

```
# include <stdio.h>
# include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 10;
int x=0;
void wait(int *s){
    --(*s);
}
void signal(int *s){
    ++(*s);
}
void producer (int count){
  for(int i=0 ; i< count ; i++){
      if(empty == 0){
          printf("Buffer is full! Can not
      produce more. \n");
          break;
      }
      wait(&mutex);
      wait(&empty);
      signal(&full);
      x++;
      printf("Produced item %d\n", x);
      signal(&mutex);
  }
}
```

ITM-2SK
12

```
void consume (int count){
for(int i=0; i< count ;i++){
   printf(" Buffer is empty! Can not consume.
more \n");
        break;
  }
wait(&mutex);
wait(&full);
signal (&empty);
  printf(" Consumed item %d\n", a);
  a--;
  signal (& mutex);
 }
}
int main (){
  int choice, num;
  printf("\n ---Producer - Consumer - Simulation ---\n");
  while(1){
    case 1;
    printf("\n1. Produce\n2. Consume \n3. Exit \n
      Enter your choice:");
    scanf("%d", &choice);
    switch(choice){
      case 1: printf(" How many items do
you want to produce? ");
              scanf("%d", &num);
              produce (num);
              break;
```

ITM-25K
12

```
case 2: printf("How many items do you want to consume?");
        scanf("%d", &num);
        consume(num);
        break;
case 3: exit(0);
default: printf("Invalid choice!\n");
}
printf("Buffer state => Full: %d, Empty: %d\n", full, empty);
}
return 0;
}
```

output :- ---producer-Consumer-Simulation---

1. produce
2. Consumer
3. Exit

Enter your choice: 1

How many items do you want to produce? 2

produced item 1.

produced item 2

Buffer state => Full: 2, Empty: 0

1. Produce
2. Consume
3. Exit

Enter your choice: 2

How many items do you want to consume? 1.

consumed item 1.

1. Produce
2. consume
3. Exit.

Experiment -7

Implement the solution for Reader-writers problem using inter process communication technique- Semaphore

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

sem_t mutex;
sem_t db;

void * reader (void* arg) {
    int id = ((int)arg);
    sem_wait(&mutex);
    rc++;
    if (rc == 1) {
        sem_wait(&db);
    }
    sem_post(&mutex);
    printf("Reader %d is reading \n", id);
    printf("Reader %d finishing reading \n", id);
    sem_wait(&mutex);
    rc--;
    if (rc == 0) {
        sem_post(&mutex);
    } return NULL;
}
```

```
void * writer (void * arg) {
    int id = ((int) arg);
    sem_wait(&db);
    printf("writer %d is writing\n", id);
    sleep(2);
    printf("writer %d finishing writing\n", id);
    sem_post(&db);
    return NULL;
}

int main () {
    pthread_t r[5], w[2];
    int reader_ids[5] = {1,2,3,4,5};
    int writer_ids[2] = {1,2};
    sem_init(&mutex, 0, 1);
    sem_init(&db, 0, 1);
    for (int i=0; i<5; i++) {
        pthread_create(&p[i], NULL, reader,
        &reader_ids[i]);
    }
    for (int i=0; i<2; i++) {
        pthread_create(&w[i], NULL, writer,
            &writer_ids[i]);
    }
    for (int i=0; i<5; i++) {
        pthread_join(r[i], NULL);
    }
```

```
for( int i=0; i<2; i++){
    pthread_join(w[i], NULL);
}
sem_destroy(&mutex);
sem_destroy(&db);
return 0;
}
```

Output :

```
Reader 5065/0404 is reading
Reader 5065/0400 is reading
Reader 5065/0408 finished reading
Reader 5065/0404 finished reading
Reader 5065/0412 is reading
Reader 5065/0400 is reading
Reader 5065/0400 finished reading
Reader 5065/0412 finished reading
writer 5065/0376 is writing
writer 5065/0376 finished writing
writer 5065/0380 is writing
writer 5065/0380 finished writing
Reader 5065/0416 is reading.
Reader 5065/0416 finished reading
```

13

Experiment - 8

Implementation of Banker's Algorithm

```
# include <stdio.h>
# include <stdbool.h>
# define MAX 10
int main(){
    int n, m, i, j, k;
    int alloc[MAX][MAX], max[MAX][MAX],
        avail[MAX];
    int need[MAX][MAX], finish[MAX],
        safeSeq[MAX];
    printf("Enter number of processes : ");
    scanf("%d", &n);
    printf("Enter number of resources : ");
    scanf("%d", &m);

    printf("Enter Allocation matrix : \n");
    for(i=0; i<n; i++){
        for(j=0; j<m; j++){
            scanf("%d", &alloc[i][j]);
        }
    }
    printf("Enter maximum matrix : \n");
    for(i=0; i<n; i++){
        for(j=0; j<m; j++){
```

```
scanf("%d", &max[i][j]);
printf("Enter available resources \n");
for (i=0 ; i<m ; i++){
    scanf("%d", &avail[i]);
}
for (i=0; i<n ; i++){
    for (j=0; j<m; j++){
        need[i][j] = max[i][j] - alloc[i][j];
}}
int count =0;
for (int i=0; i<n; i++){
    finish[i]=0;
}
while ( count <n){
    bool found = false;
    for (i=0; i<n; i++){
        if (finish[i]==0){
            for (j=0; j<m; j++){
                if (need[i][j] > avail[j],
                    break;
            if (j==m){
                for (k=0; k<m; k++){
                    avail[k] += alloc[i][k];
                }
                safeSeq [count++] = i;
                finish[i] = 1;
                found = true;
}}}
```

ITM-25K
13

```
if (!found) {
        printf("System is not in a safe state\n");
        return 0;
    }
}
printf("System is in a safe state.\n Safe
        sequence is : ");
for (i=0; i<n; i++) {
    printf("P%d", safeseq[i]);
}
printf("\n");
} return 0;
```

Output :-

```
Enter number of processes : 3
Enter number of resources : 3
Enter Allocation Matrix:
0 1 0
2 0 0
3 0 2
Enter Maximum Matrix:
7 5 3
3 2 2
9 0 2
Enter Available Resources:
3 3 2
System is in safe state.
Safe Sequence is : P0 P1 P2
```

Experiment - 9

Implementation of resource allocation graph (RAG).

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 10
int graph[MAX][MAX];
bool visited[MAX], recStack[MAX];
int n;
bool isCyclic(int v){
    if(!visited[v]){
        visited[v] = true;
        recStack[v] = true;
        for(int i=0; i<n; i++){
            if(graph[v][i]){
                if(!visited[i] && isCyclic(i))
                    return true;
                else if(recStack[i])
                    return true;
            }
        }
        recStack[v] = false;
        return false;
    }
```

```
int main(){
    int p, r;
    printf("Enter number of processes : ");
    scanf("%d", &p);
    printf("Enter number of resources : ");
    scanf("%d", &r);

    n = p + r;
    printf("Enter edges (format : from to), -1, -1 to
            stop : \n");
    printf("Process = 0 to %d, Resources = %d to %d\n",
            p-1, p, p+r-1);
    int from, to;
    while (1){
        scanf("%d %d", &from, &to);
        if (from == -1 && to == -1)
            break;
        graph[from][to] = 1;
    }
for (int i=0; i<n; i++){
    visited[i] = rec_stack[i] = false;
}
for (int i=0 ; i<n ; i++){
    if (isCyclic[i]){
        printf("Deadlock detected in the
system. \n");
        return 0;
    }
}
```

```
printf("No deadlock detected.\n");
return 0;
}
```

Output :

```
Enter number of processes : 2
Enter number of resources : 2
Enter edges (format : from to), -1 -1 to stop :
0 2
2 0
1 3
3 1
-1 -1
```

Deadlock detected in the system.

Experiment -10

Implementation of contiguous allocation techniques : Best - fit.

```
#include <stdio.h>
#define MAX 10
int main(){
  int blockSize[MAX], processSize[MAX];
  int blockCount, processCount, allocation[MAX];
  printf("Enter number of memory
          blocks :");
  scanf("%d", &blockCount);
  printf("Enter size of each block : \n");
  for(int i=0; i< blockCount; i++){
      scanf("%d", &blockSize[i]);
  }
  printf("Enter Number of processes :");
  scanf("%d", &processCount);
  printf("Enter size of each process : \n");
  for(int i=0; i< processCount; i++){
      scanf("%d", &processSize[i]);
  for(int i=0; i< processCount ; i++){
      allocation[i] = -1;
  }
```

```
for (int i=0; i< processCount ; i++){
    int bestIdx = -1;
    for (int j=0 ; j< blockCount ; j++){
        if (blockSize[j] >= processSize[i]){
            if (bestIdx == -1 || blockSize[j] <
                blockSize[bestIdx])
                    bestIdx = j;
            }
        }
        if (bestIdx != -1){
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }
    printf("\n Process No. \t Process Size \t Block No. \n");
    for (int i=0; i< processCount ; i++){
        printf("%d \t %d \t \t", i+1, processSize[i]);
        if (allocation[i] != -1){
            printf("%d \n", allocation[i]+1);
        }
        else{  printf(" Not Allocated \n");
        }
    }
    return 0;
}
```

Output :-

Enter number of memory blocks : 5
100 500 200 300 600
Enter number of process : 4
212 417 112 426

Process No.	Process Size	Block No
1	212	4
2	417	5
3	112	3
4	426	Not allocated.