1.File management calls

```
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<stdio.h>

int main()
{
        int n,fd;
        char buff[50];    // declaring buffer

        //message printing on the display
        printf("Enter text to write in the file:\n");
        //read from keyboard, specifying 0 as fd for std input device
        //Here, n stores the number of characters
        n= read(0, buff, 50);

        // creating a new file using open.
        fd=open("file",O_CREAT | O_RDWR, 0777);

        //writting input data to file (fd)
        write(fd, buff, n);
        //Write to display (1 is standard fd for output device)
        write(1, buff, n);

        //closing the file
        int close(int fd);

        return 0;}
```

FCFS CPU scheduling

```
#include <stdio.h>
int main()
{
    int pid[15];
    int bt[15];
    int n;
    printf("Enter the number of processes: ");
    scanf("%d",&n);

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

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

    int i, wt[n];
    wt[0]=0;

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

    printf("Process ID     Burst Time     Waiting Time     TurnAround Time\n");
    float twt=0.0;
    float tat= 0.0;
    for(i=0; i<n; i++)
    {
        printf("%d\t\t", pid[i]);
        printf("%d\t\t", bt[i]);
        printf("%d\t\t", wt[i]);

        //calculating and printing turnaround time of each process
        printf("%d\t\t", bt[i]+wt[i]);
        printf("\n");

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

        //for calculating total turnaround time
        tat += (wt[i]+bt[i]);
    }
    float att,awt;

    //for calculating average waiting time
    awt = twt/n;

    //for calculating average turnaround time
    att = tat/n;
    printf("Avg. waiting time= %f\n",awt);
```

```
    printf("Avg. turnaround time= %f",att);
}
```

Shortest job first

```
#include<stdio.h>
 int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
    float avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

    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
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }
    temp=bt[i];
    bt[i]=bt[pos];
    bt[pos]=temp;

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

wt[0]=0;


for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        wt[i]+=bt[j];

    total+=wt[i];
}

avg_wt=(float)total/n;
total=0;

printf("\nProcess   Burst Time    Waiting Time Turnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i];
```

```
        total+=tat[i];
        printf("\np%d         %d            %d               %d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=(float)total/n;
    printf("\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f",avg_tat);
}
```

PRIORITY BASED

```
#include<stdio.h>
 // structure representing a structure
struct priority_scheduling {

  // name of the process
  char process_name;

  // time required for execution
  int burst_time;

  // waiting time of a process
  int waiting_time;

  // total time of execution
  int turn_around_time;

  // priority of the process
  int priority;
};
```

```
int main() {

  // total number of processes
  int number_of_process;

  // total waiting and turnaround time
  int total = 0;

  // temporary structure for swapping
  struct priority_scheduling temp_process;

  // ASCII numbers are used to represent the name of the process
  int ASCII_number = 65;

  // swapping position
  int position;

  // average waiting time of the process
  float average_waiting_time;

  // average turnaround time of the process
  float average_turnaround_time;

  printf("Enter the total number of Processes: ");
  // get the total number of the process as input
  scanf("%d", & number_of_process);

  // initializing the structure array
  struct priority_scheduling process[number_of_process];

  printf("\nPlease Enter the  Burst Time and Priority of each process:\n");
```

```
// get burst time and priority of all process
for (int i = 0; i < number_of_process; i++) {

  // assign names consecutively using ASCII number
  process[i].process_name = (char) ASCII_number;

  printf("\nEnter the details of the process %c \n", process[i].process_name);
  printf("Enter the burst time: ");
  scanf("%d", & process[i].burst_time);

  printf("Enter the priority: ");
  scanf("%d", & process[i].priority);

  // increment the ASCII number to get the next alphabet
  ASCII_number++;

}

// swap process according to high priority
for (int i = 0; i < number_of_process; i++) {

  position = i;

  for (int j = i + 1; j < number_of_process; j++) {

    // check if priority is higher for swapping
    if (process[j].priority > process[position].priority)
      position = j;
  }
  // swapping of lower priority process with the higher priority process
  temp_process = process[i];
  process[i] = process[position];
  process[position] = temp_process;
}
// First process will not have to wait and hence has a waiting time of 0
process[0].waiting_time = 0;

for (int i = 1; i < number_of_process; i++) {
  process[i].waiting_time = 0;
  for (int j = 0; j < i; j++) {
    // calculate waiting time
    process[i].waiting_time += process[j].burst_time;
  }

  // calculate total waiting time
  total += process[i].waiting_time;
}

// calculate average waiting time
average_waiting_time = (float) total / (float) number_of_process;

// assigning total as 0 for next calculations
total = 0;

printf("\n\nProcess_name \t Burst Time \t Waiting Time \t  Turnaround Time\n");
printf("------------------------------------------------------\n");

for (int i = 0; i < number_of_process; i++) {

  // calculating the turn around time of the processes
  process[i].turn_around_time = process[i].burst_time + process[i].waiting_time;
```

```
    // calculating the total turnaround time.

    total += process[i].turn_around_time;


    // printing all the values

    printf("\t  %c \t\t  %d \t\t %d \t\t %d", process[i].process_name, process[i].burst_time, process[i].waiting_time, process[i].turn_around_time);

    printf("\n----------------------------------------------------------\n");

  }


  // calculating the average turn_around time

  average_turnaround_time = (float) total / (float) number_of_process;


  // average waiting time

  printf("\n\n Average Waiting Time : %f", average_waiting_time);


  // average turnaround time

  printf("\n Average Turnaround Time: %f\n", average_turnaround_time);


  return 0;

}
```

Round Robin

```
#include<stdio.h>

#include<conio.h>


void main()

{

    // initlialize the variable name

    int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

    float avg_wt, avg_tat;
    printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
    y = NOP; // Assign the number of process to variable y

// Use for loop to enter the details of the process like Arrival time and the Burst Time
for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
printf(" Arrival time is: \t");  // Accept arrival time
scanf("%d", &at[i]);
printf(" \nBurst time is: \t"); // Accept the Burst time
scanf("%d", &bt[i]);
temp[i] = bt[i]; // store the burst time in temp array
}
// Accept the Time qunat
printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
// Display the process No, burst time, Turn Around Time and the waiting time
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
for(sum=0, i = 0; y!=0; )
{
if(temp[i] <= quant && temp[i] > 0) // define the conditions
{
    sum = sum + temp[i];
    temp[i] = 0;
    count=1;
    }
    else if(temp[i] > 0)
    {
        temp[i] = temp[i] - quant;
        sum = sum + quant;
    }
    if(temp[i]==0 && count==1)
    {
        y--; //decrement the process no.
        printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
        wt = wt+sum-at[i]-bt[i];
        tat = tat+sum-at[i];
        count =0;
    }
    if(i==NOP-1)
    {
        i=0;
    }
    else if(at[i+1]<=sum)
    {
        i++;
    }
    else
    {
        i=0;
    }
}
// represents the average waiting time and Turn Around time
avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
getch();
}
```

Producer consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <unistd.h>
#define N 5
#define BUFSIZE 1
#define PERMS 0666
int *buffer;
int nextp = 0, nextc = 0;
int mutex, full, empty; // semaphores varialbes
void producer()
{
 int data;
 if(nextp == N)
 nextp = 0;
 printf("Enter data for producer to produce: ");
 scanf("%d",(buffer+nextp));
 nextp++;
}
void consumer()
{
 int g;
 if(nextc == N)
 nextc = 0;
 g = *(buffer+nextc++);
 printf("\nConsumer Consumes data %d",g);
```

```
}
void sem_op(int id, int value)
{
 struct sembuf op;
 int v;
 op.sem_num = 0;
 op.sem_op = value;
 op.sem_flg = SEM_UNDO;
 if((v = semop(id,&op,1))<0)
 {
 printf("\nError executing semop instruction");
 }
}
void sem_create(int semid, int initval)
{
 int semval;
 union semun
 {
 int val;
 struct semid_ds *buf;
 unsigned short *array;
 }s;
 s.val = initval;
 if((semval = semctl(semid, 0, SETVAL, s))<0)
 {
 printf("\nError in executing semctl");
 }
}
void sem_wait(int id)
{
 int value = -1;
 sem_op(id, value);
}
void sem_signal(int id)
{
 int value = 1;
 sem_op(id, value);
}
void main()
{
 int shmid, i;
 pid_t pid;
 if((shmid = shmget(1000,BUFSIZE, IPC_CREAT|PERMS))<0)
 {
 printf("\nUnable to create shared memory");
 return;
 }
 if((buffer = (int*)shmat(shmid,(char*)0,0)) == (int*)-1)
 {
 printf("\nShared memory allocation error");
 exit(1);
 }
 if((mutex = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
 {
 printf("\nCan't create mutex semaphore");
 exit(1);
 }
 if((empty = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
 {
 printf("\nCan't create empty semaphore");
 exit(1);
 }
```

```
if((full = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
{
printf("\nCan't create full semaphore");
exit(1);
}

sem_create(mutex,1);
sem_create(empty,N);
sem_create(full,0);

if((pid =fork())<0)
{
printf("\nError in process Creation");
exit(1);
}
else if(pid>0)
{
for( i = 0 ;i< N; i++)
{
sem_wait(empty);
sem_wait(mutex);
producer();
sem_signal(mutex);
sem_signal(full);
}
}
else if(pid == 0)
{
for( i = 0;i<N; i++ )
{
sem_wait(full);
```

```
 sem_wait(mutex);
 consumer();
 sem_signal(mutex);
 sem_signal(empty);
 }
 }
 printf("\n");
}
```

Banker’s Algorithm

```
#include <stdio.h>
int main()
{
 // P0, P1, P2, P3, P4 are the Process names here

 int n, m, i, j, k;
 n = 5; // Number of processes
 m = 3; // Number of resources
 int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
 { 2, 0, 0 }, // P1
 { 3, 0, 2 }, // P2
 { 2, 1, 1 }, // P3
 { 0, 0, 2 } }; // P4

 int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
 { 3, 2, 2 }, // P1
 { 9, 0, 2 }, // P2
 { 2, 2, 2 }, // P3
 { 4, 3, 3 } }; // P4
```

```
int avail[3] = { 3, 3, 2 }; // Available Resources

int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
for (j = 0; j < m; j++)
need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
for (i = 0; i < n; i++) {
if (f[i] == 0) {

int flag = 0;
for (j = 0; j < m; j++) {
if (need[i][j] > avail[j]){
flag = 1;
break;
}
}
if (flag == 0) {
ans[ind++] = i;
for (y = 0; y < m; y++)
avail[y] += alloc[i][y];
f[i] = 1;
}
}
}
```

```
 }
 int flag = 1;

 for(int i=0;i<n;i++)
 {
 if(f[i]==0)
 {
 flag=0;
 printf("The following system is not safe");
 break;
 }
 }
 if(flag==1)
 {
 printf("Following is the SAFE Sequence\n");
 for (i = 0; i < n - 1; i++)
 printf(" P%d ->", ans[i]);
 printf(" P%d", ans[n - 1]);
 }
 return (0);
}
```

First Fit

```
// C implementation of First - Fit algorithm
#include<stdio.h>

// Function to allocate memory to
// blocks as per First fit algorithm
void firstFit(int blockSize[], int m, int processSize[], int n)
{
```

```
int i, j;
// Stores block id of the
// block allocated to a process
int allocation[n];

// Initially no block is assigned to any process
for(i = 0; i < n; i++)
{
        allocation[i] = -1;
}

// pick each process and find suitable blocks
// according to its size ad assign to it
for (i = 0; i < n; i++)     //here, n -> number of processes
{
        for (j = 0; j < m; j++)     //here, m -> number of blocks
        {
                if (blockSize[j] >= processSize[i])
                {
                        // allocating block j to the ith process
                        allocation[i] = j;

                        // Reduce available memory in this block.
                        blockSize[j] -= processSize[i];

                        break; //go to the next process in the queue
                }
        }
}
```

```
    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++)
    {
        printf(" %i\t\t\t", i+1);
        printf("%i\t\t\t\t", processSize[i]);
        if (allocation[i] != -1)
            printf("%i", allocation[i] + 1);
        else
            printf("Not Allocated");
        printf("\n");
    }
}

// Driver code
int main()
{
    int m; //number of blocks in the memory
    int n; //number of processes in the input queue
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    m = sizeof(blockSize) / sizeof(blockSize[0]);
    n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0 ;
}
```

FIRST FIT

```
#include <stdio.h>
#include <conio.h>
#define max 25
void main()
{
 int frag[max], b[max], f[max], i, j, nb, nf, temp;
 static int bf[max], ff[max];
 printf("\n\tMemory Management Scheme - First Fit");
 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:\tFragement");
 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]);
 getch();
}
```

BEST FIT

```
#include <stdio.h>
#include <conio.h>
#define max 25
void 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]);
 getch();
}
```

INPUT: Enter the number of blocks: 3 Enter the number of files: 2 Enter the size of the blocks:- Block 1: 5 Block 2: 2 Block 3: 7 Enter the size of the files:- File 1: 1 File 2: 4 OUTPUT: File No File Size Block No Block Size Fragment 1 1 2 2 1 2 4 1 5 1

WORST FIT

```
#include <stdio.h>
#include <conio.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("\n\tMemory Management Scheme - Worst Fit");
 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:\tFragement");
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]);
 getch();
}
```

Enter the number of blocks: 3 Enter the number of files: 2 Enter the size of the blocks:- Block 1: 5 Block 2: 2 Block 3: 7 Enter the size of the files:- File 1: 1 File 2: 4 OUTPUT