National   Institute Of Technology Karnatka,Surathkal

DEPARTMENT OF INFORMATION TECHNOLOGY

OPERATING SYSTEM LAB DOCUMENTATION

COURSE CODE: IT 250

COURSE INSTUCTOR : Dr. Biju R Mohan

Submitted By: Dashan jot Singh
Roll No :- 16IT216

Signature

LAB 1(FIXED SIZE AND VARIABLE SIZE PARTITIONING)

Aim:

To implement a simulation of Fixed size and variable Size memory allocation system in C.

Description:

1. **Fixed Size Memory Allocation**:

Fixed Size Memory Allocation,uses a free list of fixed-size blocks of memory.

2. **Variable Size Memory Allocation**:

Variable Size Memory Allocation,uses a free list of Variable-size blocks of memory.

There are three types of memory allocation-:

1.First FIT:

In the first fit,partition is allocated which is first sufficient from the top of the Main memory.Means when we find first block of memory which is sufficent for a process we use that block for process.

2.Best Fit:

Allocate the process to the partition which is first smallest sufficient partition among the free available partition.

3.Worst Fit:

Allocate the process to the partition which is largest sufficient among the freely available partitions available in the main memory.

Code :

program1.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void BestFITmem(int m,int block[],int n,int process[]){
  int allocation[n]; //declaring a array for allocation status
  int arblock[m];//blocka is for the memory
  for(int x=0;x<m;x++){
    arblock[x]=block[x];
  }
  int arprocess[n];
  for(int u=0;u<n;u++){
    arprocess[u]=process[u];
  }
  memset(allocation,-1,sizeof(allocation));
  for(int i=0;i<n;i++){
    int best=-1; //best fit is basically find a block with min size >=rquired size
    for(int j=0;j<m;j++){
      if(arblock[j]>=arprocess[i]){
        if(best==-1){
          best=j;
        }
        else if(arblock[best]>arblock[j]){
          best=j;
        }
      }
    }
    if(best!=-1){
      allocation[i]=best;
```

```
      arblock[best]=arblock[best]-arprocess[i];
    }
  }
  printf("BESTFIt₩n");
  printf("₩n process No. ₩t Process Size₩t Block No.₩n");
  for(int r=0;r<n;r++){
    printf(" %d₩t₩t%d₩t₩t",r+1,arprocess[r] );
    if(allocation[r]!=-1){
      printf("%d",allocation[r]+1);
    }
    else{
      printf("Not allocated");
    }
    printf("₩n");
  }
}
void FirstFitmem(int m,int block[],int n,int process[]){
  int allocation[n];
  int block1[m];
  for(int y=0;y<m;y++){
    block1[y]=block[y];
  }
  int process1[n];
  for(int c=0;c<n;c++){
    process1[c]=process[c];
  }
  memset(allocation,-1,sizeof(allocation));
//whichever having size >=process size comes first allocate it
  for(int i=0;i<n;i++){
    for(int j=0;j<m;j++){
      if(block1[j]>=process1[i]){
        allocation[i]=j;
        block1[j]-=process1[i];
        break;
      }
    }
  }
  printf("FIRSTFIT₩n");
  printf("₩n process No. ₩t Process Size₩t Block No.₩n");
  for(int r=0;r<n;r++){
    printf(" %d₩t₩t%d₩t₩t",r+1,process1[r] );
    if(allocation[r]!=-1){
      printf("%d",allocation[r]+1);
    }
    else{
      printf("Not allocated");
    }
    printf("₩n");
  }
}
void WORSTFITmem(int m,int block[],int n,int process[]){
  int allocation[n];
  int copyblock[m];
  int copyprocess[n];
//find a bigger block for any process
  for(int i=0;i<m;i++){
    copyblock[i]=block[i];
  }
  for(int j=0;j<n;j++){
    copyprocess[j]=block[j];
  }
  memset(allocation,-1,sizeof(allocation));
  for(int i=0;i<n;i++){
    int worst=-1;
    for(int j=0;j<m;j++){
      if(copyblock[j]>=copyprocess[i]){
        if(worst==-1){
          worst=j;
        }
        else if(copyblock[worst]<copyblock[j]){
```

```
          worst=j;
        }
      }
    }
    if(worst!=-1){
      allocation[i]=worst;
      copyblock[worst]=copyblock[worst]-copyprocess[i];
    }

}
  printf("₩n process No. ₩t Process Size₩t Block No.₩n");
  for(int r=0;r<n;r++){
    printf(" %d₩t₩t%d₩t₩t",r+1,copyprocess[r] );
    if(allocation[r]!=-1){
      printf("%d",allocation[r]+1);
    }
    else{
      printf("Not allocated");
    }
    printf("₩n");
  }
}
int main(){
  int memblock;
  printf("Enter the no of memory blocks₩n");
  scanf("%d",&memblock);
  int memw[memblock];
  int i;
  printf("Enter the sizes of the memory %d blocks ₩n",memblock);
  for(i=0;i<memblock;i++){
    scanf("%d",&memw[i]);
  }
  int proce;
  printf("Enter the no of processes₩n");
  scanf("%d",&proce);
  int process[proce];
  int j;
  for(j=0;j<proce;j++){
    scanf("%d",&process[j]);
  }
  BestFITmem(memblock,memw,proce,process);
  WORSTFITmem(memblock,memw,proce,process);
  FirstFitmem(memblock,memw,proce,process);
  return 0;
}
```

Output:

Enter the no of memory blocks
5
Enter the sizes of the memory 5 blocks
200
500
300
20
100
Enter the no of processes
5
100
300
20
500
80
BESTFIt

process No.	Process Size	Block No.
1	100	5
2	300	3
3	20	4
4	500	2
5	80	1

process No.	Process Size	Block No.
1	200	2
2	500	Not allocated
3	300	2
4	20	3
5	100	3

FIRSTFIT

process No.	Process Size	Block No.
1	100	1
2	300	2
3	20	1
4	500	Not allocated
5	80	1

2.program2.c

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
struct Node{ //creating a linked list node
  int size;//size of the node or memory
  bool allocated;//allocation status
  int processId;//process id that is allocated in this block
  struct Node *next;//next pointer for next node
};
void FirstFit(struct Node** head,int prosize,int pid){
printf("FirstFit₩n");
//which blocks first acommplish the requirement of memory allocate that
struct Node* temp=*head;
struct Node* newnode=NULL;
//while(temp->next!=NULL){
while(temp!=NULL){
  if((temp->size>=prosize)&&(temp->allocated!=true)){
    newnode=(struct Node*)malloc(sizeof(struct Node));
    newnode->next=temp->next;
    int siz=(temp->size)-prosize;
    newnode->size=siz;
    temp->size=prosize;
    temp->allocated=true;
    temp->next=newnode;
    temp->processId=pid;
    break;
    return;
  }
  temp=temp->next;
}

//printf("!!!!!!Error available space is not sufficient₩n");

}
void BestFit(struct Node** head,int prosize,int pid){
printf("BestFit₩n");
struct Node* temp=*head;
struct Node* prev=NULL;
int min=temp->size;
int flag=0;
while(temp!=NULL){
  if((min>temp->size)&&(min>=prosize)&&temp->allocated!=true){
  min=temp->size;
  }
  prev=temp;
  temp=temp->next;
}
temp=prev;
struct Node* newnode=NULL;
if(temp!=NULL){
  newnode=(struct Node*)malloc(sizeof(struct Node));
  newnode->next=temp->next;
```

```
  int siz=(temp->size)-prosize;
  newnode->size=siz;
  temp->size=prosize;
  temp->allocated=true;
  temp->next=newnode;
  temp->processId=pid;
  flag=1;


}
if(flag==0){
  printf("!!!!!!Error available space is not sufficient₩n");
}
}
void WorstFit(struct Node** head ,int prosize,int pid){
struct Node* temp=*head;
printf("WorstFit₩n");
int max=temp->size;
int flag=0;
struct Node* prev=NULL;
while(temp!=NULL){
  if((max<temp->size)&&(max>=prosize)&&(temp->allocated!=true)){
    max=temp->size;
  }
  prev=temp;
  temp=temp->next;
}
temp=prev;
struct Node* newnode=NULL;
if(temp!=NULL){
  newnode=(struct Node*)malloc(sizeof(struct Node));
  newnode->next=temp->next;
  int siz=(temp->size)-prosize;
  newnode->size=siz;
  temp->size=prosize;
  temp->allocated=true;
  temp->next=newnode;
  temp->processId=pid;
  flag=1;

}
if(flag==0){
  printf("!!!!!!Error available space is not sufficient₩n");
}
}
void printList(struct Node** root)
{
  struct Node* node=*root;
  while (node != NULL)
  {
     printf(" Process ID =%d ₩t Process Size=%d₩t₩n₩n", node->processId,node->size);
     node = node->next;
  }
}
void delete(struct Node** head,int pid){
printf("delete₩n");
struct Node* temp=*head;
int flag=0;
printf("process id %d₩n",pid);
while(temp!=NULL){
  if(temp->processId==pid){
    temp->processId=0;
    temp->allocated=false;
    flag=1;
  }
  temp=temp->next;
}
if(flag==0){
  printf("Error Not found₩n");
}
```

```
else{
  printf("process deleted successfully₩n");
}
}
int main(){
  struct Node* head_ref=NULL;
  struct Node* head_ref1=NULL;
  struct Node* head_ref2=NULL;
  printf("enter the total available space memory₩n");
  int memory_size;
  scanf("%d",&memory_size);
  head_ref=(struct Node*)malloc(sizeof(struct Node));
  head_ref->size=memory_size;
  head_ref->allocated=false;
  head_ref->next=NULL;
  head_ref1=(struct Node*)malloc(sizeof(struct Node));
  head_ref1->size=memory_size;
  head_ref1->allocated=false;
  head_ref1->next=NULL;
  head_ref2=(struct Node*)malloc(sizeof(struct Node));
  head_ref2->size=memory_size;
  head_ref2->allocated=false;
  head_ref2->next=NULL;
  int choice=0;

  while(choice!=3){
    printf("Enter your choice ₩n1.Add process₩n2.Delete process₩n3.exit₩n4.printList₩n");
    scanf("%d",&choice);
    switch (choice) {
      case 1:{
        int process;
        int pid;
        printf("enter process size ₩n");
        scanf("%d",&process);
        printf("Enter process ID₩n");
        scanf("%d",&pid);
        FirstFit(&head_ref,process,pid);
        BestFit(&head_ref1,process,pid);
        WorstFit(&head_ref2,process,pid);
        break;
      }
      case 2:{
        printf("Enter process Id to delete process₩n");
        int pid;
        scanf("%d",&pid);
        delete(&head_ref,pid);
        delete(&head_ref1,pid);
        delete(&head_ref2,pid);
        break;
      }
      case 3:{
        printf("exiting.........₩n");
        exit(0);
        break;
      }
      case 4:{
        struct Node* temp=head_ref;
        printf("*******************FIRST FIT ALGORITHM₩n");
        printList(&head_ref);

        printf("************************BEST FIT ****************************₩n");
        printList(&head_ref1);

        printf("******************** WORST FIT******************₩n");
        printList(&head_ref2);
              break;
      }
      default:{
        printf("ERROR₩n");
        exit(0);
```

```
        break;
      }
    }
  }
  return 0;
}
```

Output:

```
enter the total available space memory
4
Enter your choice
1.Add process
2.Delete process
3.exit
4.printList
1
enter process size
2
Enter process ID
1
FirstFit
BestFit
WorstFit
Enter your choice
1.Add process
2.Delete process
3.exit
4.printList
1
enter process size
2
Enter process ID
2
FirstFit
BestFit
WorstFit
Enter your choice
1.Add process
2.Delete process
3.exit
4.printList
4
*******************FIRST FIT ALGORITHM
 Process ID =1      Process Size=2

 Process ID =2      Process Size=2

 Process ID =0      Process Size=0

************************BEST FIT ***************************
 Process ID =1      Process Size=2

 Process ID =2      Process Size=2

 Process ID =0      Process Size=0

******************** WORST FIT*****************
 Process ID =1      Process Size=2

 Process ID =2      Process Size=2

 Process ID =0      Process Size=0

Enter your choice
1.Add process
2.Delete process
```

3.exit
4.printList
2
Enter process Id to delete process
2
delete
process id 2
process deleted successfully
delete
process id 2
process deleted successfully
delete
process id 2
process deleted successfully
Enter your choice
1.Add process
2.Delete process
3.exit
4.printList
4
*******************FIRST FIT ALGORITHM
 Process ID =1   Process Size=2

 Process ID =0   Process Size=2

 Process ID =0   Process Size=0
************************BEST FIT ***************************
 Process ID =1   Process Size=2

 Process ID =0   Process Size=2

 Process ID =0   Process Size=0
******************** WORST FIT******************
 Process ID =1   Process Size=2

 Process ID =0   Process Size=2

 Process ID =0   Process Size=0

Enter your choice
1.Add process
2.Delete process
3.exit
4.printList
3
exiting.........

Lab 2(Paging and Page table)

Aim:

To implement a simulation of paging memory allocation system using linked lists and page table in java.

Description:

A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the accessing process, while physical addresses are used by the hardware, or more specifically, by the RAM subsystem.

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non - contiguous.

Code:
Exercise1.java

```
import java.io.*;
import java.util.*;

class Process{ //create a class process with an attribute size (process size)
  int size;
  Process(int size){
    this.size=size;
  }
}
class Paging{
  public static void main(String args[]){
  int n;
  System.out.println("no os frames");
  Scanner s = new Scanner(System.in);
  n=s.nextInt();
  int[] a = new int[n]; //memory partions
  System.out.println("");
  int Memory; //memory of the each partition
  Memory=s.nextInt();
  for(int i=0;i<n;i++){
    a[i]=Memory;
  }

  System.out.println("Enter the pagesize that you want to set");
  int page;
  page=s.nextInt();
//page size which is related to the process partitioning
  System.out.println("Enter the number of processes that you wanna allocate");
  int Process_no; //total no of processes
  Process_no=s.nextInt();
  LinkedList[] process_ar=new LinkedList[Process_no];
  int[] Pro2 = new int[Process_no];
  System.out.println("Enter the size of each process");
  for(int i=0;i<Process_no;i++){
    int temp=s.nextInt();
    int partition =(int)(temp/page);
    LinkedList l = new LinkedList(); //create a linked list
    for(int j=0;j<partition;j++){
      Process o = new Process(page); //add process to linked list
      l.add(o);
      }

    int memfinal=0;
    memfinal = temp-(page*partition);
    if(memfinal!=0){
    Process o = new Process(memfinal);
    l.add(o);
    }
    Pro2[i]=temp;
    process_ar[i]=l;
  }


    LinkedList[] allocated = new LinkedList[Process_no];
    for(int j=0;j<Process_no;j++){
      for(int k=0;k<n;k++){
        if(a[k]>=Pro2[j]){
          a[k]-=Pro2[j];
          Pro2[j]=0;
          break;
        }
        else{
          int count=0;
          int srt=0;
          count=(int)(Pro2[j]/page);
```

```
        srt=Pro2[j]-(count*page);
        if(process_ar[j].size()>count){
          while(process_ar[j].size()!=1){
            if(k<n){
              if(a[k]>=page){
                a[k]-=page;
                process_ar[j].pop();
              }
              else{
                k+=1;
              }
            }
            else{
              break;
            }
          }
          if(k<n){
            if(a[k]>=srt){
              a[k]-=srt;
              process_ar[j].pop();
              break;
            }
            else{
              k=k+1;
              if(k<n){
                a[k]-=srt;
                process_ar[j].pop();
                break;
              }
              else{
                break;
              }
            }  }
        }
        else if(process_ar[j].size()==count){
          while(process_ar[j].size()!=0){
            if(k<n){
              if(a[k]>=page){
                a[k]-=page;
                process_ar[j].pop();
              }
              else{
                k+=1;
              }
            }
            else{
              break;
            }
          }
  }

      }
    }
  }

  for(int w=0;w<Process_no;w++){
    if(Pro2[w]==0 || process_ar[w].size()==0){
      System.out.println("Process "+ (w+1) + " is allocated");
    }
    else if(process_ar[w].size()!=0){
      System.out.println("Process "+ (w+1)+ "  number of pages whose allocation still left are " +
process_ar[w].size());
    }
  }
 }
}
```

Output:

```
no os frames
4
Memory of Each partition
2
Enter the pagesize that you want to set
1
Enter the number of processes that you wanna allocate
5
Enter the size of each process
1
2
1
4
1
Process 1 is allocated
Process 2 is allocated
Process 3 is allocated
Process 4  number of pages whose allocation still left are 3
Process 5  number of pages whose allocation still left are 1
```

2. Exercise2.java

```java
import java.util.HashMap;
import java.lang.Math.*;

class Page {
  String pageId;
public Page(String pageId) { this.pageId = pageId; }
}

class Frame {
  Page page = null;
  int num;

  public Frame(int num) { this.num = num; }

  boolean isAllocated() { return page != null; }
  void assign(Page page) { this.page = page; }
}

class Process {
  Page[] pages;
  int size;
  String name;
  boolean alloc;

  HashMap<Page, Frame> pageTable = new HashMap<>(); // Table for pages in memory frames

  public int PAGE_SIZE = 100;

  public Process(int size, String name) {
    this.size = size;
    this.name = name;
    int numFrames = (int) Math.ceil((float)size / PAGE_SIZE);
    pages = new Page[numFrames];
    for (int i=0; i<numFrames; i++) {
      pages[i] = new Page(name+ "" +i);
    }
  }

  void execute() {
    if (alloc)
      for(Page page: pages) {
        System.out.println(page.pageId + " executing from frame " + pageTable.get(page).num);
      }
    else
      System.out.println("Process not in memory");
  }

  Page[] getPages() {
    return this.pages;
```

```
    }
}

class Memory {
    Frame[] frames;

    public Memory(int numFrames) {
        frames = new Frame[numFrames];
        for (int i=0; i<numFrames; i++) {
            frames[i] = new Frame(i);
        }
    }

    void allocateMemory(Process process) {
        Page[] pages = process.getPages();
        int alloc = 0;
        for (Page page: pages) {
            for (Frame frame: frames) {
                if(!frame.isAllocated()) {
                    frame.assign(page);
                    process.pageTable.put(page, frame);
                    alloc++;
                    break;
        }
    }
        }
        if (alloc != pages.length) {
            System.out.println("Only " + alloc + " Pages allocated.");
        }
        process.alloc = true;
    }

    void killProcess(Process process) {
        for(Page page: process.getPages()) {
            process.pageTable.get(page).page = null;
        }
        System.out.println("Process " + process.name + " killed.");
        process.alloc = false;
    }

    void displayFrames() {
        for (Frame frame: frames) {
            if (frame.isAllocated())
                System.out.println("Frame " + frame.num + ": Page " + frame.page.pageId);
            else
                System.out.println("Frame " + frame.num + ": empty");
        }
    }
}

class PagingTable {

    public static void main(String[] args) {
        Memory mem = new Memory(10);
        Process p1 = new Process(300, "Process 1");
        mem.allocateMemory(p1);
        Process p2 = new Process(300, "Process 2");
        mem.allocateMemory(p2);
        Process p3 = new Process(300, "Process 3");
        mem.allocateMemory(p3);
        mem.displayFrames();
        mem.killProcess(p2);
        mem.displayFrames();
        p2.execute();
        p1.execute();
        Process p4 = new Process(400, "Process 4");
        mem.allocateMemory(p4);
        mem.displayFrames();
    }
}
```

Output:

Frame 0: Page Process 10
Frame 1: Page Process 11
Frame 2: Page Process 12
Frame 3: Page Process 20
Frame 4: Page Process 21
Frame 5: Page Process 22
Frame 6: Page Process 30
Frame 7: Page Process 31
Frame 8: Page Process 32
Frame 9: empty
Process Process 2 killed.
Frame 0: Page Process 10
Frame 1: Page Process 11
Frame 2: Page Process 12
Frame 3: empty
Frame 4: empty
Frame 5: empty
Frame 6: Page Process 30
Frame 7: Page Process 31
Frame 8: Page Process 32
Frame 9: empty
Process not in memory
Process 10 executing from frame 0
Process 11 executing from frame 1
Process 12 executing from frame 2
Frame 0: Page Process 10
Frame 1: Page Process 11
Frame 2: Page Process 12
Frame 3: Page Process 40
Frame 4: Page Process 41
Frame 5: Page Process 42
Frame 6: Page Process 30
Frame 7: Page Process 31
Frame 8: Page Process 32
Frame 9: Page Process 43

Lab 3(Defragmentation)

Aim:

To Implement defragmentation(to remove the internal fragmentation join the free blocks) in Variable Sized Contiguous Memory Allocation.

Description:

In the maintenance of file systems **defragmentation** is a process that reduces the amount oS Fragmentaion . It does this by physically organizing the contents of the Mass storage device used to store files into the smallest number of contiguous regions (fragments). It also attempts to create larger regions of free space using compaction to impede the return of fragmentation. Some defragmentation utilities try to keep smaller files within a single directory together, as they are often accessed in sequence.

In contiguous memory allocation, all the empty blocks in memory are brought together and all the allocated blocks are shifted to form a contiguous block, this is called defragmentation.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
//defragmentation is basically removing the fragment in variable size //partitioning
struct Node{//create a linked list node
  int size;
  bool allocated;
  int processId;
  struct Node *next;
};
void FirstFit(struct Node** head,int prosize,int pid){
printf("FirstFit₩n");
struct Node* temp=*head;
struct Node* newnode=NULL;
//while(temp->next!=NULL){
while(temp!=NULL){
  if((temp->size>=prosize)&&(temp->allocated!=true)){
    newnode=(struct Node*)malloc(sizeof(struct Node));
    newnode->next=temp->next;
    int siz=(temp->size)-prosize;
    newnode->size=siz;
    temp->size=prosize;
    temp->allocated=true;
    temp->next=newnode;
    temp->processId=pid;
    break;
    return;
  }
  temp=temp->next;
}

//printf("!!!!!!Error available space is not sufficient₩n");

}
void BestFit(struct Node** head,int prosize,int pid){
printf("BestFit₩n");
struct Node* temp=*head;
struct Node* prev=NULL;
int min=temp->size;
int flag=0;
while(temp!=NULL){
  if((min>temp->size)&&(min>=prosize)&&temp->allocated!=true){
  min=temp->size;
  }
  prev=temp;
  temp=temp->next;
}
temp=prev;
struct Node* newnode=NULL;
if(temp!=NULL){
  newnode=(struct Node*)malloc(sizeof(struct Node));
  newnode->next=temp->next;
  int siz=(temp->size)-prosize;
  newnode->size=siz;
  temp->size=prosize;
  temp->allocated=true;
  temp->next=newnode;
  temp->processId=pid;
```

```
  flag=1;


}
if(flag==0){
  printf("!!!!!!Error available space is not sufficient₩n");
}
}
void WorstFit(struct Node** head ,int prosize,int pid){
struct Node* temp=*head;
printf("WorstFit₩n");
int max=temp->size;
int flag=0;
struct Node* prev=NULL;
while(temp!=NULL){
  if((max<temp->size)&&(max>=prosize)&&(temp->allocated!=true)){
    max=temp->size;
  }
  prev=temp;
  temp=temp->next;
}
temp=prev;
struct Node* newnode=NULL;
if(temp!=NULL){
  newnode=(struct Node*)malloc(sizeof(struct Node));
  newnode->next=temp->next;
  int siz=(temp->size)-prosize;
  newnode->size=siz;
  temp->size=prosize;
  temp->allocated=true;
  temp->next=newnode;
  temp->processId=pid;
  flag=1;

}
if(flag==0){
  printf("!!!!!!Error available space is not sufficient₩n");
}
}
void printList(struct Node** root)
{
  struct Node* node=*root;
  while (node != NULL)
  {
     printf(" Process ID =%d ₩t Process Size=%d₩t₩n₩n", node->processId,node->size);
     node = node->next;
  }
}
void delete(struct Node** head,int pid){
//while deleting the process we move that node of memory to the last and merge with the free memory
struct Node* temp=*head;
struct Node* tmp=*head;
int blankspace=0;//blankspace is space of that block which is going to free
int flag=0;
if (temp->processId==pid){//if that block is head then make head to the next block
  blankspace+=tmp->size;
  *head=tmp->next;
  flag=1;
}
while(temp->next!=NULL){
//otherwise find the process in linked list
  if(temp->next->processId==pid){
    blankspace=temp->next->size;
    temp->next=temp->next->next;
    temp->next->allocated=false;
    flag=1;
  }
  temp=temp->next;
}
if(flag==1){//check if flag=1 that means process is found and deleted successfully
```

```
  temp->size+=blankspace;

}
else{
  printf("Error process not found₩n");
}
}
int main(){
  struct Node* head_ref=NULL;
  struct Node* head_ref1=NULL;
  struct Node* head_ref2=NULL;
  printf("enter the total available space memory₩n");
  int memory_size;
  scanf("%d",&memory_size);
  head_ref=(struct Node*)malloc(sizeof(struct Node));//first linked list for first fit
  head_ref->size=memory_size;
  head_ref->allocated=false;
  head_ref->next=NULL;
  head_ref1=(struct Node*)malloc(sizeof(struct Node));//second linked list for best fit
  head_ref1->size=memory_size;
  head_ref1->allocated=false;
  head_ref1->next=NULL;
  head_ref2=(struct Node*)malloc(sizeof(struct Node));//third linked list for worst fit
  head_ref2->size=memory_size;
  head_ref2->allocated=false;
  head_ref2->next=NULL;
  int choice=0;

  while(choice!=3){
    printf("Enter your choice ₩n1.Add process₩n2.Delete process₩n3.exit₩n4.printList₩n");
    scanf("%d",&choice);
    printf("###################################################₩n");
    switch (choice) {
      case 1:{
        int process;
        int pid;
        printf("enter process size ₩n");
        scanf("%d",&process);
        printf("Enter process ID₩n");
        scanf("%d",&pid);
        FirstFit(&head_ref,process,pid);
        BestFit(&head_ref1,process,pid);
        WorstFit(&head_ref2,process,pid);
        break;
      }
      case 2:{
        printf("Enter process Id to delete process₩n");
        int pid;
        scanf("%d",&pid);
        delete(&head_ref,pid);
        delete(&head_ref1,pid);
        delete(&head_ref2,pid);
        break;
      }
      case 3:{
        printf("exiting.........₩n");
        exit(0);
        break;
      }
      case 4:{
        struct Node* temp=head_ref;
        printf("********************FIRST FIT ALGORITHM₩n");
        printList(&head_ref);
        printf("************************BEST FIT *****************************₩n");
        printList(&head_ref1);
        printf("********************* WORST FIT*******************₩n");
        printList(&head_ref2);
      }
      default:{
        printf("ERROR₩n");
```

```
        exit(0);
        break;
      }
    }
  }
  return 0;
}
```

output:

```
enter the total available space memory
10
Enter your choice
1.Add process
2.Delete process
3.exit
4.printList
1
##################################################
enter process size
2
Enter process ID
1
###############################################################################
FirstFit
BestFit
WorstFit
Enter your choice
1.Add process
2.Delete process
3.exit
4.printList

1
enter process size
3
Enter process ID
2
FirstFit
BestFit
WorstFit
Enter your choice
1.Add process
2.Delete process
3.exit
4.printList
4
******************FIRST FIT ALGORITHM
 Process ID =1      Process Size=2

 Process ID =2      Process Size=3

 Process ID =0      Process Size=5

***********************BEST FIT **************************
 Process ID =1      Process Size=2

 Process ID =2      Process Size=3

 Process ID =0      Process Size=5

******************* WORST FIT*****************
 Process ID =1      Process Size=2

 Process ID =2      Process Size=3

 Process ID =0      Process Size=5
```

Lab 4(Buddy Memory Allocation)

Aim:

To implement buddy memory allocation system.

Description:

In a buddy system, the allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence (see below for example), such that any block except the smallest can be divided into two smaller blocks of permitted sizes.

When the allocator receives a request for memory, it rounds the requested size up to a permitted size, and returns the first block from that size's free list. If the free list for that size is empty, the allocator splits a block from a larger size and returns one of the pieces, adding the other to the appropriate free list.

When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size . To make this easier, the free lists may be stored in order of address. The main advantage of the buddy system is that coalescence is cheap because the "buddy" of any free block can be calculated from its address.

Buddy system memory management algorithm:

Assume the memory size is 2U, suppose a size of S is required.

- If 2U-1<S<=2U: Allocate the whole block
- Else: Recursively divide the block equally and test the condition at each time, when it satisfies, alloacate the block and get out the loop.

CODE://
Buddy.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<stdbool.h>
struct BuddyNode{//create a buddy tree node
  int size;//size of the node(memory size)
  int RIsallocated;//Right child allocation status
  int LIsallocated;//left child allocation status
  int processId;//process id of the stored process
  int processSize;//size of the process which is stored
  struct BuddyNode *left,*right;//pointer to the right and left child
};
struct BuddyNode* new_node(int val){//for any node instead of creating a node and assigning values each time use this function
//this will create a node and intialize the values to it
  struct BuddyNode* new_node=(struct BuddyNode*)malloc(sizeof(struct BuddyNode));
```

```
  new_node->right=NULL;
  new_node->left=NULL;
  new_node->size=val;
  new_node->processSize=0;
  new_node->processId=0;
  new_node->RIsallocated=0;
  new_node->LIsallocated=0;
}
int delete(struct BuddyNode** roo,int processID){
//for deleteing any process from the buddy tree
  int res;
  struct BuddyNode* root=*roo;//maintain a pointer to the root
  if(root->left==NULL&&root->right==NULL){//process should be in leaf nodes only
    if(root->processId==processID){
      printf("found₩n");
      root->RIsallocated=0;//reset all the values as intail values of a node
      root->LIsallocated=0;
      root->processSize=0;
      root->processId=0;
      return 1;//return 1 that is for successfully deleted
    }
    else{
      return 0;
    }
  }
  res=delete(&root->left,processID);//otherwise find at the left part of the tree
  if(res==1){//if found then check if the buddy means nearest same level node is also free if free then delete those
nodes
  // root->left=NULL;
    if(root->right->LIsallocated==0&&root->right->RIsallocated==0&&root->right->left==NULL&&root->right-
>right==NULL){
      root->LIsallocated=0;
      root->RIsallocated=0;
      root->right=NULL;
      root->left=NULL;
      return 1;//return 1 for successful deletion
    }
  }
  res=delete(&root->right,processID);//if not found in the left part find in right part
  if(res==1){//if found then check if its brother node (buddy node ) is free or not if free then delete both
  //  root->right=NULL;
    if(root->left->LIsallocated==0&&root->left->RIsallocated==0&&root->left->left==NULL&&root->left-
>right==NULL){
      root->LIsallocated=0;
      root->RIsallocated=0;
      root->right=NULL;
      root->left=NULL;
      return 1;
    }}
  return 0;
}
int BuddyAllocation(struct BuddyNode** root,int process,int processId,int actualSize){
  struct BuddyNode* temp=*root;//assign temp to root of the buddy tree
  int flag;
  struct BuddyNode* new=NULL;//new node to null
//actaul size is nearest power of two of the process size
  if(temp->size<actualSize){// if the available space is less than the required one then error
    printf("Insufficient space Sorry!!!₩n");
    return 0;
  }
  else if(temp->size==actualSize){
    if((temp->left==NULL)&&(temp->right==NULL)){//the processess are only allocted at the leafs of the buddy tree
      temp->size=actualSize;
      temp->processId=processId;
      temp->processSize=process;
      temp->RIsallocated=1;
      temp->LIsallocated=1;
      return 1;//return 1 for successfull insertion
    }
    else{
```

```
      return 0;
    }
  }
  else{//if not leaf size
    if(temp->LIsallocated==0){//if left child is not allocated
        if(temp->left==NULL){//if left child is not there
          temp->left=new_node((int)(temp->size/2));//create left and right nodes of size parentsize/2
          temp->right=new_node((int)(temp->size/2));
          flag=BuddyAllocation(&temp->left,process,processId,actualSize);//and make a recursive call on left part of the
newly created part
          if(temp->left->LIsallocated==1 && temp->left->RIsallocated==1){//if left part's left and right child is allocated
then left child allocation status is also one
            temp->LIsallocated=1;
          }
          if(flag==1){
            return 1;
          }
        }
        else{//if left part is there
          flag=BuddyAllocation(&temp->left,process,processId,actualSize);//insert in left part of tree
          if(temp->left->LIsallocated==1&&temp->left->RIsallocated==1){
            temp->LIsallocated=1;
          }
          if(flag==1){
            return 1;
          } }
    }
    if(temp->RIsallocated==0){//if right part is not allocated
      flag=BuddyAllocation(&temp->right,process,processId,actualSize);//insert at the right of the tree
      if(temp->right->LIsallocated==1 && temp->right->RIsallocated==1){
        temp->RIsallocated=1;
      }
      if(flag==1){
        return 1;
      } }
  return 0;
}
}
void printtree(struct BuddyNode *root){
  if(root->left==NULL&&root->right==NULL){
    printf("%d₩t%d₩t₩t%d₩t₩t%d₩t%d₩n",root->size,root->processSize,root->processId,root-
>LIsallocated,root->RIsallocated);
  }
  else{
    printtree(root->left);
    printtree(root->right);
  }
}
int main(){
  int memorysize;
  printf("enter the size of the main memory(in power of two )₩n");
  scanf("%d",&memorysize);
  int flag1=0;
  //int processId=0;
  struct BuddyNode *root=new_node(memorysize);
  int choice=0;
  while(choice!=4){
    printf("Enter your choice₩n1.Add process₩n2.Delete process₩n3.print processess₩n4.exit₩n₩n");
    scanf("%d",&choice);
    switch (choice) {
      case 1:{
        int processSize;
        int processId;
        printf("Enter process Size₩n");
        scanf("%d",&processSize);
       printf("Enter process ID₩n");
        scanf("%d",&processId);
        int actualSize;
        int x=1;
        int nearestpower=0;
```

```
        while(x<processSize){
          x=x*2;
          nearestpower+=1;
        }
        actualSize=x;
        printf("the actual size is %d₩n",actualSize);
        flag1=BuddyAllocation(&root,processSize,processId,actualSize);
        if(flag1==1){
          processId+=1;
          printf("SIZE₩tProcessSIZE₩tPROCESSID₩tLEFT₩tRIGHT₩t₩n");
          printtree(root);

        }
        else{
          printf("Insufficent space₩n");
        }
        break;
      }
      case 2:{
        printf("Enter process Id to delete the process₩n");
        int pid;
        scanf("%d",&pid);
        delete(&root,pid);
        break;
      }
      case 3:{
        printf("printing the tree with buddy₩n");
        printtree(root);
        break;
      }
      case 4:{
        printf("Exiting ...........₩n");
        exit(0);
        break;
      }
      default:{
        printf("Error Invalid choice₩n");
        //exit(0);
        break;
      } }
  }

  return 0;
}
```

Output:

```
enter the size of the main memory(in power of two )
64
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

1
##################################################
Enter process Size
32
Enter process ID
1
the actual size is 32
SIZE    ProcessSIZE     PROCESSID       LEFT    RIGHT
32      32              1               1       1
32      0               0               0       0
Enter your choice
1.Add process
2.Delete process
3.print processess
```

```
4.exit

1
###################################################
Enter process Size
8
Enter process ID
2
the actual size is 8
SIZE    ProcessSIZE    PROCESSID    LEFT    RIGHT
32      32             1            1       1
8       8              2            1       1
8       0              0            0       0
16      0              0            0       0
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

1
###################################################
Enter process Size
3
Enter process ID
3
the actual size is 4
SIZE    ProcessSIZE    PROCESSID    LEFT    RIGHT
32      32             1            1       1
8       8              2            1       1
4       3              3            1       1
4       0              0            0       0
16      0              0            0       0
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

1
###################################################
Enter process Size
2
Enter process ID
4
the actual size is 2
SIZE    ProcessSIZE    PROCESSID    LEFT    RIGHT
32      32             1            1       1
8       8              2            1       1
4       3              3            1       1
2       2              4            1       1
2       0              0            0       0
16      0              0            0       0
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

1
###################################################
Enter process Size
1
Enter process ID
5
the actual size is 1
SIZE    ProcessSIZE    PROCESSID    LEFT    RIGHT
32      32             1            1       1
8       8              2            1       1
4       3              3            1       1
```

```
2       2               4               1       1
1       1               5               1       1
1       0               0               0       0
16      0               0               0       0
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

1
####################################################
Enter process Size
1
Enter process ID
6
the actual size is 1
SIZE    ProcessSIZE     PROCESSID       LEFT    RIGHT
32      32              1               1       1
8       8               2               1       1
4       3               3               1       1
2       2               4               1       1
1       1               5               1       1
1       1               6               1       1
16      0               0               0       0
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

2
####################################################
Enter process Id to delete the process
2
found
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

2
####################################################
Enter process Id to delete the process
5
found
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

2
####################################################
Enter process Id to delete the process
6
found
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

3
####################################################
printing the tree with buddy
32      32              1               1       1
8       0               0               0       0
4       3               3               1       1
```

```
2    2    4    1    1
2    0    0    0    0
16   0    0    0    0
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

2
###################################################
Enter process Id to delete the process
3
found
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

2
###################################################
Enter process Id to delete the process
4
found
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit

3
printing the tree with buddy
32   32   1    1    1
32   0    0    0    0
Enter your choice
1.Add process
2.Delete process
3.print processess
4.exit
```

LAB 5

Aim:

differenciate i/o and cpu bound process by two programs

Description:

CPU Bound means the rate at which process progresses is limited by the speed of the CPU. A task that performs calculations on a small set of numbers, for example multiplying small matrices, is likely to be CPU bound.

I/O Bound means the rate at which a process progresses is limited by the speed of the I/O subsystem. A task that processes data from disk, for example, counting the number of lines in a file is likely to be I/O bound.

Code: CpuBound Program : // file.c

```
#include <stdio.h>
int main()
{
   char ch;  /* Pointer for both the file*/
   FILE *fpr, *fpw;
```

```
   /* Opening file FILE1.C in “r” mode for reading */
   fpr = fopen("C:₩₩file1.txt", "r");
   /* Ensure FILE1.C opened successfully*/
   if (fpr == NULL)
   {   puts("Input file cannot be opened");
   }
   /* Opening file FILE2.C in “w” mode for writing*/
   fpw= fopen("C:₩₩file2.txt", "w");
   /* Ensure FILE2.C opened successfully*/
   if (fpw == NULL)
   {
      puts("Output file cannot be opened");
   }
    /*Read & Write Logic*/
   while(1)
   {
       ch = fgetc(fpr);
       if (ch==EOF)
           break;
       else
           fputc(ch, fpw);
   }
   /* Closing both the files */
   fclose(fpr);
   fclose(fpw);
   return 0;
}
```

PrimeNumber.c

cpu bound process

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  long long int n;
//for calcualting the prime numbers we using alu unit of the cpu
//so that cpu usage when this program runs for a big input is large
  printf("Enter the value of the n:₩n");
  scanf("%lld",&n);
  printf("prime numbers up to n₩n");
  int *prime;
  prime=(long long int *)malloc(n*(sizeof(long long int)));
  for(int i=0;i<=n;i++){
    prime[i]=1;
  }
    for (long long int p=2; p*p<=n; p++)
   {
      // If prime[p] is not changed, then it is a prime
      if (prime[p] == 1)
      {
          // Update all multiples of p
```

```
            for (long long int i=p*2; i<=n; i += p)
                prime[i] = 0;
        }
    }

    // Print all prime numbers
    for (long long int p=2; p<=n; p++)
       if (prime[p]==1)
          printf("%lld  ",p);
    printf("₩n");

   return 0;
}
```

Process Sheduling Algorithms

FCFS

Aim:

to simulate the FCFS(First come, First Served) Algorithm for Process Sheduling

Description :

First come, first served (FCFS) is an operating system process scheduling algorithm and a network routing management mechanism that automatically executes queued requests and processes by the order of their arrival. With first come, first served, what comes first is handled first; the next request in line will be executed once the one before it is complete.

Code:

FCFS.c

```
#include<stdio.h>
int main(){
        int arr[100],burst[100],ta[100],compl[100],wt[100];
```

```
int i,j,d,n;
float awt,ata;
printf("Enter no. of process: ");
scanf("%d",&n);
printf("Enter their Arrival Time and Burst Time₩n");
for (i=0;i<n;i++){
printf("process-%d :",i+1);
scanf("%d%d",&arr[i],&burst[i]);}
compl[0]=arr[0]+burst[0];
ta[0]=burst[0];
wt[0]=0;
awt=0;
ata=ta[0];
for (i=1;i<n;i++){
if (arr[i]>compl[i-1]){
compl[i]=burst[i]+arr[i];
}
else{
compl[i]=burst[i]+compl[i-1];
}
ta[i]=compl[i]-arr[i];
ata+=ta[i];
wt[i]=ta[i]-burst[i];
awt+=wt[i];
}
printf("*************************************₩n");
printf("FIRST COME FIRST SERVE SCHEDULING₩n");
printf("*************************************₩n");
printf("ProcessNo.₩tArrival₩t₩tBurstTime₩tCompletion₩tTurnAround₩tWaiting₩n");
for (i=0;i<n;i++){
printf("%d₩t₩t%d₩t₩t%d₩t₩t%d₩t₩t%d₩t₩t%d₩n",i+1,arr[i],burst[i],compl[i],ta[i],wt[i]);
}
printf("Average Waiting Time: %f₩n",awt/n);
printf("Average TurnAround Time: %f₩n",ata/n);

}
```

output:

```
Enter no. of process: 5
Enter their Arrival Time and Burst Time
process-1 :0 5
process-2 :3 6
process-3 :12 6
process-4 :10 4
process-5 :1 5
*************************************
FIRST COME FIRST SERVE SCHEDULING
*************************************
```

ProcessNo.	Arrival	BurstTime	Completion	TurnAround	Waiting
1	0	5	5	5	0
2	3	6	11	8	2
3	12	6	18	6	0
4	10	4	22	12	8
5	1	5	27	26	21

Average Waiting Time: 6.200000
Average TurnAround Time: 11.400000

SJF(Shortest job first)

Aim:

to simulate sjf (shortest job first)algorithm for process sheduling

Description:

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

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

Code: sjf.c

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

int main(){
  int Arrival_time[100];
  int Burst_time[100];
  int TurnAround_time[100]={0};
  int completion_time[100];
  int waiting_time[100];
  int tSequence[100];
  int count;
  int no_of_process;
  int min;
  int i,j;
  int pointer;
  float total_waiting_time;
  float total_turn_around;
  printf("Enter the no of process ₩n");
  scanf("%d",&no_of_process);
  printf("Enter Arrival time and Burst time of the Processess₩n");
  for(int i=0;i<no_of_process;i++){
    printf("Process - %d  :",i+1);
```

```
 scanf("%d%d",&Arrival_time[i],&Burst_time[i]);
}
pointer=Arrival_time[0];//intilize the pointer to the first process
min=0;
for(int i1=0;i1<no_of_process && Arrival_time[i1]==pointer;i1++){ //find the min burst time process till that arrival time
  if(Burst_time[min]>Burst_time[i1]){
    min=i1; //update the index when we find minimum
  }
}
tSequence[min]=1; //minimum Burst_time process shound be first
completion_time[min]=Arrival_time[min]+Burst_time[min];
TurnAround_time[min]=completion_time[min]-Arrival_time[min];
 //calculate the completion_time and TurnAround_time for the process which is having min burst time
waiting_time[min]=0; //waiting time of the min process is zero because thatis executing first
total_waiting_time=0;
total_turn_around=TurnAround_time[min];
count=no_of_process-1; //one process is gone for execution
 j=min;
//loop till all the processees is executed
while(count>0){
        for (i=0;i<no_of_process;i++){
                if (TurnAround_time[i]==0){
                        min=i;
                        while(Arrival_time[i]==Arrival_time[min]){
                                if (Burst_time[min]>Burst_time[i] && TurnAround_time[i]==0) {
                                        min=i;
                                }
                                i++;
                        }
                break;}
        }
        for (i=0;(i<no_of_process && completion_time[j]>= Arrival_time[i]);i++){
                if (Burst_time[min]>Burst_time[i] && TurnAround_time[i]==0)
                {
                        min=i;
                }
        }
        tSequence[min]=no_of_process-count+1;
        completion_time[min]=Burst_time[min]+completion_time[j];
        if (Arrival_time[min]>completion_time[j])
```

```
                    completion_time[min]=Burst_time[min]+Arrival_time[min];
                j=min;
                TurnAround_time[min]=completion_time[min]-Arrival_time[min];
                total_turn_around+=TurnAround_time[min];
                waiting_time[min]=TurnAround_time[min]-Burst_time[min];
                total_waiting_time+=waiting_time[min];
                count-=1;
        }
   printf("*****************************************₩n");
        printf("SORTEST JOB FIRST SCHEDULING(NON-PREEMPTIVE)₩n");
        printf("*****************************************₩n");
        printf("ProcessNo.₩tArrival₩t₩tBurstTime₩tCompletion₩tTurnAround₩tWaiting₩t₩tSequence₩n");
        for (int i5=0;i5<no_of_process;i5++){
        printf("%d₩t₩t%d₩t₩t%d₩t₩t%d₩t₩t%d₩t₩t%d₩t₩t%d₩
n",i5+1,Arrival_time[i5],Burst_time[i5],completion_time[i5],TurnAround_time[i5],waiting_time[i5],tSequence[i5]);
        }
        printf("Average Waiting Time: %f₩n",total_waiting_time/no_of_process);
        printf("Average TurnAround Time: %f₩n",total_turn_around/no_of_process);
  return 0;
}
```

output:

Enter the no of process

5

Enter Arrival time and Burst time of the Processess

Process - 1 :0 3

Process - 2 :0 6

Process - 3 :4 9

Process - 4 :4 2

Process - 5 :9 1

****

SORTEST JOB FIRST SCHEDULING(NON-PREEMPTIVE)

****

ProcessNo.	Arrival	BurstTime	Completion	TurnAround	Waiting	Sequence
1	0	3	3	3	0	1
2	0	6	9	9	3	2
3	4	9	21	17	8	5
4	4	2	12	8	6	4
5	9	1	10	1	0	3

Average Waiting Time: 3.400000

Average TurnAround Time: 7.600000

Round-Robin Sheduling

Aim:

to simulate the Round Robin Algorithm for process Sheduling

Description:

Round-robin (RR) is one of the algorithms employed by process and network shedulers in computing .As the term is generally used, time quantum (also known as time quanta)are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation -free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks.

Code:

roundrobin.c

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


int head_of_queue=0;
int tail_of_queue=0;
int process_queue[100];


int main(){
        int arrival_time[100];
        int Burst_time[100];
        int TurnAround_time[100]={0};
        int completion_time[100];
        int waiting_time[100];
        int dummy_list[100];
        int no_of_process;
        int minimum;
        int pointer;
        int size;
        int current_time; //curren time
        int time_quantum;
```

```
	float total_turn_around,total_waiting_time;
	printf("Enter time Quantum ₩n");
	scanf("%d",&time_quantum);
	printf("Enter no of Processess₩n");
	scanf("%d",&no_of_process);
	int count=0;
	printf("Enter Process Arrival time and Burst time₩n");
	for(int i=0;i<no_of_process;i++){
		printf("Process - %d  :",i+1);
		scanf("%d%d",&arrival_time[i],&Burst_time[i]);
		dummy_list[i]=Burst_time[i]; //creates a copy of the Burst time list
	}
	size=1; //intilize the size first to one
	current_time=arrival_time[0]; //intilize the current_time to first arrival time
	head_of_queue=0; //intilize the head of the queue to be the first element
	tail_of_queue=0; //queue tail is also the fisrt element when only one element in the queue
	process_queue[0]=0; //initally only first process is in the process queue
	while(count<no_of_process)//while count is not equal to the total process
	{
		pointer=head_of_queue; //initally the pointer is on the head of the process queue
		while(1){ //move pointer to the next element everytime we found a new uncomplete process
			if(dummy_list[pointer]>0){ //some process is still requre time to complete
				break; //we found
			}
			//if dumm[pointer]==0 means the process is completed
			pointer=(pointer+1)%no_of_process;//if we donot find a process move to next
		}
		if(dummy_list[pointer]<=time_quantum){//if remaining time of a process to complete is less than time quantum
			//then this is last itertaion on that process
			current_time+=dummy_list[pointer]; //update the current time by the remaing burst time of the process
			dummy_list[pointer]=0;//the process is completed
			completion_time[pointer]=current_time; //update the completion_time of the process
			TurnAround_time[pointer]=completion_time[pointer]-arrival_time[pointer];
			//calculate the turn around time
			waiting_time[pointer]=TurnAround_time[pointer]-Burst_time[pointer];
			total_turn_around+=TurnAround_time[pointer];
			total_waiting_time+=waiting_time[pointer];
			count+=1; //update the count as the process completed
		}
```

```
			else{
				current_time+=time_quantum;
				dummy_list[pointer]-=time_quantum; //decrese the remaing_time of a process by the time quantum
			}
			// add next process which come by currnt time  to the process queue
			for(int j=0;j<no_of_process && arrival_time[j]<=current_time;j++){
				process_queue[size]=j;
				size+=1; //update the size  of the process as the new process is added
			}
			if(count==size){ //means the all processess is in the queue
				current_time=arrival_time[size];
				size+=1;
				process_queue[size]=size;
			}
			head_of_queue=(head_of_queue+1)%size;
			tail_of_queue=(tail_of_queue+1)%size;
		}
		printf("*****************************************₩n");
		printf("ROUND-ROBIN SCHEDULING(NON-PREEMPTIVE)₩n");
		printf("ProcessNo.₩tArrival₩t₩tBurstTime₩tCompletion₩tTurnAround₩tWaiting₩n");
		for (int i1=0;i1<no_of_process;i1++){
		printf("%d₩t₩t%d₩t₩t%d₩t₩t%d₩t₩t%d₩t₩t%d₩n",i1+1,arrival_time[i1],Burst_time[i1],completion_time[i1],TurnAround_time[i1],waiting_time[i1]);
		}
		printf("Average Waiting Time: %f₩n",total_waiting_time/no_of_process);
		printf("Average TurnAround Time: %f₩n",total_turn_around/no_of_process);
		return 0;
}
```

output:

Enter time Quantum

2

Enter no of Processess

4

Enter Process Arrival time and Burst time

Process - 1 :0 2

Process - 2 :0 4

Process - 3 :2 3

Process - 4 :5 6

****

ROUND-ROBIN SCHEDULING(NON-PREEMPTIVE)

ProcessNo.	Arrival	BurstTime	Completion	TurnAround	Waiting
1	0	2	2	2	0
2	0	4	10	10	6
3	2	3	17	15	12
4	5	6	19	14	8

Average Waiting Time: 6.500000

Average TurnAround Time: 10.250000

Function Pointer

Aim:

simulate funcation pointer in c

code:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
int compare(const void *a,const void *b){
  return (*(int *)a-*(int *)b);
}
int main(){
  int n;
  printf("Enter the value of n₩n");
  scanf("%d",&n);
  int arr[n];
  printf("Enter the array elements ₩n");
```

```
  for(int i=0;i<n;i++){
    scanf("%d",&arr[i]);
  }
  qsort(arr,n,sizeof(int),compare);
  for(int i=0;i<n;i++){
    printf("%d ",arr[i]);
  }
  printf("₩n");
  return 0;
}
```

copyCommand in c

Aim:

to simulate the copycommand in c

Code:

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

int main(int argc,char* argv[]){

  FILE *source ,*target;

  //declaring file pointer for source file and target file

  char ch;

  /*if aguments are less than three throw an error*/
  if(argc!=3){
```

```
    printf("Command Error !! Insufficent arguments Given₩n");

    return 0;
  }
  source=fopen(argv[1],"r");

  //open the source file in read mode
  target=fopen(argv[2],"w");

  //open the target file in write mode
  if(source==NULL||target==NULL){

    printf("Unable to open . Error while opening file₩n");

    return 0;
  }
  while((ch=fgetc(source))!=EOF){
  fputc(ch,target);//writing to the target FILE
}
  printf("Copy is successful₩n");
  fclose(source);//closing the source file
  fclose(target);//closing the target file

  return 0;
}
```

Java RMI

Aim:

simulate java rmi

Code:

CalCulaterInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CalCulaterInterface extends Remote{
  /*The Remote interface serves to identify interfaces
  whose methods may be invoked from a non-local virtual machine.
  Any object that is a remote object must directly or indirectly
  implement this interface. Only those methods specified in
  a "remote interface", an interface that extends java.rmi.Remote
  are available remotely.
  */
  public int add(int a,int b) throws RemoteException;
  public int substarct(int a,int b) throws RemoteException;
  public int multiply(int a,int b) throws RemoteException;
  public double divide(int a,int b) throws RemoteException;
}
```

ImplementCal.java

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;
```

```
import java.rmi.RemoteException;


public class ImplementCal extends UnicastRemoteObject implements CalCulaterInterface{
/*Used for exporting a remote object with JRMP
and obtaining a stub that communicates to the remote object.
*/
  public ImplementCal() throws RemoteException{

  }
  public int add(int a,int b){
    return a+b;
  }
  public int substarct(int a,int b){
    return a-b;
  }
  public int multiply(int a,int b){
    return a*b;
  }
  public double divide(int a,int b){
    return a/b;
  }
}
```

CalServer.java

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class CalServer{
  public static void main(String args[]){
    /*
    Registry is a remote interface to a simple remote
     object registry that provides methods for storing
     and retrieving remote object references bound with arbitrary string names.
      The bind, unbind, and rebind methods are used to alter the name bindings
      in the registry, and the lookup and list methods are used to query
      the current name bindings.
    */
    try{
      Registry reg=LocateRegistry.createRegistry(1099);
      ImplementCal c=new ImplementCal();
      reg.rebind("mycalc",c);
      System.out.println("Server is Ready ...........");
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

CalClinet.java

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;
```

```java
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class CalServer{
  public static void main(String args[]){
    /*
    Registry is a remote interface to a simple remote
     object registry that provides methods for storing
     and retrieving remote object references bound with arbitrary string names.
      The bind, unbind, and rebind methods are used to alter the name bindings
      in the registry, and the lookup and list methods are used to query
      the current name bindings.
    */
     try{
      Registry reg=LocateRegistry.createRegistry(1099);
      ImplementCal c=new ImplementCal();
      reg.rebind("mycalc",c);
      System.out.println("Server is Ready ...........");
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

Priority Sheduling

Aim:

to simulate the priority sheduling algorithm for process sheduling

Description:

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with the highest priority is to be executed first and so on. Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Code:

```cpp
#include<bits/stdc++.h>
using namespace std;

struct Process
{
	int pid; // Process ID
	int bt; // CPU Burst time required
	int priority; // Priority of this process
};

// Function to sort the Process acc. to priority
bool comparison(Process a, Process b)
{
```

```
	return (a.priority > b.priority);
}

// Function to find the waiting time for all
// processes
void findWaitingTime(Process proc[], int n,
					int wt[])
{
	// waiting time for first process is 0
	wt[0] = 0;

	// calculating waiting time
	for (int i = 1; i < n ; i++ )
		wt[i] = proc[i-1].bt + wt[i-1] ;
}

// Function to calculate turn around time
void findTurnAroundTime( Process proc[], int n,
						int wt[], int tat[])
{
	// calculating turnaround time by adding
	// bt[i] + wt[i]
	for (int i = 0; i < n ; i++)
		tat[i] = proc[i].bt + wt[i];
}

//Function to calculate average time
void findavgTime(Process proc[], int n)
{
	int wt[n], tat[n], total_wt = 0, total_tat = 0;

	//Function to find waiting time of all processes
	findWaitingTime(proc, n, wt);

	//Function to find turn around time for all processes
	findTurnAroundTime(proc, n, wt, tat);

	//Display processes along with all details
	 printf("₩nProcesses	Burst time	Waiting time	Turn around time₩n");

	// Calculate total waiting time and total turn
	// around time
	for (int i=0; i<n; i++)
	{
		total_wt = total_wt + wt[i];
		total_tat = total_tat + tat[i];
		printf(  "%d₩t %d₩t₩t %d₩t₩t %d₩t₩t₩n", proc[i].pid,proc[i].bt,wt[i],tat[i]);
	}

	printf("₩nAverage waiting time =%lf₩n",(float)total_wt / (float)n);
	printf("₩nAverage turn around time = %lf₩n",(float)total_tat / (float)n);
}

void priorityScheduling(Process proc[], int n)
{
	// Sort processes by priority
	sort(proc, proc + n, comparison);
```

```
	cout<< "Order in which processes gets executed ₩n";
	for (int i = 0 ; i < n; i++)
		printf("%d   ",proc[i].pid);

	findavgTime(proc, n);
}

// Driver code
int main()
{
	int n;
	printf("enter the no of processess₩n");
	scanf("%d",&n);
	Process proc[n];
	for(int i=0;i<n;i++){
	   proc[i].pid=i+1;
	   printf("Enter provority and Burst time₩n");
	   scanf("%d%d",&process[i]->priority,&process[i]->bt);
	}
	priorityScheduling(proc, n);
	return 0;
}
```

Output:

```
Order in which processes gets executed
1 3 2

Processes  Burst time  Waiting time  Turn around time

 1       10     0        10

 3       8     10        18

 2       5     18        23



Average waiting time = 9.33333

Average turn around time = 17
```