CHAPTER 1

INTRODUCTION

1.1 Introduction

This project describes work that most efficient way of assigning memory to process to avoid wastage of memory and in terms of time complexity. Memory management is the process of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status. Memory management allocation consists of two techniques, namely: single contiguous allocation and multiple contiguous allocations. In single contiguous allocation, single allocation is the simplest memory management technique. All the exception of small portion reserved for the operating system, is available to the single application. MS-DOS is an example of a system which allocates memory in this way. A system using single contiguous allocation may still multitask by swapping the contents of memory to switch among users. Early versions of music operating used this technique. In partitioned allocation, it divides primary memory into multiple memory partitions, usually contiguous areas of memory. Each partition might contain all the information for a specific job or task. Memory management consist of allocating a partition to a job when it starts and un-allocating it when the job ends. Partitioned allocation usually requires some hardware support to prevent the jobs from interfering with one another or with the operating system.

1.2 Memory allocation techniques

There are 4 dynamic memory allocation techniques

- First Fit
- Best Fit
- Worst Fit
- Next Fit

1.2.1 First Fit: In the first fit approach process is allocated to the first free partition which can accommodate the process. It finishes the execution after finding the first suitable free partition.

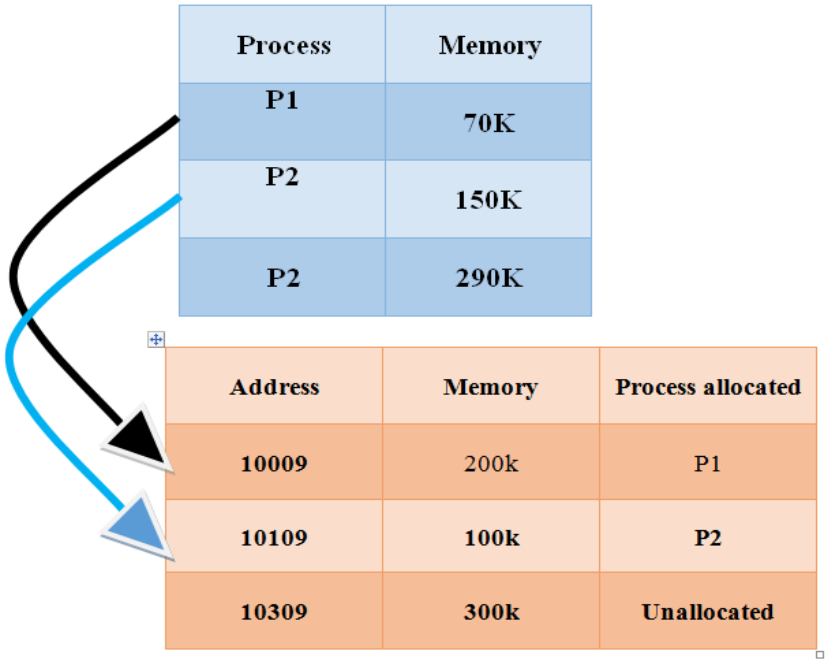

Figure: 1.1

1.2.2 Best Fit: The best fit approach deals with allocating the smallest free partition process which can be possible to meet the requirements of the requesting process. This algorithm first searches the entire list of free process partition and fit to the smallest process and then tries to find a hole which the partition is size is needed.

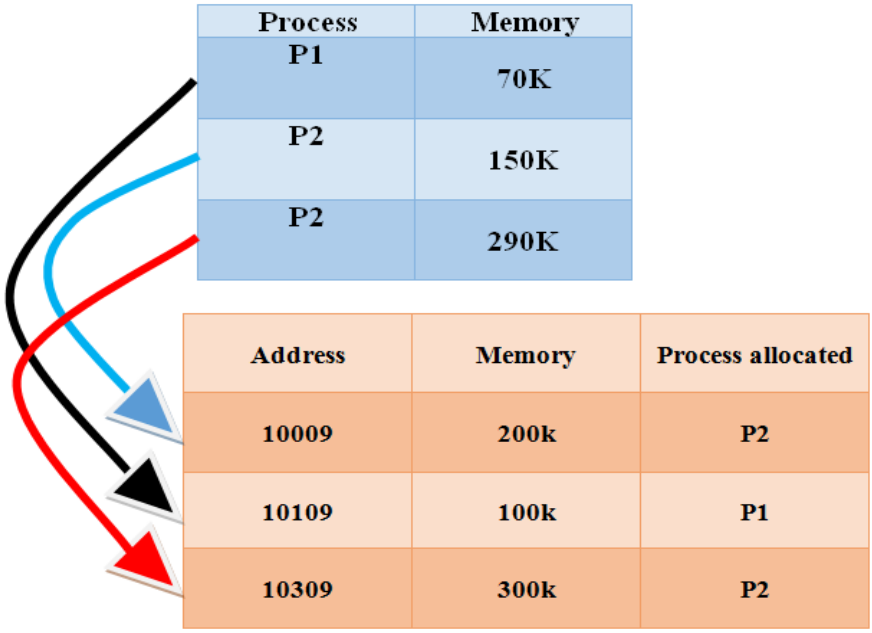

Figure: 1.2

1.2.3 Worst Fit: In worst fit approach is to allocate largest available process so that the process useful to the big enough process. It is the reverse of best fit.

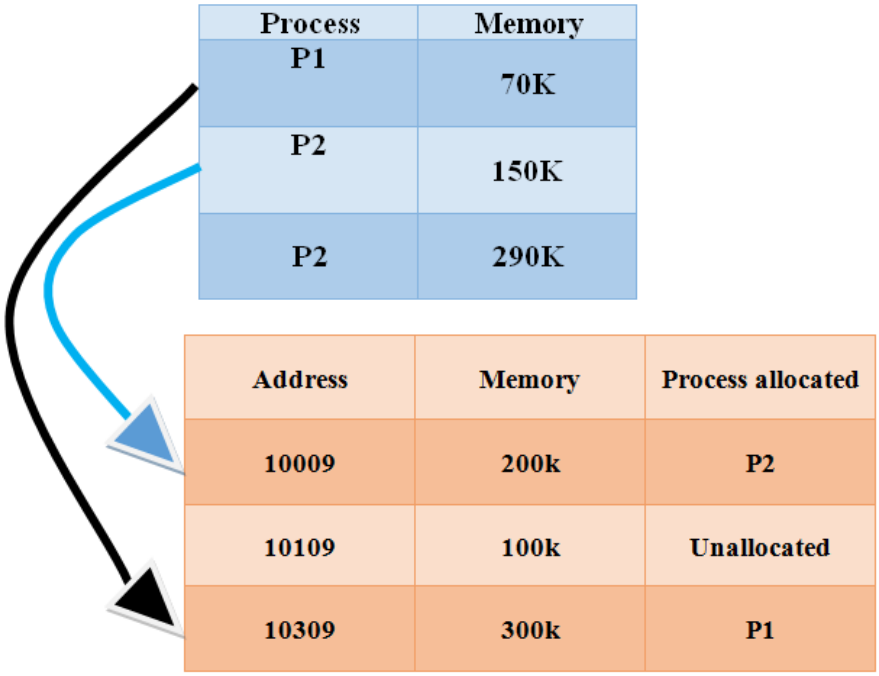

Figure 1.3

1.2.4 Next Fit: Next fit is a same as first fit. But when the next fit process is allocated the next fit allocates previously allocated process.

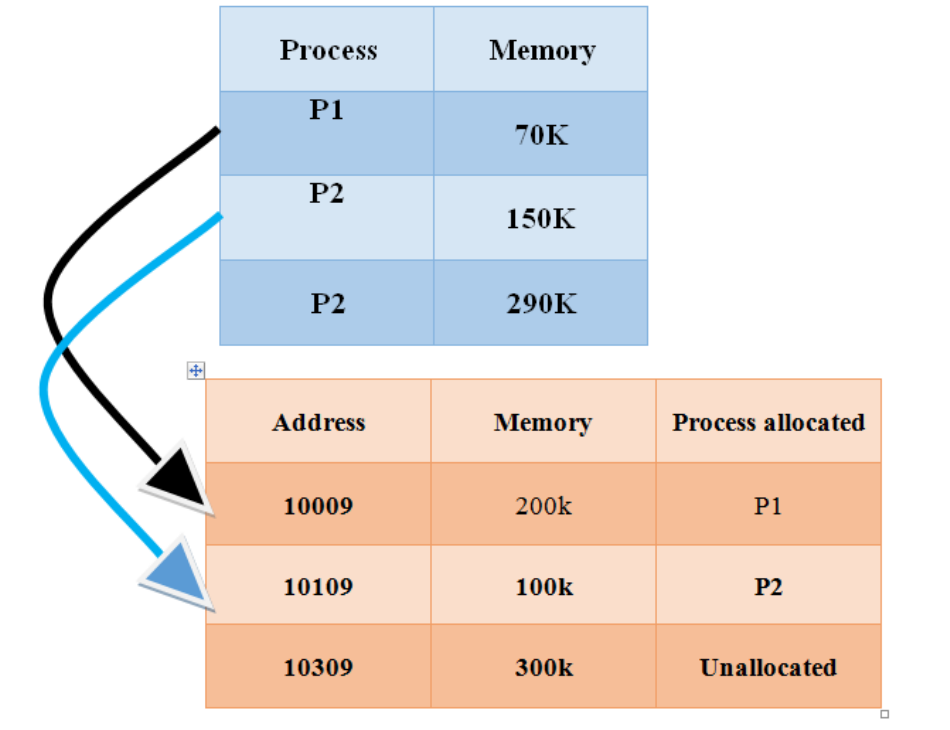

Figure 1.4

1.3 Advantages and disadvantages of Os memory management allocation:-

1.3.1 Advantages

- Allocating memory is easy and cheap.
- Any free page is ok; OS can be taken first one out of list it keeps.
- Eliminates external fragmentation.
- Data can be scattered all over physical memory.
- Pages are mapped appropriately in anyway.
- Allow s demand paging and prep-aging.
- More efficient swapping.
- No need for considerations about fragmentation.
- Just swap out page least likely to be used.

1.3.2 Disadvantages

- Longer memory access times.
- Can be improved using table.
- Guarded page tables.
- Inverted page tables.
- Memory requirements(one entry per virtual memory page).
- Improve using multilevel page tables and variable page size.
- Page table length register to limit virtual memory size.

CHAPTER 2

ANALYSIS AND DESIGN

2.1 Objectives of the project

The main objectives of operating memory management allocation are as follows:

A. **Re-locate ability:** The ability to move process around in memory without it affecting its execution. Os manages memory, not programmer, and processes may be moved around in memory. Memory management must convert program's logical addresses into physical addresses. Process first address is stored as virtual address 0.

1. **Static relocation:** Program must be relocated before or during loading of process into memory, program must always be loaded into some address space in memory, or re-locator must be run again.
2. **Dynamic relocation:** Process can be freely moved around in memory. Virtual to physical address space mapping is done during at run-time.

B. Protection: During write protection, it prevents data and instructions from being over-written. Whereas during read protection, it ensures privacy of data and instruction. Operating system needs to be protected from user processes, and user processes need to be protected from each other. Memory protection is usually supported by the hardware, because most languages allow memory addresses to be computed at run-time.

C. Sharing: Sometimes distinct processes may need to execute the same process, or even the same data. When different processes signal or wait the same semaphore, they need to access the same memory address. Operating system has to allow sharing, while at the same time ensure protection.

D. Physical organization of memory: Physical memory is expensive, so tends to be limited but the amount of physical memory helps to determine the degree of multi-programming. A two-level storage scheme can be used to virtually increase the overall amount of physical memory. Processes can be kept in secondary storage and only brought into physical memory when needed. Memory management and operating system have to manage operation of moving processes between the two levels.

2.2 REQUIREMENTS SPECIFICATIONS

2.2.1 Software Requirements Specification

1) Windows operating System version XP and above
2) Turbo C or C++ compiler 3.2 version and above

2.2.2 Hardware Requirements Specification

1) Processor : Pentium 4 and above
2) Ram : 512 MB(min)
3) Hard disk : 20 GB(min)
4) Clock speed of CPU : 1ghz(min)

2.3 Algorithm

First Fit

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Start by picking each process and check if it can be assigned to current block.
   If size-of-process <= size-of-block if yes then assign and check for next process.
   Else leave that process and keep checking the further processes
4. Display the processes with the blocks that are allocated to a respective process.
5. Stop.

Best Fit

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Start by picking each process and find the minimum block size that can be assigned to current process
   i.e., find min (block Size [1], block Size [2] ,..... block Size [n]) > process Size [current] . If found then assign it to the current process.
   Else leave that process and keep checking the further processes.
4. Display the processes with the blocks that are allocated to a respective process.
5. Stop.

Worst Fit

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Start by picking each process and find the biggest block size that can be assigned to current process
   i.e.. find max (block Size [1], block Size [2] ,..... block Size [n]) > process Size[current]
   If found then assign it to the current process.
   Else leave that process and keep checking the further processes.
4. Display the processes with the blocks that are allocated to a respective process.
5. Stop.

Next fit

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Assign value of position to first block
4. Start by picking each process and check if it can be assigned to block [position] .
   If size-of-process <= size-of-block [position] if yes then assigns the process and assign the value of position to next block and check for next process.
   Else leave that process and keep checking the further processes
5. Display the processes with the blocks that are allocated to a respective process.
6. Stop.

2.4 Class Diagram

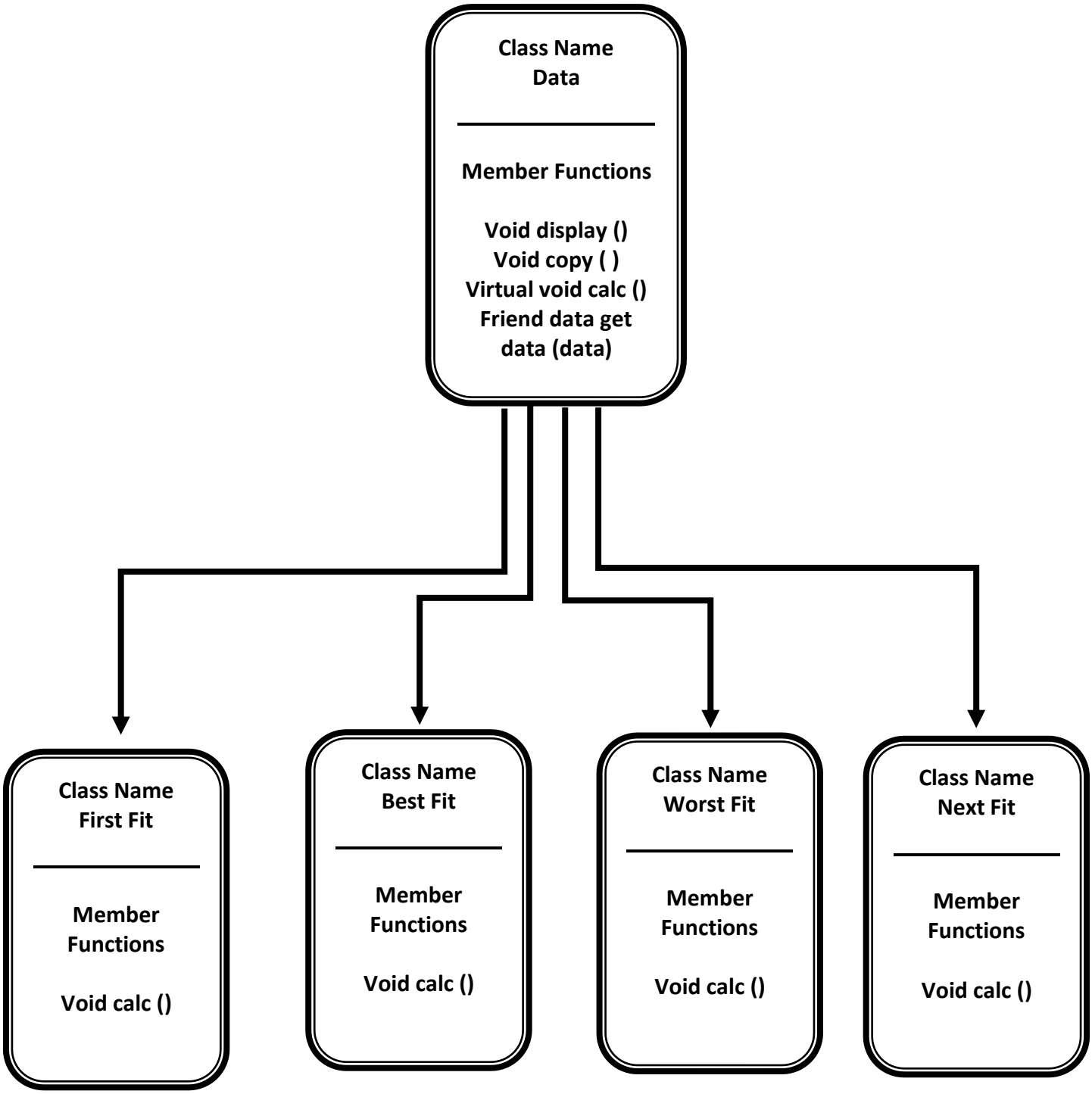

Figure 2.1

CHAPTER 3

IMPLEMENTATION

This Memory allocation techniques are implemented in C++ Program language using it's the features of C++ like class, objects, inheritance, data binding, encapsulation, polymorphism etc.

OOPS concepts Description

Class is a user defined data type, when you define a class; you define a copy for an object. In this project we have defined 5 classes namely Data, First fit, Best fit, Worst fit and Next Fit.

Objects are user defined data types of type class; we can create any number of objects for a class already defined. In this project we have defined objects for each class. And used it to store the data calculate and call the member functions.

Data Abstraction and Encapsulation: Data abstraction refers to providing only the essential data to the outside world, hidden data can't be viewed by user. A data encapsulation refers to wrapping up of the data into dingle unit or function. In this project we have provided enough security fir data by defining them in private section of data class only the derived classes and essential functions can access it, it cannot be accessed by the user or other functions.

Inheritance: One of the most useful concepts of object-oriented programming is code reusability. As the name suggests Inheritance is the process which the object of one class inherits the properties of another object of the class. Forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class. In this project we have used Hierarchical inheritance, Which means inheriting 2 or more class from one base class, here we have derived first fit, best fit, next fit, worst fit, classes from data class.

Polymorphism: Poly refers too many, Polymorphism is the technique of using same function name to do different tasks. In this project we have used calc () function to calculate the allocation for different allocation techniques in different class .

Friend Function: A friend function is a function of a class that is defined outside that class but it can access all private and protected members of the class. In this project we have used get data () function as a member function of class data so it can access all the private and protected members of class data.

Virtual functions: A virtual function is a member function which is declared within base class and is re-defined by derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class version of the function. In this project we have defined calc () function in base class data as virtual function.

Dynamic memory allocation: Dynamic memory allocation in C++ refers to performing memory allocation and de-allocation dynamically during the run time of a program. In this project we have used new and free functions to allocate memory dynamically and de-allocate them.

Dynamic Binding: Dynamic binding means that a block of code or a function executed with reference to a function call is determined only at the run time .In this project we have implemented dynamic binding as the function call calc() will know which calc() function to call only during run time based on user inputs.

CHAPTER 4

4.1 CODING

Program Code

```
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
//using namespace std;

class data              //base class
{
   protected:
   int mem[10],processes[10],n1,n2,flags[10],allocation[10];
public:
data()              // constructor to initialize predefined values
   {
      int i,j;
      for(i=0;i < 10;i++)
      {
               flags[i]=0;
               allocation[i]=-1;
      }
   }
 void display()
   {
      int i;
      cout<<"\nPartiton   Size   Process no.   Size";
      for(i=0;i < n1;i++)
      {
               cout<<"\n   "<<i<<"       "<<mem[i]<<"        ";
               if(flags[i]==1)
                        cout<<allocation[i]<<"          "<<processes[allocation[i]];
               else
      cout<<"NA";

      }
   }
  void copy(data d)
   {
      int i;
      n1=d.n1;
      n2=d.n2;
               for(i=0;i < n1;i++)
            mem[i]=d.mem[i];

      for(i=0;i < n2;i++)
          processes[i]=d.processes[i];

   }
   friend data getdata(data d);                    //friend function decleration
virtual void calc()
```

```
{
      cout<<"Vitrual class";
}
};

data getdata(data d)                        //friend function
   {
clrscr();
      int i,j;
   cout<<"\nEnter no. of memory blocks: ";
getch();
cin>>d.n1;

      for(i=0;i < d.n1;i++){
       cout<<"Enter size of block "<<i<<" : ";
       cin>>d.mem[i];
       }
      cout<<"\nEnter no. of processes :";
      cin>>d.n2;
      for(i=0;i < d.n2;i++){
       cout<<"Enter size of process "<<i<<" : ";
              cin>>d.processes[i];
   }
   return d;
   }


class firstfit :public data
{
   public:
   void calc()
   {
     int i,j;
      for(i=0;i < n2;i++)        //loop for each process
              for(j=0;j < n1;j++)        //loop for each memory block
                      if(flags[j]==0 && mem[j]>=processes[i])
                      {
                              allocation[j]=i;
                              flags[j]=1;
                              break;
                      }
   }
};



class bestfit :public data
{
   int i,j,smallest;
public:
   void calc()
   {
      for(i=0;i < n2;i++)           //loop for each process
      {
              smallest=-1;
              for(j=0;j < n1;j++)           //loop to find if process can fint n which block
```

```
                    if(flags[j]==0 && mem[j] >= processes[i])
                    {
                        smallest=j;
                        break;
                    }
            for(;j < n1;j++)            //loop to find the smallest block that can be
allocated
                    if(flags[j]==0 && mem[j] >= processes[i] && mem[j] <
mem[smallest])
                        smallest=j;
            if(smallest!=-1)
            {
                    allocation[smallest]=i;
                    flags[smallest]=1;
            }
        }
    }
};



class worstfit :public data
{
    int i,j,big;
public:
    void calc()
    {
        for(i=0;i < n2;i++)         //loop for each process
        {
            big=-1;
            for(j=0;j < n1;j++)             //loop to find if process can find n which
block
                    if(flags[j]==0 && mem[j] >= processes[i])
                    {
                        big=j;
                        break;
                    }
            for(;j < n1;j++)            //loop to find the biggest block that can be
allocated
                    if(flags[j]==0 && mem[j] >= processes[i] && mem[j] > mem[big])
                        big=j;
            if(big!=-1)
            {
                    allocation[big]=i;
                    flags[big]=1;
            }
        }
    }
};

class nextfit :public data
{
public:
    void calc()
    {
        int i,j;
```

```
        int k=0;//to remember the last position
        int c=0;//to keep the count
       for(i=0;i < n2;i++){        //loop for each process
              if(k==n1)
              {
                 k=0;
              }
              c=0;
                for(j=k;j < n1;j++){           //loop for each block
              c++;
                           if(flags[j]==0 && mem[j]>=processes[i])
                           {
                                    allocation[j]=i;
                                    flags[j]=1;
                                    k=j+1;
                                    break;
                           }//if loop
                  if(j==n1)
                  {
                     j=0;
                  }
                  if(c==n1)
       {
          break;
       }
                  }//j loop

                  }//i loop
    }
};

int main()
{
    data d,*p;
    d=getdata(d);
    bestfit b;
    firstfit f;
    worstfit w;
       nextfit m;
    int c,flag=0;
while(flag==0)
{
    cout<<"\n\n\nEnter\n   1 for first fit\n   2 for best fit\n   3 for worst fit\n   4 for next
fit\n   5 to exit\n                :";
    cin>>c;
    switch(c)
    {
case 1: clrscr();
    cout<<"\n\n*--------------------------------FIRST FIT--------------------------------*";
       f.copy(d);
       //f.calc();
       p=&f;
       p->calc();
       f.display();
       break;
case 2: clrscr();
```

```
    cout<<"\n\n*--------------------------------BEST FIT--------------------------------*";
      b.copy(d);
      //b.calc();
      p=&b;
      p->calc();
      b.display();
    break;
case 3:  clrscr();
    cout<<"\n\n*--------------------------------WORST FIT--------------------------------*";
      w.copy(d);
      //w.calc();
      p=&w;
      p->calc();
      w.display();
      break;
case 4: clrscr();
    cout<<"\n\n*--------------------------------NEXT FIT--------------------------------*";
      m.copy(d);
      //m.calc();
      p=&m;
      p->calc();
      m.display();
    break;

case 5: clrscr();
    cout<<"\n\nExiting.....\n";
    flag=1;
    break;
default:
    cout<<"Wrong choice try again\n";
    break;
}
}
getch();
return 0;
}
```

4.2 Output

```
Enter no. of memory blocks: 3
Enter size of block 0 : 200
Enter size of block 1 : 100
Enter size of block 2 : 300

Enter no. of processes :3
Enter size of process 0 : 100
Enter size of process 1 : 200
Enter size of process 2 : 300



Enter
   1 for first fit
   2 for best fit
   3 for worst fit
   4 for next fit
   5 to exit
                  :1

*--------------------------------FIRST FIT--------------------------------*
Partiton   Size   Process no.   Size
   0       200        0          100
   1       100        NA
   2       300        1          200


Enter
   1 for first fit
   2 for best fit
   3 for worst fit
   4 for next fit
   5 to exit
                  :2

*--------------------------------BEST FIT---------------------------------*
Partiton   Size   Process no.   Size
   0       200        1          200
   1       100        0          100
   2       300        2          300
```

```
Enter
   1 for first fit
   2 for best fit
   3 for worst fit
   4 for next fit
   5 to exit
                  :3

*--------------------------------WORST FIT--------------------------------*
Partiton   Size   Process no.   Size
   0       200        1          200
   1       100        NA
   2       300        0          100


Enter
   1 for first fit
   2 for best fit
   3 for worst fit
   4 for next fit
   5 to exit
                  :4

*--------------------------------NEXT FIT---------------------------------*
Partiton   Size   Process no.   Size
   0       200        0          100
   1       100        NA
   2       300        1          200
```

CHAPTER 5

CONCLUSION

Memory management continues to be a trying problem in C++, In the general case, there are several techniques for memory management. We have wrapped all memory allocation techniques in a single program this program can be used to identify which memory allocation technique is more efficient for your program, We conclude that best fit is not always the best algorithm to be used in some cases worst fit and also first fit is more efficient than best fit in case of time complexity. This project can be used to compare between all the memory allocation techniques and find the most efficient one.

5.2 Scope of future Work

During the course of developing the improvements described by this work many ideas of potential further study. There is a lot of memory wastage because of internal and external fragmentation. These can be solved using buddy system technique in which for each process allocated more that 50% of memory is used for sure. This project can be further modified with memory fragmentation and buddy system techniques.

BIBLIOGRAPHY

1. madnick, stuart; donovan, john [1974]. *operating systems*. mcgraw-hill book company. isbn 0- [07-039455-5].
2. memory management: challenges and techniques for traditional memory allocation algorithms in relation with today's real time needs muhammad abdullah awais ms150200157, virtual university of pakista
3. https://en.wikipedia.org/wiki/memory_management_(operating_systems)
4. dynamic memory allocation: role in memory management - prabhudev irabashetti
5. a dynamic virtual memory management under real-time constraints martin böhnert