Contents

LIST OF FIGURES **1**

1. FILE ALLOCATION METHODS **2**

1.1. CONTIGUOUS ALLOCATION METHOD 2
1.1.1. How it works *2*
1.1.2. Code *2*
1.1.3. Explain the code *7*
1.1.4. Parameters *7*
1.2. LINKED FILE ALLOCATION 8
1.2.1. How it works *8*
1.2.2. The code *8*
1.3. INDEXED FILE ALLOCATION 10
1.3.1. How it works *10*
1.3.2. The code *10*
1.3.3. Explain the code *11*

2. FILE HANDLING **13**

2.1. C PROGRAM TO READ A LINE FROM A FILE AND DISPLAY IT 14
2.2. C PROGRAM TO WRITE A SENTENCE TO A FILE 14

List of Figures

Figure 1-Contiguous allocation method 2
Figure 2 - Linked allocation method 8
Figure 3 - Indexed allocation method 10

1. File Allocation Methods

The direct-access nature of disks gives us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating

disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

In addition to storing the actual file data on the disk drive, the file system also stores metadata about the files: the name of each file, when it was last edited, exactly where it is on the disk, and what parts of the disk are "free". Free areas are not currently in use by the file data or the metadata, and so available for storing new files. (The places where this metadata is stored are often called "inodes"

1.1. Contiguous allocation method

1.1.1. How it works

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk.

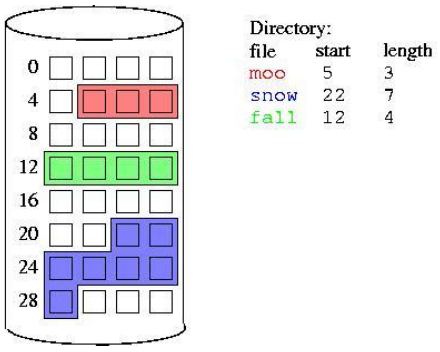

Figure 1-Contiguous allocation method

Both sequential access and direct/Random access are supported by the contiguous allocation. We can say that it supports random access as using Disk Block Address we can jump directly on the required location.

The disadvantage of contiguous allocation is that it is often difficult to increase the size of a file as the next contiguous block may not be free. Moreover, one is often not sure of the space required while creating a new file. The various methods adopted to find space for a new file suffer from external fragmentation. Internal fragmentation may exist in the last disk block of a file.

1.1.2. Code

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

void create(int,int);
void del(int);
void compaction();
void display();

int fname[10],fsize[10],fstart[10],freest[10],freesize[10],m=0,n=0,start;

int main()
{
   int name,size,ch,i;
   int *ptr;
   system("cls");
   ptr=(int *)malloc(sizeof(int)*100);
   start=freest[0]=(int)ptr;
   freesize[0]=500;

   printf("\n\n");
   printf(" Free start address         Free Size        \n\n");

   for(i=0; i<=m; i++)
      printf("    %d                    %d\n",freest[i],freesize[i]);
   printf("\n\n");
   while(1)
   {

      printf("1.Create.\n");
      printf("2.Delete.\n");
      printf("3.Compaction.\n");
      printf("4.Exit.\n");
      printf("Enter your choice: ");
      scanf("%d",&ch);
      switch(ch)
      {
      case 1:
         printf("\nEnter the name of file: ");
         scanf("%d",&name);
         printf("\nEnter the size of the file: ");
         scanf("%d",&size);
         create(name,size);
         break;
      case 2:
         printf("\nEnter the file name which u want to delete: ");
```

```
            scanf("%d",&name);
            del(name);
            break;
        case 3:
            compaction();
            printf("\nAfter compaction the tables will be:\n");
            display();
            break;
        case 4:
            exit(1);
        default:
            printf("\nYou have entered a wrong choice.\n");
        }
    }

}


void create(int name,int size)
{
    int i,flag=1,j,a;

    for(i=0; i<=m; i++)
        if( freesize[i] >= size)
            a=i,flag=0;
    if(!flag)
    {
        for(j=0; j<n; j++);
        n++;
        fname[j]=name;
        fsize[j]=size;
        fstart[j]=freest[a];
        freest[a]=freest[a]+size;
        freesize[a]=freesize[a]-size;

        printf("\n The memory map will now be: \n\n");
        display();
    }
    else
    {
        printf("\nNo enough space is available.System compaction......");
        flag=1;

        compaction();
        display();

        for(i=0; i<=m; i++)
            if( freesize[i] >= size)
                a=i,flag=0;
        if(!flag)
        {
            for(j=0; j<n; j++);
            n++;
            fname[j]=name;
            fsize[j]=size;
            fstart[j]=freest[a];
            freest[a]+=size;
            freesize[a]-=size;
            printf("\n The memory map will now be: \n\n");
            display();
        }
        else
            printf("\nNo enough space.\n");
    }
}

void del(int name)
{
    int i,j,k,flag=1;
    for(i=0; i<n; i++)
        if(fname[i]==name)
            break;
    if(i==n)
    {
        flag=0;
        printf("\nNo such process exists......\n");
    }
    else
    {
        m++;
        freest[m]=fstart[i];
        freesize[m]=fsize[i];
        for(k=i; k<n; k++)
        {
            fname[k]=fname[k+1];
            fsize[k]=fsize[k+1];
            fstart[k]=fstart[k+1];
        }
        n--;
    }
    if(flag)
    {
        printf("\n\n After deletion of this process the memory map will be : \n\n");
        display();
    }
}

void compaction()
{
    int i,j,size1=0,f_size=0;
    if(fstart[0]!=start)
    {
        fstart[0]=start;
        for(i=1; i<n; i++)
            fstart[i]=fstart[i-1]+fsize[i-1];
    }
    else
    {
        for(i=1; i<n; i++)
            fstart[i]=fstart[i-1]+fsize[i-1];
    }
    f_size=freesize[0];

    for(j=0; j<=m; j++)
        size1+=freesize[j];
    freest[0]=freest[0]-(size1-f_size);
    freesize[0]=size1;
    m=0;
}

void display()
{
    int i;

    printf("\n  ***   MEMORY MAP TABLE  ***       \n");
    printf("\n\nNAME     SIZE    STARTING ADDRESS      \n\n");
    for(i=0; i<n; i++)
       printf(" %d%10d%10d\n",fname[i],fsize[i],fstart[i]);
    printf("\n\n");
    printf("\n\n***  FREE SPACE TABLE  ***\n\n");
    printf("FREE START ADDRESS             FREE SIZE        \n\n");
    for(i=0; i<=m; i++)
       printf("     %d                        %d\n",freest[i],freesize[i]);
}
```

1.1.3. Explain the code

In this code it is required to enter the mode either create, delete, compaction, display or exit.

In creating file, you will enter its name and size and if there is enough free space it will be created (note: initial free size is 500 for simplicity).

In deleting, you will enter the name and its size will be removed from the free size.

Compaction will search for the free blocks and compact the file all aside (contiguously).

Display will show the used bytes/blocks in table and the files which use the space.

1.1.4. Parameters

Ptr : pointer to the file to use random address

fname[10] : the name of the file in integer

fsize[10] : the size of the file created

freest[10] : start address

freesize[10] : the remaining size of the 500

in creation the size will be removed from (freesize) and the numer of blocks will be added (contiguously) to (freest).

In deleteing the opposite will happen

To create or delete more than file the loop of n,m (number of blocks and address of files and number of files) goes to add or remove size and address.

1.2. Linked file allocation

1.2.1. How it works

In linked allocation, each file is a linked list of disk blocks. The directory contains a pointer to the first and (optionally the last) block of the file. For example, a file of 5 blocks which starts at block 4, might continue at block 7, then block 16, block 10, and finally block 27. Each block contains a pointer to the next block and the last block contains a NIL pointer. The value -1 may be used for NIL to differentiate it from block 0.

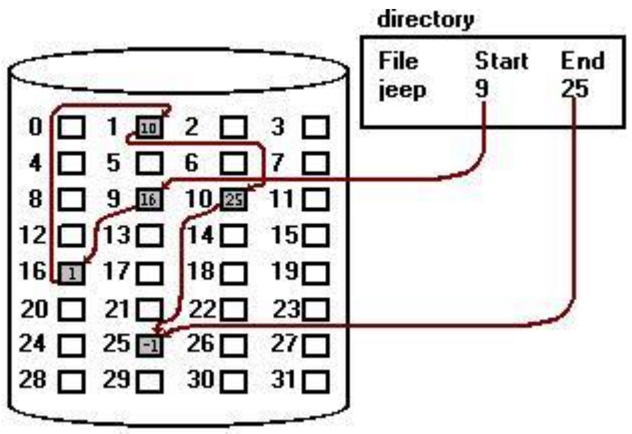

Figure 2 - Linked allocation method

1.2.2. The code

```
#include<stdio.h>
#include<conio.h>
struct file
{
   char fname[10];
   int start,size,block[10];
} f[10];
main()
{
   int i,j,n;
   system("cls");
   printf("Enter no. of files:");
   scanf("%d",&n);
   for(i=0; i<n; i++)
   {
      printf("Enter file name:");
      scanf("%s",&f[i].fname);
      printf("Enter starting block:");
      scanf("%d",&f[i].start);
      f[i].block[0]=f[i].start;
      printf("Enter no.of blocks:");
      scanf("%d",&f[i].size);
      printf("Enter block numbers:");
      for(j=1; j<=f[i].size; j++)
      {
            scanf("%d",&f[i].block[j]);
        }
    }
    printf("File\tstart\tsize\tblock\n");
    for(i=0; i<n; i++)
    {
        printf("%s\t%d\t%d\t",f[i].fname,f[i].start,f[i].size);
        for(j=1; j<=f[i].size-1; j++)
            printf("%d--->",f[i].block[j]);
        printf("%d",f[i].block[j]);
        printf("\n");
    }
    getch();

}
```

1.3. Indexed file allocation

1.3.1. How it works

Linked allocation does not support random access of files, since each block can only be found from the previous. Indexed allocation solves this problem by bringing all the pointers together into an index block. One disk block is just used to store DBAs (disk block addresses) of a file.

Every file is associated with its own index node. If a file is very large then one disk block may not be sufficient to hold all associated DBAs of that file. If a file is very small then some disk block space is wasted as DBAs are less and a single disk block could still hold more DBAs.

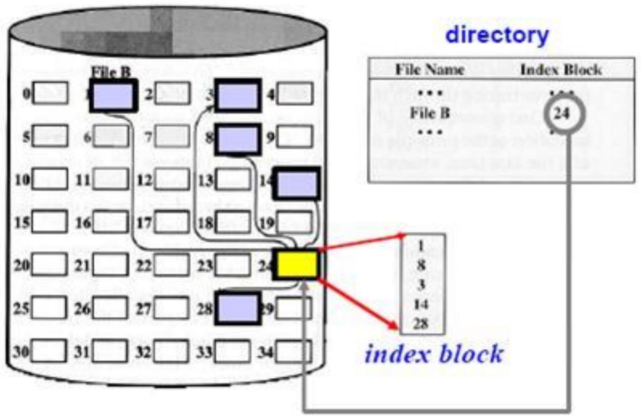

Figure 3 - Indexed allocation method

This method solves the problem of fragmentation as the blocks can be stored at any location.

1.3.2. The code

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int n;
void main()
{
   int b[20],b1[20],i,j,blocks[20][20],sz[20];
   char F[20][20],S[20],ch;
   system("cls");
   printf("\n Enter no. of Files ::");
   scanf("%d",&n);
   for(i=0; i<n; i++)
   {
      printf("\n Enter file %d name ::",i+1);
      scanf("%s",&F[i]);
      printf("\n Enter file%d size(in kb)::",i+1);
      scanf("%d",&sz[i]);
      printf("\n Enter blocksize of File%d(in bytes)::",i+1);
      scanf("%d",&b[i]);
   }
   for(i=0; i<n; i++)
```

```
    {
        b1[i]=(sz[i]*1024)/b[i];
        printf("\n\nEnter blocks for file%d",i+1);
        for(j=0; j<b1[i]; j++)
        {
            printf("\n Enter the %dblock ::",j+1);
            scanf("%d",&blocks[i][j]);
        }
    }
    do
    {
        printf("\nEnter the Filename ::");
        scanf("%s",&S);
        for(i=0; i<n; i++)
        {
            if(strcmp(F[i],S)==0)
            {
                printf("\nFname\tFsize\tBsize\tNblocks\tBlocks\n");
                printf("\n---------------------------------------------\n");
                printf("\n%s\t%d\t%d\t%d\t",F[i],sz[i],b[i],b1[i]);
                for(j=0; j<b1[i]; j++)
                    printf("%d->",blocks[i][j]);
            }
        }
        printf("\n---------------------------------------------\n");
        printf("\nDo U want to continue ::(Y=1:n=0)");
        scanf("%d",&ch);
    }
    while(ch!=0);
}
```

1.3.3. Explain the code

You will be asked to enter the number of files, names and sizes

You will also assign the size of the block on your own

You can enter the blocks of each file (note: this can be done randomly by rnd() and using pointer fot address)

The program will display the index block with the pointer to each next block, and the size of the file.

2. File handling

C language can be used to handle files of txt and bin

The file I/O functions in the stdio library are:

- fopen – opens a text file.
- fclose – closes a text file.
- feof – detects end-of-file marker in a file.
- fscanf – reads formatted input from a file.
- fprintf – prints formatted output to a file.
- fgets – reads a string from a file.
- fputs – prints a string to a file.
- fgetc – reads a character from a file.
- fputc – prints a character to a file.

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.

ab+ Open for both reading and appending in binary mode. If the file does not exists, it will be created.

2.1. C Program to Read a Line From a File and Display it

```
#include <stdio.h>
#include <stdlib.h> // For exit() function
int main()
{
    char c[1000];
    FILE *fptr;

    if ((fptr = fopen("program.txt", "r")) == NULL)
    {
        printf("Error! opening file");
        // Program exits if file pointer returns NULL.
        exit(1);
    }

    // reads text until newline
    fscanf(fptr,"%[^\n]", c);

    printf("Data from the file:\n%s", c);
    fclose(fptr);

    return 0;
```

2.2. C Program to Write a Sentence to a File

```
#include <stdio.h>
#include <stdlib.h>  /* For exit() function */
int main()
{
   char sentence[1000];
   FILE *fptr;

   fptr = fopen("program.txt", "w");
   if(fptr == NULL)
   {
       printf("Error!");
       exit(1);
   }

   printf("Enter a sentence:\n");
   gets(sentence);

   fprintf(fptr,"%s", sentence);
   fclose(fptr);

   return 0;
}
```