



FakeOS

CSD2180

OPERATING SYSTEMS

Assignment 2

AY 2021/22

Name	Student ID	Degree
Muhammad Syafiq Bin Muhammad Ismail	2001528	RTIS
Lee Ming Yang	2001311	RTIS
Wilfred Ng Jun Hwee	2001388	RTIS
Wong Wei Hao	2002316	RTIS
Lee Hon Han Leonard	2001896	RTIS
Chew Ying Ying Cassandra	2002196	RTIS

Table of Contents

1. Assignment Overview	2
1.1 Assignment Objectives	2
1.2 Implementation Requirements	2
1.3 File Allocation Algorithms	2
2. Actual Implementation	3
2.1 Common Resources	3
2.2 Free-Space Management Method	3
2.3 Allocation Methods	4
2.3.1 Contiguous Allocation	4
2.3.2 Linked Allocation	9
2.3.3 Indexed Allocation	14
3. User Manual	18
3.1 Files	18
3.2 Input	19
3.3 Compilation Instructions	19
3.4 Execution Instructions	20
3.5 Output	21
3.5.1 Example Output	21

1. Assignment Overview

1.1 Assignment Objectives

1. Simulate the disk allocation and free-space management implementation of a file system.
2. The objective is to evaluate the various methods to determine its efficiency (i.e. speed in accessing file content) and flexibility (i.e. ability in keeping different file sizes).

1.2 Implementation Requirements

- The program must be written in C language.
- The simulated hard-disk is an integer array of size 500, with 5 entries as a disk block
- Team size of 5/6 students.
- Implement a Directory structure to track basic file information and location of where file contents are being stored. Either linear or hash table method
- Directory Structure has to occupy the first two blocks of the hard drive
- Implement all three disk block allocation methods (i.e. Contiguous, Linked and Indexed).
- Bit-map or Linked list of Free-space management method to determine free block location
- Filenames are integers and the accepted filenames are from 0 to 127.
- File contents accepted in this hard-disk are 4 byte positive integer values.

1.3 File Allocation Algorithms

A file consists of numerous datas and information, usually stored on a persistent storage device such as a classic hard disk drive or a more modern solid-state storage device. Files and directories are abstractions that virtualize storage devices.

There are different methods of file allocations such as **Contiguous Allocation**, **Linked Allocation** and **Indexed Allocation**. These methods provide quick access to the file blocks and also the utilization of disk space in an efficient manner.

2. Actual Implementation

2.1 Common Resources

- 1) Hard Disk → Simulated using an array of 500 *ints* (5 elements equals to 1 block)
- 2) Free-space management using bitmaps → 100 blocks would require 2 *unsigned long long int*
- 3) Helper functions to read input .csv files and return an array of operation

2.2 Free-Space Management Method

As the hard disk is simulated as 500 *ints* where 5 *ints* are equal to a block, 100 blocks of memory will exist in the simulated hard disk. To account for the 100 blocks of memory, 2 bitmaps of type *unsigned long long int* will be required as an *unsigned long long int* has a size of 64 bits.

Example of Free-Space Management using bitmaps:

Bitmap 1

Bit/Block	0	1	2	3	...
Allocated	1	0	1	1	...

Bitmap 2

Bit/Block	50	51	...	98	99
Allocated	n	n	...	n	n

Index	Block	File Data
10	2	101
11	2	12
12	2	10
13	2	14
14	2	3
15	3	105
16	3	-1
17	3	
18	3	
19	3	

From the image above, Bitmap1 occupies blocks 0-49 and Bitmap2 occupies blocks 50-99. Using bitwise manipulation, Block 2 is considered allocated if the following code evaluates to **True**.

```
if (bitmap1 & (1 << 2)) { /* Do something */ }
```

2.3 Allocation Methods

2.3.1 Contiguous Allocation

In a contiguous file allocation, in the hard disk, all the blocks get the contiguous physical block. It has the best performance in most cases as the memory is being stored together and is also more simple to implement as compared to the linked and index allocation methods. This allocation method only requires the starting block and the number of blocks.

The 3 main algorithms used for contiguous memory allocation are **First Fit**, **Best Fit**, and **Worst Fit**.

- In **First Fit**, this first spot with enough space found will be selected regarding if the space found is much more than the required space.
- In **Best Fit**, the program will go through all the empty spaces and find the smallest spot large enough to fit the process.
- In **Worst Fit**, the largest spot found in the memory will be chosen.

In our implementation, we have chosen to use the best fit algorithm as it is more efficient. In this algorithm, we only allocate enough space to fit the process so that we will not waste large spaces and create many redundant small empty spaces.

Our implementation of the contiguous allocation method is as follows:

Adding/Allocating new files

1. Firstly calculate the required amount of blocks that is needed to store our file and check if there is enough spaces in the harddisk

```
//Finding out how many blocks we need
int requiredBlocks = (dataCount % ELEMENTSINBLOCK) != 0?
dataCount/ELEMENTSINBLOCK + 1 : dataCount/ELEMENTSINBLOCK;
```

```
// First rejection test
if (emptyBlockCount < requiredBlocks) {
    printf("Not enough space left in memory.\n");
    return false; }

//Second rejection test
if(requiredBlocks > ELEMENTSINBLOCK){
    printf("Required Blocks more than memory can hold");
    return false;
}
```

2. If there is enough space, iterate through the blocks in the hardisk and find a continuous block size that is similar to the required blocks needed to store our files. We then store the element one after another in the block. If the number of elements is more than the number of elements a block can hold, allocate data to the next block of memory.

```
//Check if contiguous blocks is equal to the number of blocks
if(contiguousBlocks == requiredBlocks)
{
    //Starting block
    startBlock = i + 1 - requiredBlocks;
    //Iterate through the block
    for(int j = startBlock * 5; k < dataCount; ++j, ++k)
    {
        //Set that element of the block to the data
        hard_disk[j] = data[k];

        //If goes to the next block
        if((k % 6) == 0)
        {
            ++checker;
        }
    }
    endBlock = startBlock + checker; //Endblock
    allocated = true;
    break;
}
```

Reading files

1. Iterate through the file directory to get the file information to be read.

```
for (; directorySlot < DIRECTORYINDEX; ++directorySlot)
{
    GetDirStruct(dirData, hard_disk, directorySlot);
    if (dirData[0] == FileNum) { break; }
    memset(dirData, 0, 4);
}
```

2. Iterate through the blocks that store that data elements and then read and print out the data content.

```
for(int i = dirData[1]; i <= dirData[2]; ++i)
{
    int elementNum = i * 5;
    int maxElements = elementNum + 5;
    //Iterate through the elements
    for(;elementNum < maxElements; ++elementNum)
    {
        //If reach last element break out of the loop
        if(hard_disk[elementNum] == -1)
        {
            break;
        }
        //Print data
        printf("%d ", hard_disk[elementNum]);
    }
}
```


Deleting files

1. Iterate through the directory to find the file to be deleted.

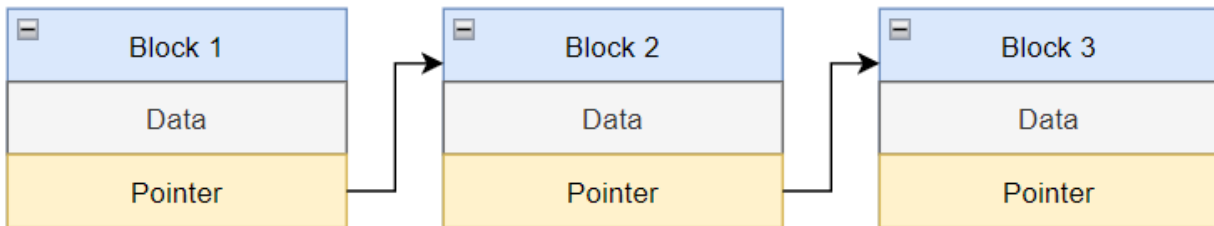
```
for (; directorySlot < DIRECTORYINDEX; ++directorySlot)
{
    GetDirStruct(dirData, hard_disk, directorySlot);
    if (dirData[0] == FileNum) { break; }
    memset(dirData, 0, 4);
}
```

2. Iterate through the blocks to be deleted and reset the content to 0

```
for(int i = dirData[1]; i <= dirData[2]; ++i)
{
    int elementNum = i * 5;
    int maxElements = elementNum + 5;
    //Clear the bits for that block
    clear_bit(&bitmap1, &bitmap2, i);
    //Iterate through the elements and set to 0
    for(;elementNum < maxElements; ++elementNum)
    {
        //If reach last element break
        if(hard_disk[elementNum] == -1)
        {
            hard_disk[elementNum] = 0;
            break;
        }
        hard_disk[elementNum] = 0;
    }
}
```

2.3.2 Linked Allocation

Linked file allocation is similar to linked list data structure, where each file is a linked list of disk-blocks. The directory structure contains the first data block which is considered to be the “head” and last data blocks of the file and each data block contains a pointer to the next block.



Our implementation of Linked Allocation is as follows :

By parsing data from the excel file, the operation, file name and data is obtained.

Adding files

1. Calculate the number of blocks required to store the file and check if there is enough space in the hard drive by checking the bitmap.

```
//sanity check that we have enough space to allocate
//getting the number of required blocks
int requiredBlocks = (dataCount % ELEMENTSINBLOCK) != 0 ? dataCount/ELEMENTSINBLOCK + 1 : dataCount/ELEMENTSINBLOCK;

///getting the number of empty block count
// First rejection test
if (getNumberOfFreeBlocks() < requiredBlocks) { return false; }
// Second rejection test
if (requiredBlocks > ELEMENTSINBLOCK) { return false; }
```

2. If there is enough space, determine the numbers of blocks required to store the file data, starting and ending block index.

```

if(blocksRequired == 1) //only one block required
{
    storageBlockIndex[0] = getFreeBlock();
    startIndex = storageBlockIndex[0];
    endIndex = storageBlockIndex[0];
    set_bit(&bitmap1,&bitmap2,storageBlockIndex[0]);
}
else // more then 1 block required
{
    for(int j = 0 ; j < blocksRequired ; j++){
        if(j == 0)
        {
            storageBlockIndex[j] = getFreeBlock();//getting the index of the free blocks
            startIndex = storageBlockIndex[j];
        }
        else if(j == blocksRequired - 1)
        {
            //last block
            storageBlockIndex[j] = getFreeBlock();//getting the index of the free blocks
            endIndex = storageBlockIndex[j];
        }
        else
        {
            storageBlockIndex[j] = getFreeBlock();//getting the index of the free blocks
        }
        set_bit(&bitmap1,&bitmap2,storageBlockIndex[j]);//setting the bit index
    }
}

```

3. Store the data using linked allocation, while taking note that the last space in the block contains the “pointer” to the next block

```

//store all the formatted data into the hard_disk
int dataIndex = 1;
for(int k = 0; k < blocksRequired; k++){
    int startHardDiskIndex = storageBlockIndex[k] * ELEMENTSINBLOCK; //getting the starting hard disk index

    //loop through the block to store data in
    for(int m = 0; m < ELEMENTSINBLOCK; m++){
        //if it's the last one, store the pointer to the next block and it's the last block
        if(m == ELEMENTSINBLOCK-1 && k != blocksRequired){
            hard_disk[startHardDiskIndex++] = storageBlockIndex[k+1]; //stores the pointer to next block
            // printf("Storing pointer to next block : %d\n", hard_disk[startHardDiskIndex-1]);
            break; //break the loop
        } else if(k == blocksRequired-1 && dataIndex == dataCount){
            //storing the last index indicator
            hard_disk[startHardDiskIndex++] = -1;
            // printf("Storing of the last data : %d\n", hard_disk[startHardDiskIndex-1]);
            break; //break the for loop
        } else{
            hard_disk[startHardDiskIndex++] = data[dataIndex]; //storing of the data into the harddisk
            // printf("Storing normal file data : %d\n", hard_disk[startHardDiskIndex-1]);
        }
        dataIndex++; //incrementing
    }
}
}

```

Reading Files

1. Determine which file is to be read by cross referencing the directory structure

```

int foundIndex = 0;
char cfileDir[4] = {0};
for(int i = 0; i < DIRECTORYINDEX; ++i)
{
    GetDirStruct(cfileDir, hard_disk, i); // the directory found
    if(cfileDir[0] == data[0]) // file found within the the file directory
    {
        foundIndex = i;
        break;
    }
    if(i == DIRECTORYINDEX){
        return false;
    }
}
}

```

2. While not end of the the file content, print content out

```

int currentBlock = cfileDir[1]; //stores the current block index
int endBlock = cfileDir[2];     //stores the end block
//main reading
printf( "Contents : ");
while(true)
{
    if(currentBlock == endBlock){

        int Index = currentBlock * ELEMENTSINBLOCK;

        while(hard_disk[Index] != -1)
        {
            printf( "%d ", hard_disk[Index++]);
        }
        break;//break out of the reading loop
    }else{
        int Index = currentBlock * ELEMENTSINBLOCK;//getting the starting index of the disk
        //read 4 entries
        for(int i = 0; i < ELEMENTSINBLOCK-1 ; ++i){
            printf( "%d ", hard_disk[Index++]); // prints out 4 spaces of the block
        }
        currentBlock = hard_disk[Index];//assign current block to the next block
    }
}
printf( "\n" );
return true;

```

Deleting File

1. Determine which file to be delete by looking at the directory structure

```

printf("Deleting file: %d..\n", data[0]);
char cfileDir[4] = {0};
for(int i =0 ; i < DIRECTORYINDEX; ++i)
{
    GetDirStruct(cfileDir, hard_disk, i); // the directory found
    if(cfileDir[0] == data[0]) // file found within the the file directory
    {
        hard_disk[i] = 0;
        break;
    }
}

```

2. While end of the file has not been reached, set content to 0

```
int currentBlock = cfileDir[1]; //stores the current block index
int endBlock = cfileDir[2];      //stores the end block
//main reading
printf( "File Name : %d, Content : ", cfileDir[0] );
while(true)
{
    //read all 5 entries
    if(currentBlock == endBlock){

        int Index = currentBlock * ELEMENTSINBLOCK; //index
        while(hard_disk[Index] != -1)
        {
            hard_disk[Index] = 0; //remove everything within the entry
            Index++;
        }
        hard_disk[Index] = 0; //removing the pointer
        clear_bit(&bitmap1, &bitmap2, currentBlock); //reseting the bitmap
        break; //break out of the reading loop
    }
    else // deleting 4 elements
    {
        int Index = currentBlock * ELEMENTSINBLOCK; //getting the starting index of the disk
        //deleting 4 entries
        for(int i = 0; i < ELEMENTSINBLOCK-1 ; ++i)
        {
            hard_disk[Index] = 0; //remove everything within the entry
            Index++;
        }
        clear_bit(&bitmap1, &bitmap2, currentBlock); //reseting the bitmap
        currentBlock = hard_disk[Index]; //assign current block to the next block
        hard_disk[Index] = 0; //deleting the 5th entry
    }
}
return false;
```

2.3.3 Indexed Allocation

For each file, the index allocation method uses an extra block known as the index block. An array of disk-block addresses will be stored in this index block. The *i*th item in the index block refers to the file's *i*th block. The index block's address is stored in the directory. The pointer in the *i*th index-block entry is used to locate and read the *i*th block.

Adding File

1. First we check if there is any space in the file directory

```
bool Add(const int* data, int dataCount)
{
    printf("Adding File: %d...\n", data[0]);
    // Find slot for directory data
    int directorySlot = 0;
    for (; directorySlot < DIRECTORYINDEX; ++directorySlot)
    {
        if (hard_disk[directorySlot] == EMPTY) { break; }
    }
    if (directorySlot == DIRECTORYINDEX) { return false; } // No empty directory slot avail
```

2. We check if there is space in the bitmap for the number of data required to store and also check if the required blocks are less than the elements in the block. If these two checks are passed, we then create data for the directory. It contains 3 characters, first character being the filename, second character being the indexing block and the last character being the null terminator.

```
char dirData[4] = {0, 0, 0, '\0'}; // Initialize data
dirData[0] = data[0]; // Store file name
int requiredBlocks = ((dataCount-1) % ELEMENTSINBLOCK) != 0 ? (dataCount-1)/ELEMENTSINBLOCK + 1 : (dataCount-1)/ELEMENTSINBLOCK;
// Check if bitmap has enough space to allocate
// First rejection test
if (getNumberOfFreeBlocks() < requiredBlocks+1) { return false; }
// Second rejection test
if (requiredBlocks > ELEMENTSINBLOCK) { return false; } // Check if there is too many blocks to store in indexing block
```

3. We get the next free block as our indexing block from the bitmap and set the bit in the bitmap. This index block contains the “pointers” to the blocks allocated with data.

```

// Get the first block as indexing block
int indexingBlock = getFreeBlock();
set_bit(&bitmap1, &bitmap2, indexingBlock);
dirData[1] = indexingBlock;

// Allocate data to data blocks
int dataIterator = 0;
// For each required block to store the data...
for (int blockNum = 0; blockNum < requiredBlocks; ++blockNum)
{
    // Allocate free block
    int freeBlockNumber = getFreeBlock();
    set_bit(&bitmap1, &bitmap2, freeBlockNumber);
    hard_disk[indexingBlock*ELEMENTSINBLOCK + blockNum] = freeBlockNumber; // Store allocated block into indexing block
    // Insert data into block
    do
    {
        hard_disk[(freeBlockNumber*ELEMENTSINBLOCK)+(dataIterator%ELEMENTSINBLOCK)] = data[dataIterator+1];
        ++dataIterator;
    } while (((dataIterator%ELEMENTSINBLOCK) != 0) && dataIterator < (dataCount-1));

    // Set end block if this is the last block allocated
    if (blockNum == requiredBlocks-1) { dirData[2] = freeBlockNumber; }
}

```

4. For each required block needed to store data, we find the next free block and set the bitmap. We then store the allocated block into the indexing block and insert data into the allocated block.

```

// Allocate data to data blocks
int dataIterator = 0;
// For each required block to store the data...
for (int blockNum = 0; blockNum < requiredBlocks; ++blockNum)
{
    // Allocate free block
    int freeBlockNumber = getFreeBlock();
    set_bit(&bitmap1, &bitmap2, freeBlockNumber);
    hard_disk[indexingBlock*ELEMENTSINBLOCK + blockNum] = freeBlockNumber; // Store allocated block into indexing block
    // Insert data into block
    do
    {
        hard_disk[(freeBlockNumber*ELEMENTSINBLOCK)+(dataIterator%ELEMENTSINBLOCK)] = data[dataIterator+1];
        ++dataIterator;
    } while (((dataIterator%ELEMENTSINBLOCK) != 0) && dataIterator < (dataCount-1));

    // Set end block if this is the last block allocated
    if (blockNum == requiredBlocks-1) { dirData[2] = freeBlockNumber; }
}
setDirStruct(dirData, hard_disk, directorySlot);
printf("File %d added!\n", data[0]);
++totalFiles;
return true;

```

Reading File

1. We iterate through the directory block 0 and 1 which is in the hard disk index 0 to 9 to find if there is such a filename. If there is, we then get the file directory and its data.


```

bool Read(int fileName)
{
    printf("\nReading file: %d...\n", fileName);
    char dirData[4];
    int directorySlot = 0;
    for (; directorySlot < DIRECTORYINDEX; ++directorySlot)
    {
        GetDirStruct(dirData, hard_disk, directorySlot);
        if (dirData[0] == fileName) { break; }
        memset(dirData, 0, 4);
    }
    if (directorySlot == DIRECTORYINDEX) { return false; }
}

```

2. For each index in the indexing block, get the block number written on each index entry and read each data in each element in the allocated blocks.

```

/*
    DirData[0] = Filename, DirData[1] = Starting block, Dirdata[3] = null terminator
    2) For each index in indexing block(until the first 0 index), go to that block of data.
*/

for (int index = 0; index < ELEMENTSINBLOCK; ++index)
{
    // Get block number written of each index entry
    int blockNum = hard_disk[(int)dirData[1]*ELEMENTSINBLOCK + index];
    if (blockNum == EMPTY) { break; } // No more indices left

    // Read each data in each element in the block
    printf("Contents: ");
    for (int elementNum = 0; elementNum < ELEMENTSINBLOCK; ++elementNum)
    {
        if (hard_disk[blockNum*ELEMENTSINBLOCK + elementNum] == EMPTY) { break; }
        printf("%d ", hard_disk[blockNum*ELEMENTSINBLOCK + elementNum]);
    }
    printf("\n");
}

return true;

```

Deleting File

1. We iterate through the directory block 0 and 1 which is in the hard disk index 0 to 9 to find if there is such a filename. If there is, we then get the file directory and its data.

```

bool Delete(int fileName)
{
    printf("\nDeleting File: %d...\n", fileName);
    char dirData[4];
    int directorySlot = 0;
    for (; directorySlot < DIRECTORYINDEX; ++directorySlot)
    {
        GetDirStruct(dirData, hard_disk, directorySlot);
        if (dirData[0] == fileName) { break; }
        memset(dirData, 0, 4);
    }
    if (directorySlot == DIRECTORYINDEX) { return false; }
}

```

2. For each block entry in the index, we delete each element by setting the element to 0. We get the block number written for each index entry and clear the bitmap.

```

/*
1) For each block entry in index, set each element to EMPTY
2) Clear the bit in bitmap
*/
for (int index = 0; index < ELEMENTSINBLOCK; ++index)
{
    // Get block number written of each index entry
    int blockNum = hard_disk[(int)dirData[1]*ELEMENTSINBLOCK + index];
    if (blockNum == EMPTY) { break; } // No more indices left







    for (int elementNum = 0; elementNum < ELEMENTSINBLOCK; ++elementNum)
    {
        hard_disk[blockNum*ELEMENTSINBLOCK + elementNum] = EMPTY;
    }
    clear_bit(&bitmap1, &bitmap2, blockNum);
    hard_disk[(int)dirData[1]*ELEMENTSINBLOCK + index] = 0;
}
clear_bit(&bitmap1, &bitmap2, (int)dirData[1]);
hard_disk[directorySlot] = 0;
--totalFiles;
printf("File %d deleted!\n", fileName);
return true;

```

3. User Manual

3.1 Files

In the submission, the files required for compilation and execution are as follows:

Name	Date modified	Type	Size
 input.csv	27/11/2021 4:37 PM	Comma Separate...	1 KB
 Linked.c	27/11/2021 4:36 PM	C Source File	15 KB
 Index.c	27/11/2021 4:36 PM	C Source File	12 KB
 CSVReader.c	27/11/2021 4:37 PM	C Source File	3 KB
 Contiguous.c	27/11/2021 4:37 PM	C Source File	12 KB
 CSVReader.h	27/11/2021 4:37 PM	C Header Source F...	1 KB

Main files

The following files contain the main program based on the allocation methods(contiguous, linked, and index). Compiling and executing them with an input file will yield an output based on the allocation method.

- 1) Contiguous.c
- 2) Linked.c
- 3) Index.c

Helper files

The following files contain some helper functions to improve readability and ease programming by dividing the program into smaller chunks. Mainly, the files listed reads CSV files and returns an array containing the operation, filename and data required.

- 1) CSVReader.c
- 2) CSVReader.h

Input files

Input files are files of comma-separated values format and provide the program with the operation to execute (Add, read, delete), the filename, and the data required for the operation. The file *input.csv* is identical to the provided sample file.

- 1) input.csv

3.2 Input

Input files are fed to a program which will be used to generate an output into the console. Input files are required to follow a specific format for the program to work as intended.

```
1  add,121,101,12,10,14,105
2  add,27,201,202,203
3  read,121
4  delete,27
5
```

General Format

- 1) Each line in the input file represents an operation to be executed along with the data required for that operation.
- 2) Each value in a line is delimited by a comma(,).

Operation-specific Formats

Add → Represented by the first value in the line as “*add*”, this operation adds a file and its contents into the hard disk. The first numerical value represents the name of the file to be added and subsequent values represent the contents of the above-mentioned file.

Read → Represented by the first value in the line as “*read*”, this operation reads a file of the specified filename in the hard disk and prints them into the console. The numerical value in the line represents the name of the file to be read.

Delete → Represented by the first value in the line as “*delete*”, this operation deletes a file from the hard disk. The numerical value in the line represents the name of the file to be deleted.

3.3 Compilation Instructions

Before executing each program, the source code must first be compiled and linked. Each of the **Main Files**(See Section 3.1) must be compiled and linked with **CSVReader.c** for it to execute as intended.

Contiguous

```
gcc -o contiguous.out Contiguous.c CSVReader.c
```

Linked

```
gcc -o linked.out Linked.c CSVReader.c
```

Index

```
gcc -o index.out Index.c CSVReader.c
```

3.4 Execution Instructions

To execute the program which reads an input file and outputs to the console, execute the file generated by the compiler and pass the input file name as a command line argument.

Contiguous

```
./contiguous.out input.csv
```

Linked

```
./linked.out input.csv
```

Index

```
./index.out input.csv
```

Note: *It is recommended that the user writes the output of the executable to a text file for ease of reading*

```
./index.out input.csv > IndexOutput.txt
```

3.5 Output

Each successful operation will generate an output to the console, while unsuccessful operations will print an error message. Outputs for all operations are similar, with the exception that **Read** operations will print the file name and its contents extracted from the simulated hard disk.

For the sake of simplicity, we will be using the output of a **Read** operation.

Note: The output of all operations are in the same format, with the exception that the **Read** operation's output has extra information.

3.5.1 Example Output

Example: Index Allocation method with sample input provided in submission file.

Reading file: 121... → Operation type and file name
Contents: 101 12 10 14 105 → Unique to **Read**. Contents of file 121 extracted from HD

***** HARD DISK INFO *****

TOTAL FILES: : 2 → Number of files currently in the HD
OVERALL SIZE: : 488 → Number of free *ints*(elements) post-operation in the HD

PRINTING HARD DISK MAP...

----- Directory Block 0 ----- → First Directory Block
Header Name Index → File headers (Differs based on Allocation Methods)
File Data : 121 2 → File related data(File name, Indexing Block)
File Data : 27 4
File Data : 0 0
File Data : 0 0
File Data : 0 0
----- Directory Block 1 ----- → Second Directory Block
Header Name Index
File Data : 0 0
File Data : 0 0
File Data : 0 0
File Data : 0 0
File Data : 0 0

----- File Block 2 ----- → Subsequent blocks are all data blocks...
|3 |0 |0 |0 |0

----- File Block 3 -----

|101 |12 |10 |14 |105

----- File Block 4 -----

|5 |0 |0 |0 |0

----- File Block 5 -----

|201 |202 |203 |0 |0

----- File Block 6 -----

|0 |0 |0 |0 |0

----- File Block 7 -----

|0 |0 |0 |0 |0

----- File Block 8 -----

|0 |0 |0 |0 |0

----- File Block 9 -----

|0 |0 |0 |0 |0

.....(***Continue printing***)

----- File Block 97 -----

|0 |0 |0 |0 |0

----- File Block 98 -----

|0 |0 |0 |0 |0

----- File Block 99 -----

|0 |0 |0 |0 |0

...PRINTING COMPLETE

***** HARD DISK INFO END *****