CPSC 321
Project Assignment 2

"A Study of File Systems and Allocation Methods"

Craig Lacey – 2300xxxxx
Email: laceyc@unbc.ca

Terry Regner – 2300xxxxx
Email: regnert@unbc.ca

University of Northern British Columbia
April 13, 2005

Table of Contents

Introduction

The goal behind CPSC 321 is to introduce the concepts and fundamentals of the structure and functionality of Operating Systems. There are four main components to the course: Scheduling, Deadlock detection, Memory management, and File Systems. The purpose of this project was to turn our attention to the topic file systems and methods of file allocation on disk space. These topics will be studied through the implementation of two different allocation schemes, one scheme implements the FAT style of allocation, and the other is a link list based approach.

Problem Statement

The problem assumes that we are looking at a disk space of 5MB with each file having a size in the range of 100k to 500k that remains constant throughout the simulation. Initially the file system will contain 10 files, with other files randomly being created and destroyed. The simulations file management should have the functions of file creation, file deletion, and file access settings.

File Allocation: In this operation, using the first linked list based approach we will be scanning through all the empty blocks to be allocated to the files. In the case of the FAT scheme will only be required to scan the File Allocation Table.

File Deletion: In this operation the linked list scheme requires that each of the pointers in the files blocks have their status updated with the deletion of the file to represent that these are now empty blocks. The FAT implementation requires that the pointers in the table be changed to represent the status of the corresponding blocks.

Read/Write Operation: For read/write operation we need to access the last block of the file, for the linked list approach this requires following the links to the last block, whereas the FAT scheme can allow us to go straight to the last block of the file.

Linked List Based File Allocation Scheme

The linked list based approach requires that each block of a file contains a pointer to the next block of data contained in that file. The block containing the final data of the file has its pointer to the next block assigned to NULL. When blocks are allocated/reallocated the pointers are updated according to the new memory structure.

The FAT Allocation Approach

This problem involves building and maintaining a table of all the pointers necessary to keep track of the blocks associated with a file. The allocation table contains each blocks pointer to the next block of information within that file, and also records the last block of the file having a pointer to NULL.

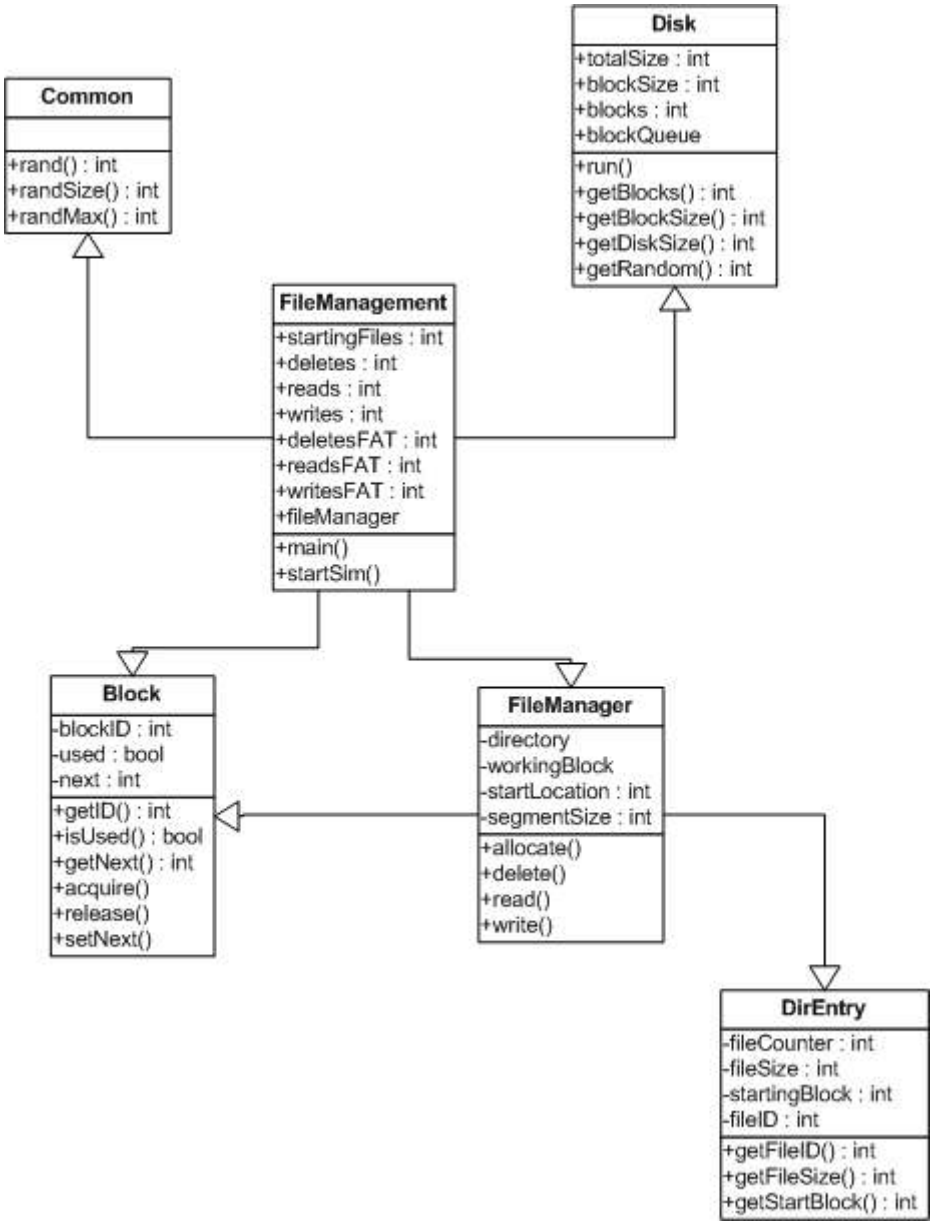
Solution Strategy

The first step in completing this assignment was a careful analysis of the problem description assigned. This involved looking for the objects, behaviors and properties that were detailed within the handout. All possible objects that were required for the implementation of the algorithms were recorded and then we were able to assign the properties and behaviors of each. The FileManagement class contains the main run method which contains the *startSim()* and the Disks *run()* function to allocate blocks to the disk and start the simulation running. The simulation then runs through the requirements specified for the project comparing the average time for reads writes and deletes across both schemes.

A detailed view of the classes implemented is provided in the following documentation along with their functions and interactions.

Solution Design

Class Diagram



Class Descriptions

Class FileManagement()
This is the driver class, it contains the main method, and initializes the disk and invokes the simulations run function. This class is also responsible for displaying the collected data in an easy to view format, for simple comparison of the two different schemes implemented.

Members:

```
public static int startingFiles;
public static FileManager fileManager;

public static int deletes = 0;
public static int reads = 0;
public static int writes = 0;

public static int deletesFAT = 0;
public static int readsFAT = 0;
public static int writesFAT = 0;
```

Methods:

public static void main(String[] args)
This is the main function, required by all programs, this function includes initialization of the number of starting files and the starting state of the disk.

public static void startSim()
This function starts the simulation with the parameters defined in the main function. It runs the simulation for both the linked list and FAT schemes.

Class FileManager

This is where most of the computation for the file allocation scheme comparisons takes place. The manager executes the required read, write, and delete functions as specified by the Problem Statement.

Members:

```
public static Vector directory = new Vector();

public static Block workingBlock = null;
public static DirEntry workingDir = null;
private static int startLocation = 0;
private static int segmentSize = 0;
```

Methods:

public static void allocate(int size)

The allocate() function allocates disk space to the files based on the two different schemes of memory allocation.

public static void delete(int ID)

The delete function removes a file from the system and updates the necessary pointers

public static void read()

This function was added to implement the possibility of future versions of the software being able to read. The read function is not currently necessary for this project.

public static void write()

Similar to the read function, the write is not necessary for the current project.

Class Disk

The disk class was created as a means of representing the structure of a disk for the simulation. The disk attributes were made private, and associated “get” functionality was implemented to retrieve the necessary information about the disk.

Members:

```
private static int totalSize;
private static int blockSize;
private static int blocks;
public static Vector blockQueue = new Vector();
```

Methods:

public static void run()
This function creates the starting state of the disk to be used in the simulation.

public static int getBlocks()
Returns the number of blocks on the disk

public static int getBlockSize()
Returns the size of the blocks on the disk

public static int getDiskSize()
Returns the size of the disk

public static int getRandom()
Used for random read/write/delete actions

Class Block

The block class was created to represent the logical structure of a block of memory on the disk used in the simulation.

Members:

```
private int blockID = -1;
private boolean used = false;
private int next = -1;
```

Methods:

public Block(int blockID)

The constructor initializes the ID of each block

public int getID()

Returns the ID of the block

public boolean isUsed()

Returns a Boolean value corresponding to whether or not the block is being used: true -> is being used false-> not being used

public int getNext()

This returns the ID of the next block in the files allocation, an ID of -1 was used to denote a NULL pointer.

public void acquire()

The acquire function of this class sets the blocks used status to true, indicating that there is file data being stored on this block that should not be overwritten

public void release()
The release method sets the used status of the block to false and updates the pointer to -1 indicating that this block is not being used to store file data

public void setNext(int next)
This method is used to indicate which block contains the next section of data in the current file.

Class DirEntry
This class is used to represent the files in the system, denoting the file size, the file ID and which block this file starts at. These file attributes were made private to simulate a certain degree of file information protection.

Members:

```
private static int fileCounter ;
private int fileID;
private int fileSize;
private int startingBlock;
```

Methods:

public DirEntry(int size, int start)
The constructor sets the fileID, fileSize and startingBlock members of the class object.

public int getFileID()
This function simply returns the files ID

public int getFileSize()
Returns the size of the current file

public int getStartBlock()
Returns the address of the starting block for this entry.

Class Common
A common class was developed as a means of putting commonly used functionality in one place. The common class is responsible for generating the random numbers that drive the simulations behaviour.

Members:

NONE

Methods:

public static int rand()
Generates a random integer

public static int randSize()
Used to generate a random size for file allocation

public static int randMax(int max)
Generates a large random integer

User Guide

For Windows:
Unzip the ”Project3.zip” file, extract the contents to a Directory of choice, then run Command Prompt, Change Directory to where the contents of the ”Project3.zip” are, and type the following:

1 cd \FileSystem \src\

Files included in this folder should be:
Block.java, Common.java, Disk.java, dirEntry.java, FileManager.java, FileManagement.java

2 javac -nowarn *.java
3 java FileManagement

For Linux/Unix:
Change Directory to where the contents of the ”Project3.tar” are, and type the following:

1 tar xvf Project3.tar
2 cd \FileSystem\src

Files included in this folder should be:
Block.java, Common.java, Disk.java, dirEntry.java, FileManager.java, FileManagement.java

3 javac *.java
4 java FileManagement

Results

Simulation Outputs

Output for a total of 10 files of random size.

```
**************************************************************************
*       Linked List                              FAT                     *
**************************************************************************
*  Reads Writes Deletion              Reads Writes Deletion               *
**************************************************************************
*  152939 149994 302485                10       10       10             *
Press any key to continue...
```

Output for a total of 25 files of random size.

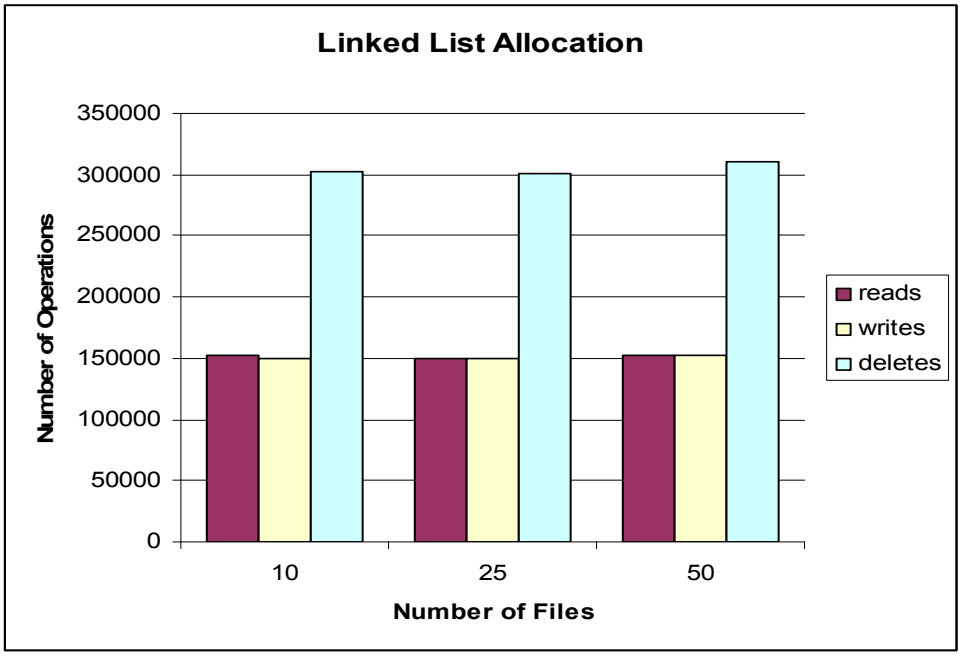
```
**************************************************************************
*          Linked List                           FAT                     *
**************************************************************************
*  Reads   Writes   Deletion          Reads   Writes   Deletion           *
**************************************************************************
*  149506 150033  301540               25         25         25          *
Press any key to continue...
```

Output for a total of 50 files of random size.

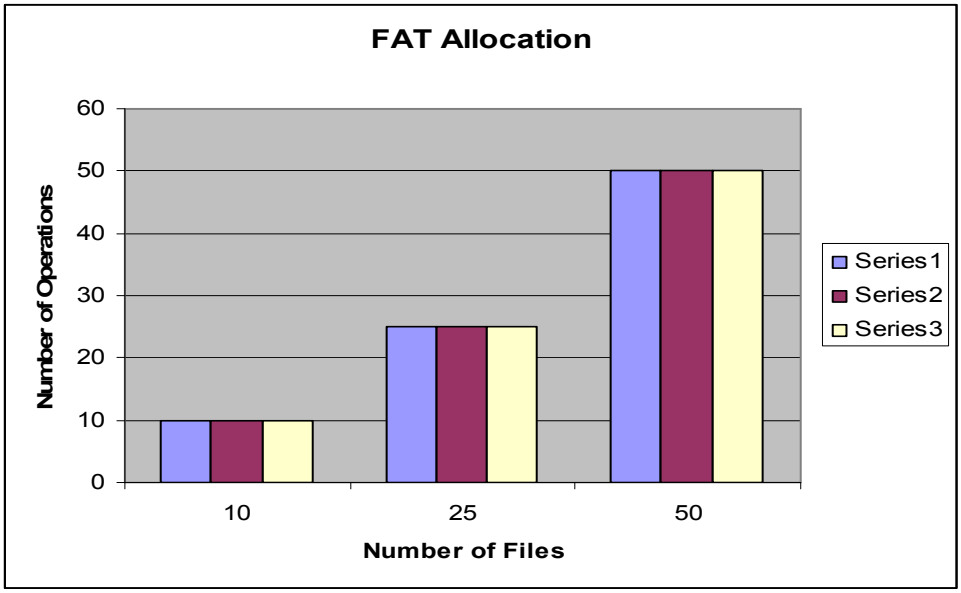
```
**************************************************************************
*       Linked List                      FAT              *
**************************************************************************
*  Reads Writes Deletion              Reads Writes Deletion       *
**************************************************************************
*  152911 153206 310032                50  50    50  *
Press any key to continue...
```

File Allocation Graph

The following graph is for the Linked List allocation of files on the disk for the simulation.



The following graph the results of the FAT allocation scheme for this project.



Analysis and Conclusion

Analysis

The Linked List allocation scheme required a tremendous number of operations which in a practical system would result in a huge overhead, these operations can be associated with the need for pointer maintenance on each block especially when a file is deleted. The contrasting number of operations for the FAT system shows that maintaining a table requires a significantly fewer number of operations on each file, as each read, write, and delete operation requires that only the tables pointers be updated once. The draw back of the FAT system is the storage space needed for keeping the allocation table in a separate logical file.

Conclusion

The amount of overhead seen for the Linked List scheme suggests that if you are implementing an allocation scheme for a system where processor time is at a premium that this scheme is not a wise choice. However, if processor time is not an issue, the scheme is a plausible one.

The FAT system is ideal for systems where CPU time is needed for more pertinent operations due to the lower number of operations needed for file operations. However if there exist memory space restrictions on the system and processor time is not at a premium then a Linked List implementation could be used effectively.