CS 3100/5100
Project #3

For project 3, you are to implement a file management system for a simulated disk drive. There will be two parts to this problem. For the first part, you must implement the routines necessary to record which tracks of the disk are in use and provide all of the functions necessary to allocate and release space; and for the second part, you must create all of the functions necessary to associate file names and sizes with their physical locations on the simulated disk.

PART I: Managing the storage allocation table

You must implement several C++ routines to manage the allocation of tracks on the simulated disk drive including: format, allocate, release, and table. The prototypes for these functions are shown below and the action that is performed by each function is described immediately following the prototype.

void FormatDisk (int Tracks); This procedure is invoked by the driver (our program) to inform you of the disk size (in tracks) and give you an opportunity to initialize your allocation tables. Disk tracks are numbered starting from zero so the range of valid addresses is 0...Tracks-1. Each track is exactly 128 bytes in length.

int Allocate (int& address, int size); This function is called to request a cluster of tracks of the given size. The function call:

// int rc; int address;
rc = Allocate (address, 100);

attempts to allocate a cluster of 100 tracks and returns the starting address of the cluster. The function returns a code indicating the success or failure of the operation. Return codes are as follows:

Success	rc = 0	
Invalid Size	rc = 1	// size < 1 or size > Tracks-1
Insufficient Space	rc = 2	

int Release (int address, int size); This function is called to release a cluster of tracks of length size, starting at the location specified by address. The function call:

int rc = Release (300, 100);

releases a cluster of 100 tracks starting at address 300. The function returns a code indicating the success or failure of the request. Return codes are as follows:

Success	rc = 0	
Invalid Size	rc = 1	// size < 1 or size > Tracks-1
Invalid Address	rc = 3	// address < 0 or address > Tracks-1

If number of tracks in the cluster is not correct, the return code should be Invalid Address (rc=3).

void Table(); This procedure is called to display the allocation table in a readable format. A suggested format is as follows:

```
                              Allocation Table
                 1         2         3         4         5         6
       0123456789012345678901234567890123456789012345678901234567890123
00000  1111111111111111100000000111111111110000000000100111111111100000
00064  0000000010000000000000000000000000000000000000111000000000001111
00128  1111111111111111111111111111111111111111111111111111111111111111
00192  0000000000000000000000000000000000000000011111111111111111101111
00256  1111111111111111000000000000000000000000000000111111000000000111
```

The numbers on rows represent addresses in increments of 64 and the numbers on the columns indicate additions to the row address. For example, at position 64 + 8 = 72 there is a 1 and at position 192 + 59 = 251 there is a value 0. The value 1 indicates the track at address 72 is allocated and the value 0 indicates that the track at address 251 is available. The value of the last row of the table is ((Tracks-1) / 64) * 64. In some cases, the last row may not be completely used.

You will want to use the Standard Template Library (STL) bitset to access a tightly packed array of bits To use the bitset class include “bitset” in your source code.

#include <bitset>

You can define a bitset as follows:

const int N = 256;
bitset<N> b;

In this example, the variable b contains a bitset with the bits tightly packed into bytes so the variable b requires only 256 bits / 8 bits per byte = 32 bytes of space (equivalent to 32 unsigned characters).

You may wish to define a space allocation class that uses a bitset to manage space on the simulated disk. This class provides support for routines Allocate, Release and Table.

PART II: Managing the file allocation table

To complete this assignment, you will have to write the additional functions, CreateFile, DestroyFile, FindFile, and ListDirectory, to associate file names with the allocated tracks.

int CreateFile (char* fileName, int fileSize, int& fileAddress); This procedure is called to create a file. The variable fileName points to a string of eight characters. The fileSize variable indicates the number of tracks requested. If you can satisfy the request, then store the file name, the address, and the size of the file in your allocation table and return the file address through the argument list. The return codes are the same as those used for the function Allocate with the addition of a return code of 4 to indicate that the file already exists

int DestroyFile (char* fileName); This procedure is called to remove a file from the file table and release the associated block of tracks. The fileName variable is a pointer to a file name (8 characters). The return codes are the same as those used for the release function with the addition of a return code of 4, if the file name does not exist.

int FindFile (char* fileName, int& fileSize, int& fileAddress); This functions is called to locate a file in the file table and return its address and size. The fileName variable is a pointer to a string of eight characters and the fileAddress and fileSize variables return the location and size of the file. The return codes are same as those used for the DestroyFile operation.

void ListDirectory(); This procedure displays the contents of the file management table in a readable (unsorted) format.

void Restart(int Tracks); This procedure is called to indicate a power failure has occurred and you must recover the file system from disk.

Implement your file directory using any technique you want such as hashing or higher ordered trees. Your solution must not have a fixed limit on the number items that can be stored in the directory. Keep in mind the following storage requirements. Each directory entry requires a key (8 characters) and two data fields (unsigned short integers) track address and file size. Also remember a disk track can hold only 128 bytes of information.

Keys are composed of upper and lower case alpha-numeric data values. Examples of valid keys are: FILE0020, datafile, program, and 93821123. If you use hashing, you may use any hash function you desire, but the function should map the key space uniformly into the table space.

You may wish to define a file allocation class to manage the file allocation task. This class would provide support for routines CreateFile, DestroyFile, FileFIle, ListDirectory and Restart. Please note these function must call the space allocation functions.

ACCESSING THE SIMULATED DISK

We will provide two functions that allow you to access the simulated disk: ReadDisk, and WriteDisk. The calling conventions for these functions are as follows:

```
int ReadDisk (unsigned char *, int);
int WriteDisk (unsigned char *, int);
```

For ReadDisk and WriteDisk, the first argument is a pointer to a buffer of characters (bytes) and the second argument is a disk address in the range of 0..Tracks-1. The following example demonstrates the use of these operations.

```
//Assume the track size is 80 bytes -- create a buffer of size 80 bytes
unsigned char buffer[80];
rc = ReadDisk (buffer, 50);   // Read track at address 50 into the buffer
rc = WriteDisk (buffer, 100); // Write buffer to the track at address 100
```

The read operation is nondestructive while the write operation overwrites the information currently stored at the target address. These operations return 0 on success and 1 on failure.

IMPORTANT RESTRICTIONS ON SPACE

In real life, file management software is part of the operating system; therefore, it must be efficient with respect to both time and space. To enforce this efficiency, you are limited to a total of 8 * 128 bytes of storage for variables. This means you should NOT use recursion in your solution. We will count the number of bytes of active storage used by variables in your program and if you exceed 1024, you will lose points. For example, an integer requires 4 bytes of storage and an array of 32 integers requires 128 bytes on our machine. Defined constants and enumerations are not included in this count and should be used freely to improve the readability of your code.

SUMMARY

The complexity of this assignment necessitates good communication between students and the instructor. Make sure to read Pilot news items regularly for updates on this assignment. To help you get started we have provided a file called stub.cpp that has function stubs for each of the operations we expect you to implement. ***DO NOT CHANGE THE NAMES OR PARAMETERS OF THESE OPERATIONS***! Your job is to fill in the function stubs with calls to your class member functions that implement the storage allocation table and the file allocation table.

To summarize, we are providing:
1. driver program (driver.h / driver.cpp) that calls the routines in stub.cpp
2. simulated disk drive (disk.h / disk.cpp)
3. a stub file (stub.cpp) to insure correct naming of routines
4. a make file which ***you will have to modify*** to compile your classes
5. a set of test files that test various aspects for your program

You will provide:
1. a completed stub.cpp which calls your class member functions as necessary
2. additional class headers and source code to implement a storage allocation table and a file allocation table
3. a modified makefile that builds the complete program

To test your compiled program, you will use one of our test files in the following form: `driver < testfile > outfile`

where testfile contains a sequence of instructions to the driver and the results are captured in the file outfile. If your program fails to operate correctly, submit a README documenting your problem.