Design of a Simple Block-Based File System

This assignment aims to design and implement a simple block-based file system. By using **bitmaps** and **block linked lists**, you'll manage the storage of files, implementing basic operations such as file creation and deletion, block insertion, defragmentation, and status viewing.

Basic Concepts

Block

A block is the smallest unit of data storage in the file system, imagined as a fixed-size storage space. A file's data is stored in one or more blocks.

Bitmap

A bitmap is a data structure used to represent the usage status of blocks, using binary bits to indicate whether each block is occupied:

- `0` indicates the corresponding block is free.
- `1` indicates the corresponding block is occupied.

Example:

Assuming the bitmap index ranges from `0` to `5`, where blocks `1`, `2`, and `5` are occupied, and the rest are free.

```
 Index:  0   1   2   3   4   5
Bitmap: [0] [1] [1] [0] [0] [1]
```

Block Linked List

A block linked list records the sequence of blocks occupied by a file, representing the storage order of the file's data. Each block contains an index pointing to the next block. The "next block index" of the last block is `-1`, indicating the end of the linked list. In this assignment, we use an **array** to simulate the block linked list. If you're familiar with the concept of linked lists, feel free to implement one.

Example: The file starts at block `1`, occupying block indices `1`, `3`, `5`.

- Block linked list array representation:

  ```
    Index:   0    1    2    3    4    5
  Content: [-1]  [3] [-1]  [5] [-1] [-1]
  ```

- The block linked list is represented as:

  ```
  [Block 1] -> [Block 3] -> [Block 5] -> -1
  ```

System Design

Initialization (INIT)

After the program starts, input an integer to specify the **total number of blocks** in the file system. Create a bitmap based on the total number of blocks, with all bits initially set to `0`. The system maintains a file list, recording each file's name and the starting block of its corresponding block linked list.

Create File (CREATE)

Create a file with a specified filename and required number of blocks. Allocate the necessary blocks, set the corresponding bits in the bitmap to `1`, and link the allocated blocks in order to form a block linked list. Add the new file's name and starting block to the file list.

Delete File (DELETE)

Delete a specified file, release the blocks occupied by the file, update the bitmap, and remove the file from the file list.

Append Blocks (APPEND)

Append a specified number of blocks to the **end** of a specified file. Allocate the necessary blocks, update the bitmap, and link the newly allocated blocks to the end of the file's block linked list.

Defragmentation (DEFRAG)

Due to frequent operations on file blocks, a file's block linked list can easily become fragmented (non-contiguous block indices), affecting read/write efficiency. The defragmentation operation reorganizes the file's block linked list using a "compact defragmentation" approach, which means moving all files' blocks forward as much as possible, concentrating free blocks at the back. This enhances the performance of the file system.

Example:

Before defragmentation:

- Bitmap status:

```
   Index:   0    1    2    3    4    5
  Bitmap:  [1]  [0]  [1]  [0]  [1]  [1]
Block LL:  [2] [-1]  [4] [-1] [-1] [-1]
```

- File `file1`'s block linked list is `[Block 0] -> [Block 2] -> [Block 4] -> -1`.
- File `file2`'s block linked list is `[Block 5] -> -1`.

After defragmentation:

- Bitmap status:

```
   Index:  0   1    2    3    4    5
  Bitmap: [1] [1]  [1]  [1]  [0]  [0]
Block LL: [1] [2] [-1] [-1] [-1] [-1]
```

- File `file1`'s block linked list is updated to `[Block 0] -> [Block 1] -> [Block 2] -> -1`.
- File `file2`'s block linked list is updated to `[Block 3] -> -1`.

Display Status (STATUS)

Display the current status of the file system, including each file's name and its corresponding block linked list. The output format is: `<filename> [<block index 1> <block index 2> ... -1]`.

Example:

```
file1 [0 1 2 -1]
file2 [3 -1]
```

Exit Program (QUIT)

Exit the file system program.

Block Allocation Strategy

To simplify the design, we adopt a strategy of allocating one block at a time, allocating blocks in order of their indices. In actual file systems, more complex allocation strategies are often used, such as **First Fit**, **Best Fit**, **Worst Fit**, etc. If you're interested, you can explore more on these topics.

Input and Output Format

The first line inputs the `INIT` operation, followed by an integer representing the total number of blocks `N` in the file system (`5 <= N <= 500`). Subsequent inputs are a series of commands until `QUIT` is input.

Command Format

- `CREATE <filename> <number of blocks>`
  - If the filename already exists, output `<filename> EXISTED`.
  - If insufficient system blocks are available, output `<filename> INSUFFICIENT BLOCKS`.
- `DELETE <filename>`
  - If the file does not exist, output `<filename> NOT FOUND`.
- `APPEND <filename> <number of blocks>`
  - If the file does not exist, output `<filename> NOT FOUND`.
  - If insufficient system blocks are available, output `<filename> INSUFFICIENT BLOCKS`.
- `DEFRAG`
  - Rearrange file blocks to concentrate all files' blocks forward and free blocks at the back.
- `STATUS`
  - Output in the order files were created.
  - Output format: `<filename> [<block index 1> <block index 2> ... -1]`.
  - If a file has no blocks, output `<filename> [-1]`.
  - If the number of files is `0`, output `EMPTY`.
- `QUIT`
  - Exit the program.

Note:

- All inputs are in valid formats; no need to check for invalid inputs.

- Filenames do not exceed 10 characters and contain no special characters.
- The number of blocks is a non-negative integer (`>=0`).
- The total number of created files does not exceed `100`.

Examples

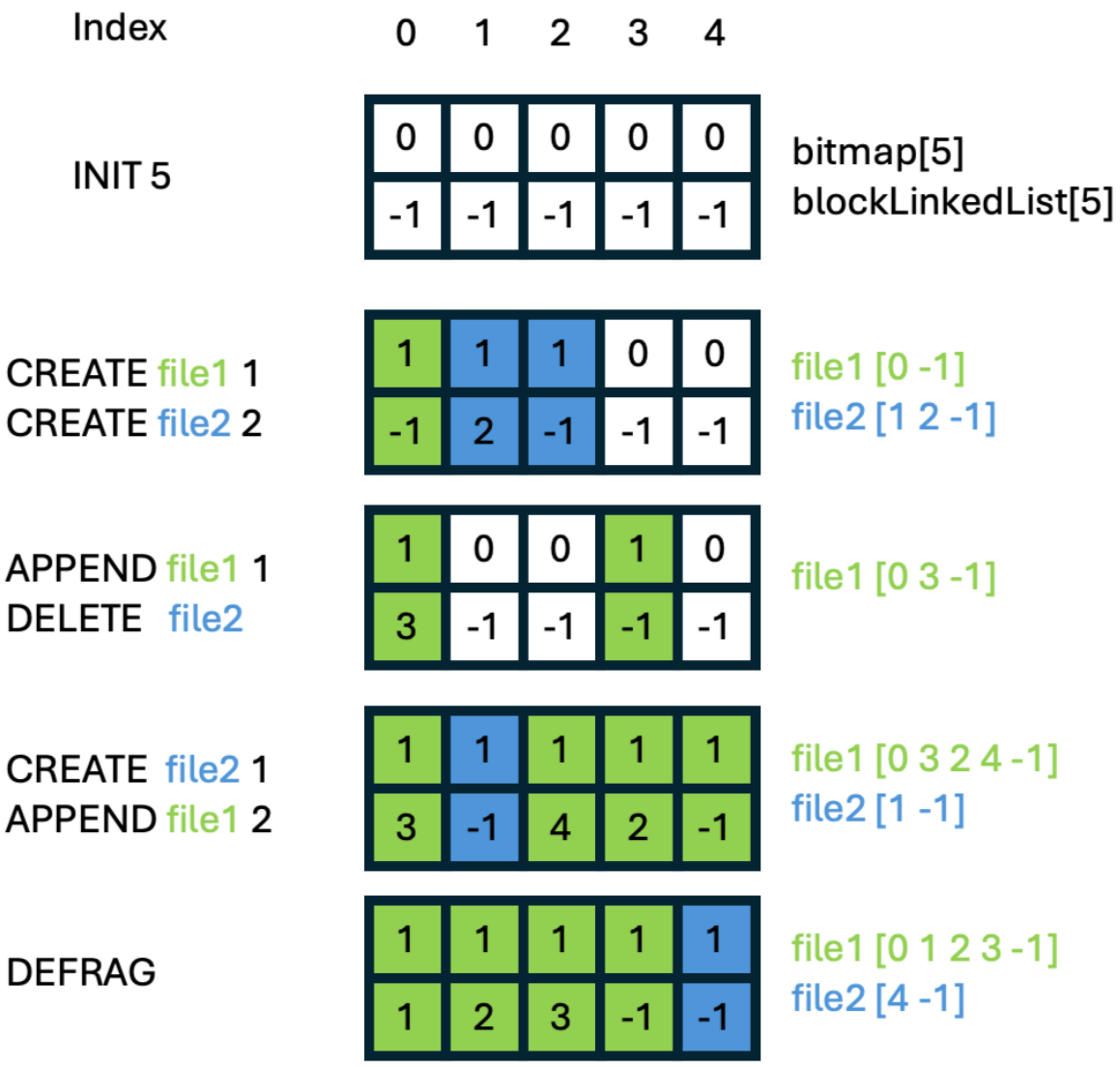
Sample Input 1:

```
INIT 5
CREATE file1 1
CREATE file2 2
STATUS
APPEND file1 1
DELETE file2
CREATE file2 1
APPEND file1 2
STATUS
DEFRAG
STATUS
QUIT
```

Sample Output 1:

```
file1 [0 -1]
file2 [1 2 -1]
file1 [0 3 2 4 -1]
file2 [1 -1]
file1 [0 1 2 3 -1]
file2 [4 -1]
```

Explanation for Sample 1:



Sample Input 2:

```
INIT 5
CREATE file1 10
STATUS
APPEND file1 1
CREATE file1 5
DELETE file1
CREATE file1 0
STATUS
QUIT
```

Sample Output 2:

```
file1 INSUFFICIENT BLOCKS
EMPTY
file1 NOT FOUND
file1 [-1]
```

Sample Code

The sample code is provided for reference implementation. You can implement it according to your own logic or modify it as needed.

```cpp
#include <iostream>

using namespace std;

const int MAX_FILES = 100;
const int MAX_BLOCKS = 300;
const int MAX_FILE_NAME = 11;

////////////////// COMMANDS //////////////////
/**
 * Initializes the file system with a specified number of blocks.
 * This function sets up the initial state of the file system, including
 * the bitmap and block linked list.
 * @param n - The total number of blocks available in the file system.
 */
void initFileSys(int n);

/**
 * Creates a file with a specified name and required block count.
 * This function allocates the required blocks and links them for the file.
 * If there are insufficient blocks or if the file already exists, it should
 * handle these cases.
 * @param filename - The name of the file to create.
 * @param blockCount - The number of blocks required for the file.
 */
void createFile(const char *filename, int blockCount);

/**
 * Deletes a file by name, freeing its allocated blocks.
 * This function releases the blocks used by the specified file,
 * updates the bitmap, and removes the file from the file list.
 * If the file is not found, it should handle this case.
 * @param filename - The name of the file to delete.
 */
void deleteFile(const char *filename);

/**
 * Appends a specified number of blocks to an existing file.
 * This function allocates new blocks and adds them to the end of the file's
block chain.
 * If the file does not exist or there are insufficient blocks, it should handle
these cases.
 * @param filename - The name of the file to which blocks will be appended.
 * @param blockCount - The number of additional blocks required.
 */
void appendBlocks(const char *filename, int blockCount);

/**
 * Defragments the file system by reorganizing blocks to be contiguous.
 * This function attempts to remove fragmentation by moving all used blocks to
the
 * beginning of the block space, thereby consolidating free space at the end.
 */
void defragment();

/**
 * Displays the current status of the file system.
```

```
 * This function outputs the block chains of all files in the system in creation
order.
 * If there are no files, it outputs "EMPTY". Each file's block chain is shown
in
 * the format <filename> [block1 block2 ... -1].
 */
void displayStatus();

////////////////// HELPER FUNCTIONS /////////////////
/**
 * Finds the index of a file in the file list by name.
 * This function helps check if a file exists and locate its position
 * within the file list.
 * @param filename - The name of the file to locate.
 * @return - The index of the file in the file list, or -1 if not found.
 */
int findFile(const char *filename);

/**
 * Allocates a specified number of free blocks.
 * This function searches for available blocks in the bitmap, allocates them,
 * and updates the bitmap to reflect the allocation.
 * @param allocatedBlocks - An array to store the indices of the allocated
blocks.
 * @param blockCount - The number of blocks to allocate.
 * @return - True if the required blocks are successfully allocated, otherwise
false.
 */
bool allocateFreeBlocks(int *allocatedBlocks, int blockCount);

/**
 * Appends a series of blocks to the end of an existing file's block chain.
 * This function links the new blocks to the end of the specified file's
existing block chain.
 * @param newBlocks - An array containing the indices of the new blocks to
append.
 * @param newBlockCount - The number of new blocks to append.
 */
void appendToBlockChain(const int *newBlocks, int newBlockCount);

/**
 * Retrieves the full block chain of a file starting from a specified block.
 * This function follows the block links from the start block, recording each
block in the
 * chain, up to the end block (indicated by -1).
 * @param blockChain - An array to store the indices of blocks in the chain.
 * @param startBlock - The starting block of the file's chain.
 */
void getBlockChain(int *blockChain, int startBlock);

////////////////// FILE STRUCTURE /////////////////
/**
 * File structure representing a single file within the file system.
 * - name: The name of the file (up to 10 characters).
 * - headBlock: The first block in the file's block chain.
 * - tailBlock: The last block in the file's block chain.
 * - blockNum: The total number of blocks currently allocated to the file.
 */
```

```cpp
struct File {
    explicit File() : name(), headBlock(-1), tailBlock(-1), blockNum(0) {
        name[0] = '\0';
    }

    char name[MAX_FILE_NAME];
    int headBlock;
    int tailBlock;
    int blockNum;

    // TODO: If you want to use `cout<<file` to output a File object, you should be able to do so with the following code:
    friend ostream &operator<<(ostream &os, const File &file) {
        //......
        return os;
    }
};

int totalBlocks;             // Total blocks in the file system
int freeBlockCnt;            // Count of free blocks currently available
unsigned char bitmap[MAX_BLOCKS]; // Bitmap to manage block allocation
int blockLinkedList[MAX_BLOCKS];  // Array to represent the block links (linked list)
File fileList[MAX_FILES];    // Array to store all file structures
int fileCount = 0;           // Current number of files in the file system

int main() {
    // TODO: Implement the main logic of the file system
}

// TODO: implement the command functions here

// TODO: implement the helper functions here
```