Scheduling 3: Buffer Pool and Memory Manager

Assignment:

For this project, you will implement a buffer pool and basic best-fit memory manager. Buffer pools are described in Section 8.3 of the textbook, and the best-fit memory manager in Section 12.3.1. The data managed by the program are strings, and these will be stored on disk in a binary file. All access to the file will be mediated by a buffer pool. The memory manager will decide where in the file to store the strings.

Input and Output

There will be two input parameters to the program. The first will be the name of a file that will be used to store the string data. The second will be the number of buffers in the buffer pool. Your program will also read a command file from standard input (`stdin`) and write responses to the commands to standard output (`stdout`). The input command file conains a series of commands (some with associated parameters, separated by spaces), one command for each line. No command line will require more than 80 characters. Commands are free format in that an arbitrary number of additional spaces may be interspersed between parameters, and blank lines may appear between commands. You do not need to check for syntax errors in the command lines.

Each input command should result in meaningful feedback in terms of an output message. Each input command should be echo'ed to the output.

Commands and their syntax are as follows.

insert *ID*
One or more string lines
blank line

The basic entities being stored are strings. Each string has an ID number associated with it, and the string itself. The value of ID must be in the range 0 to 999. Strings may require multiple lines in the input file. You should preserve all white space and newlines in the input string. Each input command is terminated by a line with nothing but white space (any extra spaces on the blank line should not be stored as part of the string).

If an ID number is given that is currently being used, then the existing message is deleted from the memory manager, and the new message is inserted. For example, if a command is given to store a string with ID 5, and then another command is given to store another string with ID 5, the result is to tell the memory manager to release the space for the first message, and then to insert the second message.

print *ID*

Print the string with the given *ID*.

remove *ID*

Remove the string with the given *ID* from the database.

dump

Print out the free block list. You should print the blocks in the order they appear on your list, and for each block you should show its size and position in the file.

Implementation:

The strings will be stored in the disk file. The disk file is only accessible through the buffer pool. The memory manager will decide exactly where the strings go in the file. The buffer pool will manage the file in blocks of 512 bytes each.

A key consideration for your implementation is how the memory manager's client interacts with the memory manager to store and retrieve its messages. Your memory manager should take a message and its size as input, and give back a **handle**. When the client wants the contents of a stored message back from the memory manager, it gives the handle (that it has stored) to the memory manager, along with a suitable buffer in which the memory manager can copy the contents of the message. The handle will contain the byte position in the file for the message (although only the memory manager, not the client, actually knows this detail).

To manage access to the strings at the command level, there will be a simple array of 1000 positions. When a string is given to the memory manager to store, the memory manager returns the handle to the string with which to retrieve the string at a later time. These handles will be stored in the array, in the slot defined by the ID.

For example, assume the command is given to store string "hello" with ID 23. We will assume for this example there were no extra spaces on the command line containing "hello". The memory manager will be asked to find space for the 6-byte string (the 5 letters and the newline character). Once the memory manager decides on a position, it passes the actual string on to the buffer pool (and tells the buffer pool where the string goes). The memory manager then creates and returns to its caller a handle object This handle is then placed in slot 23 of the array for our example.

To recover a message, the memory manager actually needs to know the starting position of the message and the length of the message. The memory manager will get the starting position from the handle. The memory manager should put the length of the message at the beginning of the message, storing this as a four-byte quantity. For example, if the message from our example is 6 bytes long, then the memory manager will locate the smallest free block that is at least 10 bytes long. It will store the message length in the first four bytes, and then the message itself in the remaining 6 bytes. Any additional space in the free block beyond 10 bytes will be retained as a free block. For example, if the smallest free block is 40 bytes long, then 30 bytes will be returned to the free list.

If a request is made for a free block that cannot be serviced by the existing pool of free blocks, then new space is allocated at the end of the file, causing the file size to grow as necessary. It is acceptable to allocate an entire block (equal to a buffer pool's buffer size of 512 bytes) or multiple blocks (when necessary) to service the request, but no more blocks than necessary should be allocated. Any additional space allocated beyond that needed to store the message would be added to the free block list. For example, if the block (buffer) size is 512 bytes and the message (including 4 bytes for the length) is 300 bytes long, then the remaining $512 - 300 = 212$ bytes should make up a new free block. Be careful that any existing free block at the end of the file is used as part of this allocation. For example, if the last 50 bytes of the file are free, the message is 100 bytes long (including 4 bytes for the message length), and blocks (buffers) are 512 bytes, then a new block will be allocated, yielding 562 total free bytes at the end of the file. 100 of those bytes are allocated to store the message, leaving a free block of 462 bytes.

The memory manager will be implemented as a simple sequential-fit memory manager using the "best-fit" strategy for selecting a free block. The memory manager will maintain a linked list of free blocks in memory (you may use an STL list object to manage the free list if you wish). Any given request for space will be filled by the smallest available free block that can store the request (if there is extra space in the free block, this is returned to the free block list).

Your memory manager must merge adjacent free blocks together whenever space for a message is freed by the client. You will need to work out some suitable approach for doing so, by examining the current free blocks in the free block list to determine if the newly freed block is adjacent to one or two existing free blocks.