Report
Assignment 4

Remplacementmanager.h for a C++ implementation of a memory manager. It contains the class declaration for MemoryManager which has methods for allocating and deallocating memory, as well as methods for managing handles and defragmentation.

The header file defines a Block struct which represents a block of memory. It contains a boolean indicating if it's free or not, a boolean indicating if it's locked or not, the size of the block, and a pointer to the next block.

The MemoryManager class has a private member variable head which is a pointer to the first block of memory. It also has four private vectors smallList, mediumList, largeList, and veryLargeList, which are used for storing free blocks of different sizes to improve allocation performance. The class also has a public boolean verbose which can be set to true to enable verbose output for debugging.

The MemoryManager class has methods for first fit, best fit, and worst fit allocation strategies, as well as a smart fit strategy which selects the most appropriate allocation strategy based on the size of the requested memory. There are also methods for deallocating memory, printing a map of the heap, and managing handles.

The MemoryManager class also has methods for defragmentation, which moves all free blocks to the end of the heap and merges adjacent free blocks. There are also methods for checking for locked blocks and for locking and unlocking handles.

replacementManager.cpp a simple memory manager that allocates memory using three different strategies: first-fit, best-fit, and worst-fit. The memory is divided into blocks, each represented by a struct containing a flag indicating whether the block is free or allocated, the size of the block, and a pointer to the next block. When a request for memory is made, the memory manager scans through the linked list of blocks and looks for a block that is free and has enough space to accommodate the request. Depending on the strategy used, the memory manager selects the block that is best suited to fulfill the request.

The memory manager also supports the concept of handles. A handle is essentially a pointer to a block of memory that has been allocated using the memory manager. When a handle is created, the memory manager allocates a block of memory of the requested size using the selected strategy and returns a pointer to the beginning of the block. The memory manager keeps track of all the handles that have been created, so that they

can be deallocated later. When a handle is deallocated, the memory manager frees the associated block of memory and removes the handle from its list.

The code also includes a helper function, printHeapMap(), that prints out the current state of the memory map, showing the addresses, sizes, offsets, and statuses of all the blocks of memory. This can be helpful for debugging and understanding how the memory manager is working.

replacementmalloc.h
This header file defines the interface for the replacement memory allocation functions. It declares functions for allocating, freeing, and manipulating memory blocks, as well as functions for managing memory handles and performing defragmentation.

The functions declared in the header file are:

myMalloc(size_t size): Allocates a block of memory of the specified size and returns a pointer to the start of the block.
myCalloc(size_t n, size_t size): Allocates a block of memory of the specified size and initializes all bytes to zero. Returns a pointer to the start of the block.
myFree(void* ptr): Frees a previously allocated block of memory pointed to by ptr.
verbose(bool v): Sets the verbosity level of the memory manager to v. If v is true, the memory manager will print out a heap map after each allocation and deallocation operation.
makeHandle(int n, size_t size): Allocates a block of memory of the specified size and associates it with a handle n. Returns the handle.
smartMakeHandle(int n, size_t size): Allocates a block of memory of the specified size and associates it with a handle n. Returns the pointer to the start of the block.
freeHandle(Handle h): Frees the block of memory associated with the handle h.
myDefrag(): Defragments the heap by coalescing adjacent free blocks.
smartFit(size_t size): Allocates a block of memory of the specified size using a "smart" algorithm that selects the best-fit free block.
smartFree(void* ptr): Frees a previously allocated block of memory pointed to by ptr. If the freed block is adjacent to another free block, the two blocks are coalesced.
This header file is intended to be included in client code that wishes to use the replacement memory allocation functions. By including this header file, client code can call the functions declared in the header file without having to know the implementation details of the memory manager.

Replacementmalloc.cpp provides an implementation of the memory management functions myMalloc, myCalloc, and myFree, as well as several other functions related to memory management.

The implementation uses an instance of the MemoryManager class, which is defined in a separate header file replacementManager.h. This class contains the implementation of the memory management algorithms, including first fit, best fit, and worst fit.

The myMalloc and myCalloc functions both call one of the memory management algorithms based on a preprocessor macro FIT_CHOICE that is set at compile time. The myFree function deallocates memory previously allocated by myMalloc or myCalloc.

In addition to these basic memory management functions, the implementation provides several other functions, including verbose, which enables/disables verbose output during memory management operations, makeHandle and freeHandle, which allow the creation and destruction of handles that can be used to reference allocated memory, myDefrag, which defragments the memory heap, and smartFit and smartFree, which provide a more intelligent allocation and deallocation strategy based on handle usage patterns.

Test First Fit vs Best Fit:
Allocate blocks of varying sizes using both first fit and best fit algorithms and compare the number of blocks allocated, as well as the fragmentation of the heap.
RunSequences/runSeq1.txt
RunSequences/runSeq2.txt

Test Failure Handling:
Allocate more memory than the heap can hold and verify that the memory manager properly handles the failure. For example:
RunSequences/runSeq3.txt

Test Defragmentation:
Allocate several blocks of memory and then free some of them, causing fragmentation. Then call the defrag function and verify that the heap is defragmented. For example:
RunSequences/runSeq4.txt

Test calloc:
Allocate a block of memory using calloc, verify that the block is cleared, and then free it.
RunSequences/runSeq4.txt

Test Handle Creation:
Allocate several blocks of memory and then create handles for them using the makehandle function. Verify that the handles point to the correct blocks and then free them using the handle.
RunSequences/runSeq5.txt