C347 – Operating Systems

Mini-Project

Customised Memory Manager in C++

NAMES:

P.N. Aditya	-150070044
Saqib Azim	-150070031
Ravi Kumar Kushawaha	-150070045

Introduction

We often use variable sized arrays created in the stack part of memory whenever we wish to store large amount of data of some type. However, this is only possible if we (read compiler toolkit) knows prior to the execution of statement how space is required. However, most of the times in programming we don't know how much space is required to store our variables i.e. we need to resort to dynamic allocation of memory. As and when new data is required to be added viz. during the expansion of a binary tree, we make use of pre-defined C++ library functions such as new/malloc. Also when we have to free some space which is more than what is required or it is no longer required and we wish to return it to the OS so that it is available for future allocations we use delete/free.

However, in these standard functions the problem (for heavily heap dependant programs) is that they are general-purpose memory allocators. Even if one's code is single-threaded, the malloc it is linked to can handle multi-threaded paradigms just as well. It is this extra functionality that degrades the performance of these routines. Moreover, the malloc and new functions make calls to the Operating System kernel requesting memory, while free and delete make requests to delete memory. This means that the operating system has to switch between user-space code and kernel code every time a request for memory is made. Programs making repeated calls to malloc or new eventually run slowly because of the repeated context switching.

Memory that is allocated in a program and subsequently not needed is often unintentionally left undeleted, and C/C++ don't provide for automatic garbage collection. This causes the memory footprint of the program to increase. In the case of really big programs, performance takes a severe hit because available memory becomes increasingly scarce and hard-disk accesses are time intensive.

Thus, in this basic learning project of ours we have tried realise in practice how a basic memory manager for a single data type, in our case it is Complex, can be made. We write 2 identical memory allocation programs, one which uses the default compiler-provided allocator and the other in which we insert a custom defined memory manager. We wish to time the two programs and check whether the latter gives better results or not.

Design Goals

Speed:
The memory manager must be faster than the compiler-provided allocators. Repeated allocations and de-allocations should not slow down the code. If possible, the memory manager should be optimized for handling certain allocation patterns that occur frequently in the code.

Robustness:
The memory manager must return all the memory it requested to the system before the program terminates. That is, there should be no memory leaks.

User convenience:
Users should need to change only a minimal amount of code when integrating the memory manager into their code.

Portability:
The memory manager should be easily portable across systems and not use platform-dependant memory management features.

General Approaches for creating a memory manager

Requesting large memory chunks

One of the most popular memory management strategies is to request for large memory chunks during program start-up and then intermittently during code execution. Memory allocation requests for individual data structures are carved out from these chunks. This results in far fewer system calls and boosts the performance time.

Optimisation for common request sizes

In any program, certain specific request sizes are more common than others. A memory manager will do well if it's optimized to handle these requests better. In this case the manager knows how much to allot and not carry the overhead of first determining the size to be allotted.

Pooling deleted memory in containers

Deleted memory during program execution should be pooled together. Further requests from memory should then be served from these containers. If a call fails, memory access should be delegated to any one of the large chunks allocated during program start. While memory management is primarily meant to speed up program execution and prevent memory leaks, this technique can potentially result in a lower memory footprint of the program because deleted memory is being reused.

Details of our programming exercise

Our program basically entails a simple allocation and de-allocation scheme. We created a class named Complex (having 2 data members). Then we proceed to perform a relatively huge number of repeated allocation and de-allocation cycles. Precisely, we perform 1000 rounds of 20000 new and delete operations for the complex data structure.

Thus, in case of a compiler-allotted allocator this would lead to a total of 1000*20000*2= 40 million context switches between the kernel code and user code which is quite high and hence we take this up as an exercise to reduce the number.

In order to create a custom memory manager for the Complex class that improves the compiler implementation, we need to override the Complex class-specific new and delete operators.

We start our program by creating a data structure named “Block” which stores the pointer to another block. Essentially we create a linked list of blocks which are going to store our future heap data.

We create an abstract class named ”GenMag”. Sometimes implementation of all functions cannot be provided in a base class because we don’t know the implementation. Such a class is called an abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn’t have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes. However, we can create pointers to an abstract class. In our case the various derived classes would be those of Memory Managers of various data-types. We declare two “pure” virtual functions named allocate and free in it b assigning ‘0’ to them.

Concept of virtual functions is as follows. Let us suppose we create a pointer to an abstract class. When a virtual function is called upon this pointer then the actual function execute depends on the address of which of its derived classes it is holding. Eg: If pointer to GenMag holds address of the memory manager class of char type, then <u>during execution,</u> the overriding (over that defined in the abstract class) function executed is that of “char” and not that of “Complex”, “int” , or any other user defined Objects for which we may have created customised memory managers. This is what distinguishes “Late binding” from “Early binding”. Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding (Compile time) is done according to the type of pointer.

Now, we move onto the next class we have defined Complex_mem_mag which is derived from GenMag. In this we have the functions expandPoolSize, Cleanup and BlockHead.

Let us consider expandPoolsize(). This is called whenever a Complex_mem_mag type variable is defined as part of constructor definition or when the allocated pool of memory has been used up and hence again a new pool needs to be obtained. The latter is implied when the pointer to the chunk which has to be allocated next when new is called is “NULL”. This is the crux of the customised allocator. The Block data structure needs to hold both the pointer to the next block as well as needs to contain the data-type which has been allocated. Hence, the minimal space which needs to be allotted to each Block pointer is max(size of data-type, size of block pointer). After this comparison, the “Blockhead” member variable of Complex_mem_mag is allotted this much amount of space using the **compiler-alloted** new function on a character array (character array is chosen because each character is of 1byte and hence the required space can be directly obtained by using size as argument in “new char[size]” where size is the minimal thing mentioned above). Since, this is a pointer to a char array it has to be “reinterpret_cast”(ed) to a pointer of type “Block”. We create a pool size equal to 32 * **size** bytes. Initially, before a chunk (equal to size of complex data) is allotted to any variable it contains the “next” member variable which is the pointer to the next chunk. The last chunk of “Block” has the next pointer assigned the value “NULL” indicating no more memory is available further.

Back to Complex_mem_mag. Its destructor consists of the function “Cleanup” which returns back to OS the pool that it received by de-allocating all chunks using the compiler-allotted delete function.

Allocate function expands poolsize if required i.e. when blockhead is pointing to NULL. After this it returns the pointer to the chunk corresponding to the blockhead and assigns the new Blockhead to be the pointer to next chunk.

Free function also does something which is interesting. If the space allotted to a pointer is to be deleted, it is called. What it does is that it makes the deleted chunk as the new BlockHead and makes the chunk which was initially (before delete) the head to be pointed by the “next” member in the new Blockhead. Essentially what we are doing is that we are pooling together deleted space and reuse it again instead of going further ahead. We are recycling!!

Finally, the new and delete overloaded operators are just wrappers for the Complex_mem_mag routines. The allocate, free routines are inline to avoid function call overhead. Essentially the function is placed directly after the caller function in Instruction memory.

Evaluation

When we execute the program which uses default functions, the runtime turns out to be around 3.5 seconds. On the other hand, the same program with the customised memory manager incorporated in it takes 0.5 seconds which is less than the former by around 7 times. Thus, our manager worked!!

This improvement can be explained by the following points:

1) The number of switches between the user and kernel code has been reduced markedly because it reuses the deleted memory by pooling it back to the free-store.
2) This memory manager is a single-threaded allocator that suits the purposes of the execution. We don't intend to run the code in a multithreaded environment, yet.

The issue in this manager is that if a Complex object created using new is not deleted, then there's no way to reclaim the lost memory in this design. This is, of course, the same if the developer uses the compiler-provided new and delete global operators. However, a memory manager is not just about performance improvement, it should also be able to remove memory leaks from the design. The cleanUp routine is only meant to return memory back to the operating system for cases in which the Complex object created using new has been explicitly deleted. This problem can only be resolved by requesting bigger memory blocks from the operating system during program initialization and storing them. Requests for memory using Complex's new operator should be carved out from these blocks. The cleanUp routine should work on freeing up these blocks as a whole, not individual objects that are created from these memory blocks.

Reference :
IBM Developer Works project, published by Arpan Sen and Rahul Kardam