CS347 - Mini-Project Abstract

Building a memory manager for an application

Project Abstract : In this project we are going to learn why a memory manager is needed and how one can write a customized manager for a simple c++ programs.

Introduction :

The control over memory allocation helps code run faster.The standard library functions malloc, free, calloc, and realloc in C and the new, new [ ], delete, and delete [ ] operators in C++ form the crux of the memory management. The general-purpose memory allocator functions such as malloc and new can handle multi-threaded paradigm even if one's code is single-threaded. This extra functionality degrades the performance of these routines. In their turn, malloc and new make calls to the operating system kernel requesting memory, while free and delete make requests to release 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

Design Goals:

- Try to increase the speed of execution of a specific program relative to what we get based on the default compiler provided allocator
- The memory manager should return all the memory it requested from the system after the program terminates i.e. we wish to prevent memory leakage
- User needn't change the code much to integrate the memory manager
- Memory manager shouldn't use platform dependant features, hence we wish to make it portable

Method :

We are going to follow certain useful strategies while creating a memory manager

- Request large memory chunks.
- Optimize for common request sizes.
- Pool deleted memory in containers.

Creating a BitMapped Memory Manager:

It is a kind of refinement over the original fixed-size memory allocation scheme.In this scheme, memory is requested from the operating system in relatively big chunks and memory allocation is achieved by carving out from these chunks. Deallocation is done by freeing the entire block at a single go. In this approach, a large chunk of memory is requested by the memory manager. This chunk is further subdivided into smaller fixed-size blocks. In this case, the size of each of these blocks is equal to the size of the T object.If the memory manager runs out of free blocks, it makes further requests to the operating system for large memory chunks. Every bitmap in the MemoryManager data structure is meant to augment the corresponding memory chunk. However, when a delete is called, the corresponding block is made available to the user. Thus non-sequential delete calls create what is known as *fragmented memory* and blocks of suitable size can be provided from this memory.

Thus, In our project we will try to understand he following concepts :

- The need to have a memory manager in user code.
- The design requirements for a memory manager.
- How to create a fixed-size allocator or memory manager.
- How to create a variable-sized allocator.
- Memory allocation in a multi-threaded environment.

Citation : https://www.ibm.com/developerworks/aix/tutorials/au-memorymanager/index.html#list2

Team Members:

1) P.N.Aditya-150070044

2) Ravi Kumar Kushawaha-150070045

3) Saqib Azim – 150070031