AN APPROACH TO IMPROVE THE EFICIENCY OF PRIORITY QUEUE IMPLEMENTATIONS

Dhananjay Gupta[1], Avani Kothari[2], Neetesh Gupta[3], Akshay Khandelwal[4]

[1,2,3,4] *Senior Undergraduate, Department of Information Technology, Shri Vaishnav Institute of Technology and Science, M.P., India*

ABSTRACT

This paper is an extension to our previous work [1]. Based on their running time efficiencies, earlier we gave a comparative study of various available ways to implement priority queues. In this paper we demonstrate a way to further improve their running time efficiency. We bring an introduction to memory pooling and later describe how this concept may be used to improve the running time efficiency of priority queues. Priority queues involve a large number of system calls during memory allocation. This leads to extra time being required for node generation. However, this problem can be solved by using a already available memory pool. For the purpose of memory allocation, a pre assigned pool of memory could be used instead of making system calls on every node generation. This approach significantly reduces the time frame of program execution. Although no changes in algorithm for priority queues have been made, this approach works on the system level and is very beneficial when a very large number of data nodes have to be created. Likely so, in our previous work [1] we had shown that we already have the constant time executable algorithms available. So with this approach we go out of the box to improve the efficiency of priority queues.

Keyword: - *Priority queues, Memory pooling, efficiency, running time, complexity analysis.*

1. INTRODUCTION

After having been established that priority queue works best with heaps its time to answer: is there any way we may further improve the efficiency of priority queue implementations. The answer to that question is fortunately a yes. When making a system call for memory allocation a switch from user code to kernel code is required. It has certain tolls mainly the time required to do it. One may therefore want to save time here. A priority queue requires a large number of node entries, each time a node is allocated memory, this switch has to be made. A very good solution to this problem is using memory pooling.

1.1 Memory Pooling

Memory pools are fixed sized blocks of memory. In short, memory pool is a memory block which you got from system and use some unit of it to replace the system call malloc/free and new/delete. The advantage of the technology is reuse existing memory block so that reduce the times of system call. There are certain goals that can be achieved using memory pooling which are:

- **Speed** : It allows for faster memory allocation than the compiler provided allocators.

- **Robustness** : the memory pool manages returns all the allocated memory before the program terninates.
- **User Convinience** : Only a little amount of changes in the code are required when using a memory manager. This allows for no major changes the already available algorithms.
- **Portability** : It dose not use plateform dependednt memory management system.

2. Various ways to implement memory pooling

2.1 Making greater size memory chunks

One of the most popular memory management strategies is to request for large memory chunks during program startup 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.

2.2Optimize for common request sizes

In any program, certain specific request sizes are more common than others. Your memory manager will do well if it's optimized to handle these requests better.

2.3Pool deleted memory in containers

Deleted memory during program execution should be pooled in containers. 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. Yet another reason to write your own memory allocator!

3. Timing C++ new/delete operators

We'll start with a simple example. Say your code uses a class called Complex that is meant to represent complex numbers, and it uses the language provided by the new and delete operators, as shown in Program 1.

Program 1.Complex class coded in C++

```
class Complex {
  public:
  Complex (double a, double b): r (a), c (b) {}
  private:
  double r; // Real Part
  double c; // Complex Part
  };
int main(int argc, char* argv[])  {
  Complex* array[1000];
  for (int i = 0;i  <  5000; i++) {
    for (int j = 0; j  <  1000; j++) {
      array[j] = new Complex (i, j);
      }
    for (int j = 0; j  <  1000; j++) {
      delete array[j];
      }    }
  return 0;
  }
```

Each iteration of the outermost loop causes 1000 allocations and de allocations. 5000 such iterations result in 10 million switches between user and kernel code. Compiling this test on a gcc-3.4.6 in a Solaris 10 machine took an average of 3.5 seconds. This is the baseline performance metric for the compiler-provided implementation of the global new and deletes operators. To create a custom memory manager for the Complex class that improves the compiler implementation, you need to override the Complex class-specific new and delete operators.

4. New/Delete: A detailed look

In C++, organized memory management is all about overloading the new or delete operator. Different classes in the code might be candidates for different memory allocation policies, which implies that a class-specific new is required. Otherwise, the new or delete global operator must be overridden. Operator overloading can be done in either of the forms, as shown in Program 2.

Program 2.Overloading the new or delete operator

```
void* operator new(size_t size);
void   operator delete(void* pointerToDelete);
-OR-
void* operator new(size_t size, MemoryManager& memMgr);
void   operator delete(void* pointerToDelete, MemoryManager& memMgr);
```

The overridden new operator is responsible for allocating raw memory of the size specified in the first argument, and the delete operator frees this memory. Note that these routines are meant only to allocate or deallocate memory; they don't call constructors or destructors, respectively. A constructor is invoked on the memory allocated by the new operator, while the delete operator is called only after the destructor is called on an object.

The second variant of new is the placement new operator, which accepts a MemoryManager argument -- basically a data structure to allocate raw memory on which the constructor of an object is finally called. For the purposes of this tutorial, we recommend using the first variant of new or delete because the placement variation results in a massive proliferation of MemoryManager& arguments in the user code, violating the design goal of user convenience.

We use the MemoryManager class to do the actual allocation or deallocation with the new and delete operator routines serving as wrappers for the following MemoryManager routines, as shown in Program 3: MemoryManager gMemoryManager; // Memory Manager, global variable.

Program 3.The new, new[ ], delete, and delete[ ] operators as wrappers

```
MemoryManager gMemoryManager; // Memory Manager, global variable

void* operator new (size_t size)   {
  return gMemoryManager.allocate(size);
  }

void* operator new[ ] (size_t size)  {
  return  gMemoryManager.allocate(size);
  }

void operator delete (void* pointerToDelete)  {
  gMemoryManager.free(pointerToDelete);
  }

void operator delete[ ] (void* arrayToDelete)  {
  gMemoryManager.free(arrayToDelete);
  }
```

Program 4.Memory manager interface

```
class IMemoryManager
  {
  public:
    virtual void* allocate(size_t) = 0;
    virtual void   free(void*) = 0;
  };

class MemoryManager : public IMemoryManager
  {
  public:
    MemoryManager( );
    virtual ~MemoryManager( );
    virtual void* allocate(size_t);
    virtual void   free(void*);
  };
```

We also like to make the allocate and free routines inline for quicker dispatch.

5. Our first memory manager in a single-threaded environment for the Complex type

We designed our first memory manager for this tutorial keeping in mind the principles we've discussed so far. To keep matters simple, this memory manager is customized specifically for objects of type Complex and works only in single-threaded environments. The basic idea is to keep a pool of Complex objects inside the memory manager available and have future allocations happen from this pool. If the number of Complex objects that need to be created exceeds the number of objects in the pool, the pool is expanded. Deleted objects are returned to this pool. Figure 1 is a good illustration of things to come.

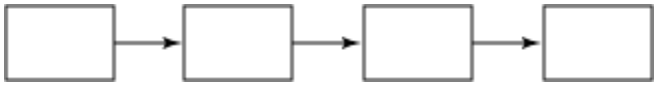

Figure 1. Creating a pool of Complex objects

Each block in the pool serves a dual purpose:

- It stores a Complex object.
- It must be able to connect itself to the next block in the pool.

Storing a pointer inside the Complex data structure is not an option because that increases the overall memory footprint of the design. A better option is to wrap the private variables of the Complex data structure into a structure and create a union with the Complex pointer. When used as part of the pool, the pointer is used to point to the next free element in the pool. When used as standalone Complex object, the structure holding the real and complex part is utilized. Program 5 shows the modified data structure.

Program 5.Modified data structure to store Complex* with no extra overhead

```
class Complex  {
  public:
    Complex (double a, double b): r (a), c (b) {}
    private:
    union {
      struct {
```

```
      double r; // Real Part
      double c; // Complex Part
      };
    Complex* next;
};};
```

However, this strategy violates the design goal of user convenience because we expect to make minimal changes in the original data structure while integrating the memory manager. Improving on this, we designed a new FreeStore data structure that is a wrapper data structure meant to serve as a pointer when kept as part of the pool and as a Complex object otherwise. Program 6 shows the data structure for the FreeStore object.

Program 6.Data structure for FreeStore object

```
struct FreeStore
 {
 FreeStore* next;
 };
```

The pool is then a linked list of FreeStore objects, each of which can point to the next element in the pool and be used as a Complex object. The MemoryManager class must keep a pointer to the head of the first element of the free pool. It should have private methods to expand the size of the pool when needed and a method to clean up the memory when the program terminates. Program 7 contains the modified data structure for a Complex class with FreeStore functionality.

Program 7. Modified data structure for a Complex class with FreeStore functionality

```
#include <sys/types.h>
class MemoryManager: public IMemoryManager   {
 struct FreeStore     {
   FreeStore *next;
  };
 void expandPoolSize ();
 void cleanUp ();
 FreeStore* freeStoreHead;
 public:
  MemoryManager () {
    freeStoreHead = 0;
    expandPoolSize ();
    }
  virtual ~MemoryManager () {
    cleanUp ();
    }
  virtual void* allocate(size_t);
  virtual void   free(void*);
 };
MemoryManager gMemoryManager;
 class Complex  {
 public:
  Complex (double a, double b): r (a), c (b) {}
  inline void* operator new(size_t);
  inline void   operator delete(void*);
 private:
  double r; // Real Part
  double c; // Complex Part
 };
```

The following is the pseudo code for the memory allocation:

1. If the free-store hasn't been created yet, create the free-store, and then go to step 3.
2. If the free-store has been exhausted, create a new free-store.
3. Return the first element of the free-store and mark the next element of the free-store as the free-store head.

The following is the pseudo code for the memory deletion:

1. Make the next field in the deleted pointer point to the current free-store head.
2. Mark the deleted pointer as the free-store head

6. CONCLUSION

That's it! You've created your customized memory manager for the Complex class. Recall that the baseline test took 3.5 seconds. Compiling and running that same test (no need to make any changes to the client code in the main routine) now shows that the program execution takes a stunning 0.67 seconds! Why is there such a dramatic performance improvement? Primarily for two reasons:

- 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.
- 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.

This memory pool is now ready to be implemented with priority queues in its raw form. One can directly make calls to this memory manager while allocating nodes for priority queues.

7. REFERENCES

[1]. Dhananjay Gupta, Avani Kothari, Neetesh Gupta, and Akshay Khandelwal. "COMPARATIVE STUDY OF PRIORITY QUEUE IMPLEMENTATIONS" Internation Journal Of Advance Research And Innovative Ideas In Education Volume 3 Issue 2 2017 Page 4242-4250
[2]. http://stackoverflow.com/questions/11749386/implement-own-memory-pool
[3]. https://www.codeproject.com/articles/27487/why-to-use-memory-pool-and-how-to-implement-it