Multi Threaded Memory Management

By

Jay Narale 150110014

Rutvik Karbhari 150110012

NTRODUCTION

This project is focused on reducing the number of system calls to the kernel for allocation and de-allocation of objects or classes. Basically, in simple terms, in one system call, we request for a large number of Objects and assign them using our logic as the process requests. In this way, we can decrease the number of system calls and make the memory management faster and more effective than the traditional one.

Memory allocation in C++ is done using new, new [ ], delete, and delete [ ] operators. Memory that is assigned in a program and not required by the program is often unintentionally left undeleted. This causes the memory footprint of the process to increase which eventually affects the performance severely.This is because available memory reduces and hard disk access takes a lot of time. We have created a struct freeStore pointer in the complex class that is used to traverse in the pool of empty Complex objects. When we create a new complex number, we allocate the head of the free_space pointer to the newly made object and consequently that pointer is reallocated to the next member in the pool. The freeStore* has been linked with the complex class. Further this is extended to multi threaded process. We lock the freeStore using a mutex lock every time we want to create a new Complex number or delete one.

Design goals

Our memory manager satisfies the following design goals:
 Speed
Robustness
User convenience
Portability

Running Function

Let us say we have a class Complex representing complex numbers with a real part and a complex part. It uses new and delete operations for creation and deletion of our complex objects.

Our Basic Object

```
class Complex{
  public:
  Complex (double a, double b){
             r=a;
            c=b;
  }
  private:
            double r; // Real Part
            double c; // Complex Part
  };
```

Main Function that we are to run is

```
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 deallocations. And 5000 such iterations result in 10 million switches between the user and kernel code.

Compiling this test on a gcc compiler in an Intel I5 machine took an average of 0.168477 seconds. Our Custom Memory Manager takes 0.105 seconds that is a 37% improvement in time.

Custom New and Delete

The implementation of new and delete operations gives us this result. To improve the compiler implementation, we need to override the traditional new and delete operations and replace them with a custom made class.

```
void* operator new(size_t size);
void   operator delete(void* pointerToDelete);
```

The original New operator allocates raw memory of the size taken as input and the delete operator frees this memory.

Our custom new and delete operators
Note-gMemoryManager is a new instance of our New Class

```
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);
 }
```

Note:The size taken as the argument to the `new[ ]` operator is equal to the number of elements in the array multiplied by the size of each element in the array.

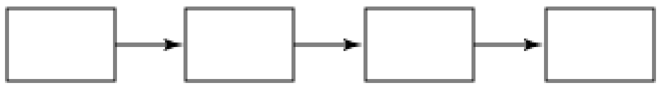
Memory Manager Class

Our Customised Memory Manager Class

```
class MemoryManager
 {
           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*);
 };
```

This customized memory manager is made for objects of type `Complex`. We basically made a pool of Complex objects that are available in the memory manager and the allocations that will be made will happen from this pool. If the number of objects to be created are more than the available number of objects than the pool has to be expanded. The deleted objects will be made available by returning them back to the pool.



Each block in this pool should be able to store a Complex object and should also be able to connect itself with the next block so that it stays connected. The already made objects have no connection to this pool except that when they are deleted, they will be returned back.

Making the Linked List of the Empty Unallocated Objects

Now the problem is storing a pointer in the complex data structure will defeat the purpose. As it leaves memory footprint for the design. The better way is to wrap up all the private variables into a complex data structure and create a pointer with the Complex pointer.

```
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;
   };
 };
```

But, in this case we contradict the design goal of user convenience as we expect to make minimum changes to the original data structure while adding our custom new functions. So we use a freeStore structure that is a wrapper data structure serving as a pointer when it is a part of the pool and as a Complex object otherwise.

```
struct freeStore
  {
  freeStore* next;
  };
```

So the pool is a linked list of freeStore objects, each of which points to the next element in the pool and can be used to assign to a new Complex object. The Memory Manager class keeps the pointer to the head of the first element of the free pool. It should have private methods to expand the size of the pool and to clean up when we are done.

```
#include <sys/types.h>
 #include <bits/stdc++.h>
Using namespace std;
class MemoryManager
  {
  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;
```

So for memory allocation,

1. Create the freeStore if it is not created.
2. If it is exhausted,then create a freeStore.

3. Return the first element of freeStore while marking its next element as freeStoreHead.

For deletion,

1. Make a next field in deleted pointer and fill it with current freeStore head

---

Our function for memory allocation

```
inline void* MemoryManager::allocate(size_t size)
 {
 if (0 == freeStoreHead)
   expandPoolSize ();

 FreeStore* head = freeStoreHead;
 freeStoreHead = head->next;
 return head;
 }
```

Our function for memory freeing

```
inline void MemoryManager::free(void* deleted)
 {
            FreeStore* head = static_cast <FreeStore*> (deleted);
          head->next = freeStoreHead;
          freeStoreHead = head;
 }
```

We are using the same freeStore* pointer as the Complex object so the size of our freeStore pointers must be the one which is greater among the complex and the freeStore* pointer. cleanUp literally cleans up the pool of free available objects that we created. It deletes all the freeStore pointers as well as frees up the memory allocated to the corresponding complex objects.

Functions : expandPoolSize and cleanUp

```
void MemoryManager::expandPoolSize ()
```

```
 {
 size_t size = (sizeof(Complex) > sizeof(FreeStore*)) ?
   sizeof(Complex) : sizeof(FreeStore*);
 FreeStore* head = reinterpret_cast <FreeStore*> (new char[size]);
 freeStoreHead = head;

 for (int i = 0; i < POOLSIZE; i++) {
   head->next = reinterpret_cast <FreeStore*> (new char [size]);
   head = head->next;
   }

 head->next = 0;
 }

void MemoryManager::cleanUp()
 {
 FreeStore* nextPtr = freeStoreHead;
 for (; nextPtr; nextPtr = freeStoreHead) {
   freeStoreHead = freeStoreHead->next;
   delete nextPtr; // remember this was a char array
   }
 }
```

Extension to Multi-Threaded Programs

This idea can be extended to Multi Threaded Programs as well by making use of mutex locks. In a multithreaded environment, memory allocation and deallocation could potentially be attempted by multiple threads at the same time. This means we must ensure that the allocation and free operations in our memory manager are **atomic**. That is, we must provide for a mechanism that guarantees mutual exclusion among two threads when they attempt these operations simultaneously. The standard way to ensure that the allocation and free methods are atomic is to put a lock-based mechanism in place (mutex Lock)
Example

Concurrent versions of the allocation and free methods

```
void* MemoryManager::allocate(size_t size)
 {
 pthread_mutex_lock (&lock);
 ... // usual memory allocation code
 pthread_mutex_unlock (&lock);
 }

void* MemoryManager::free(size_t size)
 {
 pthread_mutex_lock (&lock);
 ... // usual memory deallocation code
 pthread_mutex_unlock (&lock);
 }
```

This can be done for all our Functions similarly.
Note: we use only 1 mutex Lock.