Singleton Class Design

Typical way of implementing a Singleton class by most of the programmer is as below:

```
class Singleton {
public:
    // Returns pointer to the singleton Object
    static Singleton* GetInstance() {
        if(!pInstance) {
           pInstance = new Singleton();
        }
        /* if reference counting is used to keep track of the reference,
           increment the reference count */
        return pInstance;
    }
    static void DestroyInstance() {
        //If reference counting is used, then delete it only if count is 0
        //if(--refCount == 0)
        delete pInstance;
    }
    /* expose all public methods needed here */

private:
    // Hide constructors
    Singleton() {}
    Singleton(const Singleton& s) { }
    // Hide destructor
    ~Singleton() { }
    // Its not mandatory to make assignment private
    Singleton& operator=(const Singleton& s) { }

private:
    static Singleton* pInstance;
    /* encapsulate the data/resources here */
};
```

Though this is widely used pattern for Singleton, the problem with this approach is, its not thread-safe. To make it thread-safe, we would be needing a locking mechanism in GetInstance(), to make it critical section of code.

```
    static Singleton* GetInstance() {
        //Lock the mutex
        mutex_lock();
        if(!pInstance) {
            pInstance = new Singleton();
        }
        //Unlock the mutex
        mutex_unlock();
        /* if reference counting is used to keep track of the reference,
           increment the reference count */
        return pInstance;
    }
```

Above changes, makes this implementaion of Singleton class as thread safe. But at the cost of mutex lock and unlock for every GetInstance. Think of a system where in reference to Singleton object is made heavily, then this will degrede the performance of the Process, expecting lots of context switch and waits.

There is another approach for this problem with a different design, in which it does not need a lock to make the implementaion thread safe. The basic idea is to have a global object, which would be constructed only once (during process load time) and destructed once (during process unloading/termination time)

In this approach, I have tried this:

- Encapsulate all the resources which needs to be singleton in to a class.
- Make all the constructors private so that nobody should be able to construct this class on heap or stack.
- Hide the destructor as well, so that user cant release/destroy the object al all.
- Expose one static GetInstance() function (along with other public functions) to get the pointer to singleton object.
- Implement an internal class [CreateSingleton], which is hidden from the user, that has all the access "rights" to construct and destruct the above mentioned singleton class

Implementaion Details:

File: SingletonClass.h

```
#ifndef _SINGLETON_CLASS_H
#define _SINGLETON_CLASS_H

//forward declaration
class CreateSingleton;

class Singleton {
    //Give access "right" to CreateSingleton class
    friend class CreateSingleton;
public:
    // Returns pointer to the singleton Object
    static Singleton* GetInstance();
    /* expose all public methods needed here */

private:
    // Hide constructors
    Singleton(/*Init Values*/);
    Singleton(const Singleton& s);
    //Hide destructor as well
    ~Singleton();
    // Its not mandatory to make assignment private
    Singleton& operator=(const Singleton& s);

private:
    /* encapsulate the data/resources needed for the Singleton class here */
};

#endif //_SINGLETON_CLASS_H
```

File: SingletonClass.cpp

```
#include "SingletonClass.h"

/*
 * This is an internal class (not exposed to the user) which encapsulates single
 * instance of Singleton object as static data member.
 * So keep the defination/declaration in .cpp file
 * This class has all the access "rights" to construct and destruct the
 * singleton class as this class is "friend" to Singleton class
 */

class CreateSingleton {
    friend class Singleton;
private:
    /*
     * Make Singleton "friend" to this class, so that Singleton can acess this
     * self object
     */
    static Singleton instance;
};

Singleton CreateSingleton::instance;


Singleton::Singleton(/*Init Values*/) /* : initialization_list */ {
    /* Do other initialization now */
}
Singleton::Singleton(const Singleton& s) /* : initialization_list */ {
    /* Do other initialization now */
}

Singleton()::~Singleton() {
    /* Release all the resources held by this class */
}

//return the pointer to the singleton object
Singleton* Singleton::GetInstance() {
    return &CreateSingleton::instance;
}
```

Advantage of this approach is:

1) This does not need a lock/mutex, by design its thread safe
2) Even before executing Main, this singleton object is constructed while loading data segment
3) This does not need a reference counting mechanism

Disadvantage of this approach is:

1) Process loading might take a while, if this singleton class has many resources encapsulated in this

We can expose a Release/Destroy function from SingletonClass to the user, which would release all the resources held by the Singleton class, but user can not release/delete the object itself. Actual destruction of this singleton object happens during process termintaion/unloading.