6.0 Acknowledgements

Most of the initial work of making libC mt-safe was done by Keith Cooley. Several of the issues discussed here and some of the examples are taken from his design document and code changes made by him. The information about exceptions was provided by Kin-Man Chung. Many valuable suggestions were received from the people who reviewed this document and provided feedback, specifically, Robert Hagmann, Steve Kleiman, Dwight Hare, Dalibor (Dado) Vrsalovic, Wayne Gramlich, Ted Goldstein, and Alan Sloane.

7.0 References

[Ellis 1990] Margaret A. Ellis, Bjarne Stroustrup, “The Annotated C++ Reference Manual”.

[Lippman 1991] Stanley B. Lippman, “C++ Primer, second edition”.

[POSIX 1993] POSIX P1003.4a, “API Threads Extension [C Language]”, IEEE.

[Schwarz 1993] Jerry Schwarz, “Doc No. X3J16/93-0004, WG21/N0216, Input/Output Revision 6”.

[SunOS 1993] SunOS 5.2, “Guide to Multi-Threaded Programming”, SunSoft

```
                lockp->rmutex_lock();
            }
            void do_lock_set(stream_rmutex *ptr, int lock_flag) {
                lockp = ptr;
                lock_count = 0;
                if (lock_flag)
                    do_lock();
            }
            void do_unlock() {
                if (lock_count) {
                    lockp->rmutex_unlock();
                    lock_count--;
                }
            }

    public:
        enum lock_choice { lock_defer=0, lock_now=1 };
        stream_locker (stream_MT& obj, lock_choice flag=lock_now) {
            do_lock_set(&(obj.get_rmutex()), flag);
        }
        stream_locker(stream_MT *ptr, lock_choice flag=lock_now) {
            do_lock_set(&(ptr->get_rmutex()), flag);
        }
        stream_locker(stream_rmutex& mut, lock_choice flag=lock_now) {
            do_lock_set(&mut, flag);
        }
        stream_locker(stream_rmutex *ptr, lock_choice flag=lock_now) {
            do_lock_set(ptr, flag);
        }
        ~stream_locker () { do_unlock(); }
        void lock () { do_lock(); }
        void unlock () { do_unlock(); }
    }

#endif    /* _RLOCKS_H */
```

```
    int         count;
    // Initialization is not done in a constructor because of special
    // needs of static objects of this class.
    void        rmutex_init (); // initialization routine
    void        rmutex_lock ();
    void        rmutex_unlock ();

    friend class ios;
    friend class streambuf;
    friend class stream_MT;
    friend class stream_locker;
};

// Class stream_MT is used as a base class. It provides the interfac
// for a class that wants to be mt-safe
class stream_MT
private:
      stream_rmutex       mutlock;
      stream_bool_t       safe_flag; // safe_object or unsafe_object
protected:
      stream_rmutex&      get_rmutex() { return mutlock; }
public:
      enum { unsafe_object=0, safe_object=1 };
      stream_MT() { safe_flag = safe_object; }
      stream_MT(stream_bool_t flag) { safe_flag = flag; }
      void set_safe_flag( stream_bool_t flag) {
            STREAM_REENTRANT(mutlock.rmutex_lock());
            safe_flag = flag;
            STREAM_REENTRANT(mutlock.rmutex_unlock());
      }
      friend class stream_locker; // to provide access to get_rmutex()
};

class stream_locker {
private:
      rmutex_t      *lockp;
      int           lock_count;

      void do_lock() {
            lock_count++;
```

```
unsafe_istream&      ws(unsafe_istream&);
unsafe_ios&          dec(unsafe_ios&);
unsafe_ios&          hex(unsafe_ios&);
unsafe_ios&          oct(unsafe_ios&);

char        *dec_r(char*, int, long, int = 0);
char        *hex_r(char*, int, long, int = 0);
char        *oct_r(char*, int, long, int = 0);

char        *chr_r(char*, int, int, int = 0);
char        *str_r(char*, int, const char *, int = 0);
char        *form_r(char*, int, const char *, ...);
```

5.4 New header file, rlocks.h

This file is included in iostream.h.

```
#ifndef _RLOCKS_H
#define _RLOCKS_H
#include <synch.h>
#include <thread.h>

#ifdef _REENTRANT
#define STREAM_REENTRANT(x)       x
#define STREAM_RMUTEX_LOCK(m, sym)         \
        stream_locker      sym(m, stream_locker::lock_defer);       \
        if (test_safe_flag()) sym.lock();
#else
#define STREAM_REENTRANT(x)
#define STREAM_RMUTEX_LOCK(m, sym)
#endif

typedef char stream_bool_t;

// Class stream_rmutex implements recursive mutex locks
// Note: this class is supposed to be used only by iostreams and not
//          by a user program.
class stream_rmutex {
private:
        mutex_t       mutex;
        thread_t      owner;
        int sputc_unlocked(int);
        int sputn_unlocked(const char *, int);
        int out_waiting_unlocked();
protected:
        char *base_unlocked();
        char *ebuf_unlocked();
        int blen_unlocked();
        char *pbase_unlocked();
        char *eback_unlocked();
        char *gptr_unlocked();
        char *egptr_unlocked();
        char *pptr_unlocked();
        char *epptr_unlocked();
        void setp_unlocked(char*, char*);
        void setg_unlocked(char*, char*, char*);
        void pbump_unlocked(int);
        void gbump_unlocked(int);
        void setb_unlocked(char*, char*, int);
        void unbuffered_unlocked(int);
        int unbuffered_unlocked();
        int allocate_unlocked();
};

class filebuf : public streambuf {
public:
        int is_open_unlocked();
        filebuf* close_unlocked();
        filebuf* open_unlocked(const char*, int, int = filebuf::openprot);
        filebuf* attach_unlocked(int);
};

class strstreambuf : public streambuf {
public:
        int freeze_unlocked();
        char* str_unlocked();
};

unsafe_ostream&     endl(unsafe_ostream&);
unsafe_ostream&     ends(unsafe_ostream&);
unsafe_ostream&     flush(unsafe_ostream&);
```

5.0 Interface Changes to libC

This section lists the interface changes that we have made to the existing libC to make it mt-safe.

5.1 New Classes

stream_rmutex
stream_MT
stream_locker
unsafe_ios
unsafe_istream
unsafe_ostream
unsafe_iostream
unsafe_fstreambase
unsafe_strstreambase

5.2 New Class Hierarchies

```
class streambuf : public stream_MT { ... };
class unsafe_ios { ... };
class ios : virtual public unsafe_ios, public stream_MT { ... };
class unsafe_fstreambase : virtual public unsafe_ios { ... };
class fstreambase : virtual public ios, public unsafe_fstreambase { ... };
class unsafe_strstreambase : public virtual unsafe_ios { ... };
class strstreambase : virtual public ios, public unsafe_strstreambase { ... };
class unsafe_istream : virtual public unsafe_ios { ... };
class unsafe_ostream : virtual public unsafe_ios { ... };
class istream : virtual public ios, public unsafe_istream { ... };
class ostream : virtual public ios, public unsafe_ostream { ... };
class unsafe_iostream : public unsafe_istream, public unsafe_ostream { ... };
```

5.3 New Functions

```
class streambuf {
public:
        int sgetc_unlocked();
        int snextc_unlocked();
        int sbumpc_unlocked();
        void stossc_unlocked();
        void sgetn_unlocked(char *, int);
        int sputbackc_unlocked(char);
        int in_avail_unlocked();
```

The language also does not specify the behavior of a program when the runtime stack gets exhausted before a handler is located during the unwinding of the stack in a thread.

We assume that propagation of exceptions across thread boundaries is not allowed, i.e., an exception thrown in one thread cannot be caught by a handler in another.

4.0 Other libC functions

4.1 set_new_handler

The library function set_new_handler() in libC sets the value of a global variable in the library to a user specified pointer to a function referred to as the *new-handler.* This function is called if *operator new()* cannot find storage to return. A zero argument to set_new_handler() indicates that new-handler should not be called. This is also the default behavior. If a new-handler has been specified, then *operator new()* will make another attempt to allocate memory by calling this function.

Since set_new_handler() uses a global variable, this function is made mt-safe. Although not specified in the draft standard, the global variable _new_handler can also be accessed directly. Examples of using this variable are given in [Lippman 1991] and [Ellis 1990]. Accessing this variable directly in a user program will not be mt-safe unless the user protects reading or modifying this variable with appropriate locking.

We have two choices in defining the behavior of new_handler in a multi-threaded environment:

- Make it specific to each thread.

  This implies using thread-specific data for the new-handler. This also introduces a new semantic for the new-handler.
- Make it common across all threads.

  In this case we can continue to use a global variable for the new-handler.

One of the guiding principles for making a library mt-safe is to minimize the use of thread-specific data. This is mainly for performance reasons. Thus we choose to make the new-handler common across all threads. This does not change the language semantics for new-handler and also avoids using thread-specific data in libC.

4.2 Issues related to exceptions

Some of the functions in libC throw exceptions for error conditions. We document here the issues related to exceptions in a multi threaded environment.

4.2.1 Exception stack

In a typical implementation, an exception stack is used for storing exceptions and bookkeeping information. Initially this stack is allocated statically. When the stack becomes full, additional storage is allocated from the heap. A static variable is used to point to the top of the stack. In an mt-environment, the static stack and the static stack pointer are made thread-specific. An initialization routine initializes the stack when first entering a thread. This is done automatically by the exception handling support in libC.

4.2.2 Behavior in mt-environment

The C++ language does not specify the behavior of exceptions in an mt-environment. In particular, the current implementation of global functions such as set_terminate() and set_unexpected() determine the global behavior of all exceptions in an mt-environment. It may be desirable in some cases to make these functions thread-specific.

3.7.1 Accessing public static data members

Public static data members of a class are like global variables and can be accessed anywhere in a program. Access to these members is inherently mt-unsafe because these members can be modified outside of the protection provided by the library.

In iostreams, only two classes, ios and filebuf, have public static members.

- Class ios:

  static const long basefield;

  static const long adjustfield;

  static const long floatfield;
- Class filebuf:

  static const int openprot;

Since these values are constants, there is no need for access of these members to be made mt-safe.

3.7.2 Operating directly on streambuf

Class ios contains a streambuf object as a data member. The member functions of class ios use this streambuf object to perform the low level production and consumption of characters. For performance reasons, the ios member functions lock the streambuf object, then call the unlocked version of streambuf member functions, and finally unlock the streambuf object.

Instead of using the iostream member functions, a user could obtain the encapsulated streambuf object of ios by using the function

```
streambuf *ios::rdbuf()
```

and operate on it directly. This usage is considered mt-unsafe.

3.7.3 Functions iword() and pword()

The member functions iword() and pword() in class ios return references to dynamically allocated memory. A user of these routines should protect access to values returned by iword() and pword() by using appropriate locking. As a typical use of these routines, one would do the following

1. Use xalloc() to get a unique index. xalloc() is protected by a static lock so it is mt-safe.
2. Use iword() or pword() to get the memory item indexed by the unique index obtained in step 1. Again, iword() and pword() are protected by static mutex locks and are mt-safe.
3. Assign and access values to the iword or pword. This operation is mt-unsafe.

The problem occurs if more than one thread is sharing a unique index to an iword or pword. This will not be mt-safe. The threads will have to use appropriate locking mechanism to make this mt-safe.

```
        owner = self;
        count = 1;
    }
```

3.6.4 rmutex_unlock()

```
void rmutex_t :: rmutex_unlock() {
    if (_thr_main() == -1) {
        return;
    thread_t   self = thr_self();
    mutex_t    *lck = &mutex;
    if (MUTEX_HELD(lck) && owner == self) {
        if (--count == 0) {
            owner = 0;
            mutex_unlock(lck);
        }
    }
    else {
        __stream_abort("Trying to release a lock not acquired in this thread");
    }
    return;
}
```

If the thread holding the lock has made the call to rmutex_unlock(), indicated by examining the mutex structure with MUTEX_HELD(lck), and the owner id is the same as the current thread id, the count is decremented. If the count becomes zero then all functions in the thread have released the lock and a call is made to mutex_unlock:

```
    if (MUTEX_HELD(lck) && owner == self) {
        if (--count == 0) {
            owner = 0;
            mutex_unlock(lck);
        }
    }
```

However, if another thread has managed to call this function, there is an error because it should not be holding the lock. In this case __stream_abort() is called:

```
    else {
        __stream_abort("Trying to release a lock not acquired in this thread");
    }
```

The function __stream_abort() prints the string passed as argument to stderr and calls abort(). rmutex_lock() and rmutex_unlock() must bracket areas of code to be protected, they cannot be used in isolation.

3.7 Unsafe usage of mt-safe classes

Even after modifying libC to make it mt-safe, the library can still be used in a mt-unsafe manner. We discuss here three cases of unsafe usage.

```
    memset(&m->mutex, 0, sizeof(mutex_t));
    count = 0;
    owner = 0;
}
```

3.6.3 rmutex_lock()

```
void stream_rmutex :: rmutex_lock() {
    if (_thr_main() == -1)
        return;
    thread_t   self = thr_self();
    mutex_t   *lck = &mutex;
    if (mutex_trylock(lck) == 0) {
        owner = self;
        count = 1;
        return;
    }
    if (MUTEX_HELD(lck) && owner == self)
        count++;
    else {
        mutex_lock(lck);
        owner = self;
        count = 1;
    }
    return;
}
```

When a thread attempts to acquire a lock the first time, mutex_trylock() will return 0. The function rmutex_lock() will save the thread id and set the initial count to 1 in the recursive mutex structure:

```
    if (mutex_trylock(lck) == 0) {
        owner = self;
        count = 1;
        return;
    }
```

If the thread has already acquired the lock, indicated by examining the mutex structure with MUTEX_HELD(lck), and if the owner id is the same as the current id, the count of the number of times the lock has been acquired is incremented:

```
    if (MUTEX_HELD(lck) && owner == self)
        count++;
```

If another thread is trying to acquire the lock, it will block on a mutex_lock() call until the thread that has the lock releases it, at which time the lock is acquired, the thread id is saved and the initial count is set to 1:

```
    else {
        mutex_lock(lck);
```

```
            lockp2.lock ();    // lock streambuf object
        }
        unsafe_istream::read(p, len);
        return *this;
    }
```

3.6 Implementing recursive locking

This section discusses the definition of class stream_rmutex used for implementing recursive locking.

The threads library does not support recursive mutex locks. Acquiring a lock that a thread already owns results in a deadlock. We provide an implementation of recursive locks to support application level locking and locking for public virtual functions where recursive locking may occur.

Recursive locking requires the ordering of lock/unlock calls; any deviation from this will result in an error.

3.6.1 class stream_rmutex

The recursive mutex class is:

```
// Note: this class is supposed to be used only by iostreams and not by a user program.
class stream_rmutex {
private:
        mutex_t       mutex;     // mutex lock
        thread_t      owner;      // owner of the recursive lock - id returned by thr_self
        int           count;      // count of number of times the lock is acquired
        void rmutex_init();      // initialization routine
        void rmutex_lock();
        void rmutex_unlock();

        friend class ios;
        friend class streambuf;
        friend class stream_MT;
        friend class stream_locker;
};
```

For applications that do not link with the threads library, there is an opportunity at the beginning of each member function of the class to exit without proceeding further.

3.6.2 Initialization routine rmutex_init()

Initialization of stream_rmutex objects is done in rmutex_init() and not in a constructor because of special needs of static objects of this class. The class ios has a static member of stream_rmutex and this member is used for acquiring and releasing a lock in the member functions sync_with_stdio(), bitalloc(), xalloc(), iword() and pword(). Initialization of static objects of stream_rmutex is done only once, in the same place where the global objects cout, cin, cerr and clog are initialized.

```
stream_rmutex::rmutex_init() {
```

For example, consider an istream class. It contains a pointer to a streambuf object and this pointer is passed as an argument to its constructor and through its assignment. Several member functions of the istream class perform operations on this pointer, such as, getting characters from the streambuf and incrementing the buffer pointer in the streambuf object. These operations must be made atomic.

Example:

```
istream& istream::read(char *p, int len) {
        if (ipfx1()) {
                int    c = 0;
                gcount_ = 0;

                // Critical region for streambuf bp starts here
                while (--len >= 0 && (c = bp->sgetc()) != EOF) {
                        *p++ = c;
                        ++gcount_;
                        bp->stossc();
                }
                // Critical region ends here
        }
        return *this;
}
```

bp is the pointer to an object of streambuf class and a member variable of istream. streambuf::sgetc() gets a character from the buffer and streambuf::stossc() increments the get pointer. An intervening call on a bp member function from another thread must not be allowed to occur within the defined critical region. The mt-safe version of this function is:

```
unsafe_istream& unsafe_istream::read(char *p, int len) {
        if (ipfx1()) {
                int    c = 0;
                gcount_ = 0;
                while (--len >= 0 && (c = bp->sgetc_unlocked()) != EOF) {
                        *p++ = c;
                        ++gcount_;
                        bp->stossc_unlocked();
        }
        return *this;
}

istream& istream::read(char *p, int len) {
        stream_locker      lockp1(this, stream_locker::lock_defer);
        streambuf          *buf = rdbuf();
        stream_locker      lockp2(buf, stream_locker::lock_defer);
        if (test_safe_flag()) {
                lockp1.lock ();     // lock istream object
```

tions to be used by an application. It also is a much cleaner design as the code changes are easier to maintain.

3.4.4 Special case: class streambuf

The class streambuf is mainly used in two different ways:

1. As a base class from which other classes can be derived.

   Most of the work of buffer management in streams is done by classes derived from streambuf.
2. As encapsulated objects in other classes.

   The class ios is an example of this. To perform a sequence of operations on the encapsulated streambuf object, the encapsulated object is locked before the operations start and unlocked after the operations are completed. This is discussed further in section 3.5 below.

We treat the class streambuf as a special case and modify it using the “fat” class approach instead of deriving it from an unsafe version. This is done mainly for improved performance; we avoid the extra overhead of calls to mutex lock and unlock routines for every call to a streambuf member function. The locking and unlocking of the streambuf object is done in the caller of the streambuf member functions by following these steps:

1. lock the streambuf object
2. call unlocked version of streambuf member function (this is usually done several times by the caller, inside a loop.)
3. unlock the streambuf object

An example is given in section 3.5 below.

3.4.5 Link-time compatibility

To ensure link-time compatibility between binaries built for single and multi-threaded cases, the same classes are used for single as well as for multi-threaded applications. Also, to improve performance for the single threaded case, a set of stub functions is defined for the libthreads interface in libC itself. If the application is to run in a multi-threaded environment, libthread is linked before libC causing the stub definitions to be ignored. If the application is to run in a single thread, libthread is not linked and the stub definitions from libC will get linked.

A compiler option, -mt, can be provided that will perform all necessary link steps for the user to build an mt-safe application. This option will pass -lthread to the linker and pass -D_REENTRANT to the preprocessor so that code relevant only for mt-safe could be added in header files within #ifdef _REENTRANT ... #endif.

3.5 Compound Operations

In our design approach of deriving mt-safe classes from unsafe ones, we lock an entire routine to achieve reentrancy. We should also ensure that all operations on encapsulated class objects are atomic.

restriction in the iostreams library as there is little scope for concurrency in the existing functions.

3.4.2 Fat MT-safe classes

With this scheme, the existing libC classes are modified to create “fat” mt-safe classes. Every public and protected member function of each class is modified to become reentrant. The modifications involve protecting all references to internal state by locked mutual exclusion regions. Also, for performance reasons, calls to other public/protected member functions from within a locked region may be replaced by calls to unlocked versions of the member functions. The changes involved here are much more extensive than those in the other approach.

Example:

The original function code:

```
int istream::ipfx( int _need) {
    return ((_need ? (ispecial & ~skipping) : ispecial) ?
            do_ipfx(_need) : 1);
}
```

is replaced by the following code which calls an unlocked version of the function do_ipfx():

```
int istream::ipfx( int _need) {
    int    return_val;
    stream_locker     lockp(this, stream_locker::lock_defer);
    if (test_safe_flag())
        lockp.lock ();
    return_val = ((_need ? (ispecial & ~skipping) : ispecial) ?
                do_ipfx_unlocked(_need) : 1);
    return return_val;
}
```

Advantages:

- The class hierarchy is unchanged
- Mutex locks can be placed to maximize potential concurrency.

Disadvantages:

- Many calls to public/protected member functions must be replaced by calls to unlocked versions.
- More than 200 functions need to be modified causing problems for maintenance and debugging.
- Changes are not well partitioned.

3.4.3 Implementation

We chose the design alternative of deriving mt-safe classes from unsafe ones described in section 3.4.1. This approach allows both unsafe and safe versions of member func-

```
class istream : virtual public ios, public unsafe_istream{ // mt-safe version of the class
        . . .
};
```

As far as possible, any changes to make libC mt-safe are confined to the mt-safe classes. The mt-safe version of the class contains the same protected and public member functions as the **unsafe_** base class. Each public/protected member function of the mt-safe class acts as a wrapper function and performs the following steps:

Acquire lock

Call same function in the **unsafe_** base class

Release lock

Return result, if needed

Examples of wrapper functions:

```
long ios::flags( long l) {
        stream_locker     lockp(this, stream_locker::lock_defer);
        if (test_safe_flag())
                lockp.lock ();
        return unsafe_ios::flags( l);
}

int istream::ipfx( int _need) {
        stream_locker     lockp(this, stream_locker::lock_defer);
        if (test_safe_flag())
                lockp.lock ();
        return unsafe_istream::ipfx (_need);
}
```

Each public/protected member function simply acts as a wrapper for calling the same function in the **unsafe_** base class. The wrapper functions can be made inline to improve performance.

Advantages of deriving mt-safe classes from unsafe classes:

- Encapsulates most of the changes in the derived classes making it easier to modify and maintain.
- Allows an application access to the original unsafe classes for best performance provided the application knows what it is doing.

Disadvantages:

- Changes the class hierarchy.
- Fine grain concurrency may not be achieved because an entire function is locked even if only a small portion of the code needs locking. However, this is not a major

```
        if (test_safe_flag())
            lockp.lock();   // The destructor of lockp will release the lock
        . . .
    }
```

For all iostream classes, the default value of safe_flag will be set to stream_MT::-safe_object by the constructor. Note that the function set_safe_flag() is unconditionally protected by mutex locks. The function test_safe_flag() does not need any mutex locks; we assume that reading of a boolean value is always atomic.

If more than one thread tries to change safe_flag then the threads must coordinate among themselves by using an appropriate locking mechanism such as mutex locks. This situation could arise if one thread was performing some operation on a global object after setting it to be in unsafe mode and another thread tried to set the same object to safe mode. The second thread would proceed assuming that the object is in safe mode when actually it may not be since the first thread may not have completed its operation on the object.

3.3.4 Implementation

In our implementation, we have made class objects dynamically switchable between safe and unsafe. This provides maximum flexibility at a small cost. Each public and protected member function has an added cost of a test at the beginning and end of the function. This additional cost applies to both safe as well as unsafe objects.

3.4 Design Alternatives for mt-safe classes

There are two different ways of modifying the iostreams library to make it mt-safe: deriving mt-safe classes from existing unsafe classes, and, modifying the existing unsafe classes to create ‘fat’ mt-safe classes. These two approaches are discussed in the sections below.

3.4.1 MT-safe classes derived from unsafe classes

In this approach, mt-safe classes are derived from the existing unsafe versions. The original classes are renamed by adding a prefix **unsafe_** to the names. Thus for each existing class, there will be an unsafe version which is the original class itself and an mt-safe version derived from the unsafe class. The following example illustrates this:

```
class unsafe_ios { // this is the original class named ios and is mt-unsafe
    . . .
};

class ios : virtual public unsafe_ios, public stream_MT { // mt-safe version
    . . .
};

class unsafe_istream : virtual public unsafe_ios { // original class istream, is mt-unsafe
    . . .
};
```

object when using these classes in a multi-threaded environment. This approach will have no performance penalty for the single threaded case.

A class containing static data members needs special attention because the static members are shared by all objects of the class. For a program using such a class to be mt-safe, all objects of the class should be locked before calling any class member function that accesses these static members, i.e, a user should have a way of providing a lock around the class instead of around an object. Since a user of the class may not be aware if there are any static data members, he may have to lock the entire class every time before calling any member function.

3.3.2 Safe class instances

In this model, all member functions are made reentrant by placing a lock at entry to the function and an unlock before exit as in the examples given in section 3.1 above. The lock and unlock routines will be called even for a single-threaded case. This will have maximum performance penalty for the case of a single thread.

3.3.3 Dynamically changeable class instances

In this approach, the locking and unlocking of an object is performed conditionally on a flag which can be modified at execution time. The following example illustrates this:

```
typedef char    stream_bool_t;

class stream_MT {
private:
        rmutex_t            mutlock;
        stream_bool_t    safe_flag;    // safe_object or unsafe_object
protected:
        stream_rmutex&    get_rmutex() { return mutlock; }
public:
        enum { unsafe_object=0, safe_object=1 };
        stream_MT () { safe_flag = safe_object; }
        stream_MT(stream_bool_t flag) { safe_flag = flag; }
        void set_safe_flag (stream_bool_t  i) {
                mutlock.rmutex_lock();
                safe_flag = i;
                mutlock.rmutex_unlock();
        }
        stream_bool_t    test_safe_flag() { return safe_flag; }
        friend class stream_locker;
}

class ios : public stream_MT { ... };
class streambuf : public stream_MT { ... };
class istream : virtual public ios { ... };

istream& istream::get(char& c) {
        stream_locker    lockp(this, stream_locker::lock_defer);
        }
        stream_locker (stream_MT  *ptr, lock_choice flag=lock_now) {
                do_lock_set(&(ptr->get_rmutex()), flag);
        }
        ~stream_locker () { do_unlock(); }
        void lock () { do_lock(); }
        void unlock() { do_unlock(); }
}
```

The above example can then be written as:

```
{
    stream_locker     lockp(cout);
    cout << func1() << func2();
}
```

If an exception is thrown by func1() or func2(), the stack unwinding will ensure that the destructor of lockp also gets called, thus unlocking cout. If no exception is thrown, the destructor of lockp will get called when the block of code is exited.

The first argument of the stream_locker constructor specifies the stream object to be locked by the stream_locker object. The second argument of the constructor is the lock_choice flag. This flag decides if the stream object is to be locked at the time the stream_locker object is created. The default value of the lock_choice flag in the constructors is lock_now; this specifies that the stream object passed as argument to the stream_locker constructor is to be locked when the stream_locker object is created. The value lock_defer means that the stream object is not to be locked when the stream_-locker object is created. A subsequent call to the stream_locker member function lock() will lock the stream object.

The recommended way of using stream_locker is to create only local objects of this class. The member functions do_lock() and do_unlock() don't use any locking themselves to protect access to data members of the class. They will work correctly for local objects of the class (unless the address of the local object is passed between threads. See section 2.1. We consider this usage unlikely in practice). A user could use global/dynamic objects of this class and provide additional locking himself before calling the members lock() and unlock() but we don't recommend doing this.

3.2.4 Implementation

We use a separate stream_locker class described in section 3.2.3 in our implementation.

3.3 Performance and mt-safety

There are three different user models that address the issues of performance and mt-safety: unsafe class instances, safe class instances, and dynamically changeable class instances.

3.3.1 Unsafe class instances

In this model, no locking mechanism is provided in the classes. As a result, all member functions are mt-unsafe. It is the user's responsibility to lock operations on a class

3.2.3 A separate stream_locker class

There is a problem with the two aproaches described above if we take into account exceptions. Consider the following test case:

```
{
   cout.lock();
   cout << func1() << func2();
   cout.unlock();
}
```

If either func1() or func2() throws an exception, cout.unlock() will not be called and the object cout will remain locked. To solve this problem, we define a special class whose constructor acquires the lock and whose destructor releases the lock:

```
class stream_rmutex {
      . . .
      friend class stream_locker; // to provide access to rmutex_lock() and rmutex_unlock()
};

class stream_MT {
      . . .
      friend class stream_locker; // to provide access for get_rmutex()
};

class stream_locker {
private:
      stream_rmutex    *lockp;
      int              lock_count;
      void do_lock() {
            lock_count++;
            lockp->rmutex_lock();
      }
      void do_lock_set(stream_rmutex *ptr, int lock_flag) {
            lockp = ptr;
            lock_count = 0;
            if (lock_flag)
                  do_lock();
      }
      void do_unlock() {
            if (lock_count) {
                  lockp->rmutex_unlock();
                  lock_count--;
            }
      }
public:
      enum lock_choice { lock_defer=0, lock_now=1 };
      stream_locker (stream_MT& obj, lock_choice flag=lock_now) {
               do_lock_set(&(obj.get_rmutex()), flag);
```

```
class streambuf : public stream_MT { ... };

istream& istream::get( char& c) {
      get_rmutex().rmutex_lock(); // acquire lock (recursive mutex lock)
      . . .
      get_rmutex().rmutex_unlock(); // release lock (recursive mutex unlock)
};
```

These additional calls for locking/unlocking will degrade the performance of the member functions in a single-threaded case. The issue of performance and mt-safety is discussed further in section 3.3 below.

3.2 Implementing a sequence of operations on an object

We consider three ways of implementing a sequence of operations on an object atomically: provide lock/unlock public member functions in each class, use manipulators, and use constructors/destructors of a separate class to do locking and unlocking.

3.2.1 lock/unlock member functions

In this scheme, we would provide two public member functions in each class:

```
lock() { ... }
unlock() { ... }
```

These two functions allow the user to lock a particular stream object, perform a sequence of operations on the object, and then unlock that object. Note that these routines use the same member variable for the lock as the other public member functions of the class.

Example:

```
cout.lock();
cout << “Version = ” << version_no << “\n”;
cout.unlock();
```

ensures that the three strings “Version = “, version_no, and “\n” are sent to the output stream cout without any intervening strings from any other thread.

It is the user’s responsibility to ensure proper use of the routines lock() and unlock(). In the above example, if the functions cout.lock() and cout.unlock() are not called then output to cout from other threads could get mingled with the three strings. Moreover, if the user calls cout.lock() but forgets to call cout.unlock(), the stream cout will remain locked.

3.2.2 lock/unlock manipulators

Instead of defining member functions lock() and unlock() in each class, we could define them as manipulators. The above example could then be written as:

```
cout << lock << “Version = ” << version_no << “\n” << unlock;
```

3.0 Making iostreams library mt-safe

This section discusses implementation details for making the iostreams library mt-safe.

3.1 Making member functions reentrant

The preferred way of providing mutual exclusion and making public class member functions reentrant is to use mutex locks. A simple way of achieving this is to add a data member of type stream_rmutex in each class and use this for acquiring a lock on entry to a function and to release the lock before exiting the function. As mentioned in section 2.3, we use recursive locking to protect the state of objects. The class stream_rmutex is used to implement recursive locking. This class is discussed in section 3.6.

In our implementation, we define a class stream_MT that provides the interface needed for any class to be mt-safe. The classes ios and streambuf are derived from stream_MT.

Example:

```
// Class stream_rmutex implements recursive mutex locks.
// Note: this class is supposed to be used only by iostreams and not by
// a user program, so all its members are private.
class stream_rmutex {
private:
    mutex_t      mutex;
    thread_t     owner;
    int          count;

    // Initialization is not done in a constructor because of special needs
    // of static objects of this class.
    void rmutex_init ();// initialization routine
    void rmutex_lock ();
    void rmutex_unlock ();

    friend class ios;
    friend class streambuf;
    friend class stream_MT;
};

// Class stream_MT is used as a base class. It provides the interface
// for a class that wants to be mt-safe.
class stream_MT {
private:
    stream_rmutex    mutlock;
protected:
    stream_rmutex&     get_rmutex() { return mutlock; }
};

class ios : public stream_MT { ... };
```

- By default all operations in iostreams should be mt-safe. Where needed for performance reasons, an mt-unsafe version of the operation should be provided.

2.6 Constructors and Destructors

2.6.1 Constructors

Class constructors will normally require locking if they access a static member variable or a global variable. If a static member variable or a global variable is not used in a constructor then locking may not be needed. In libC, there are no class constructors that use static member variables or global variables.

2.6.2 Destructors

While one thread is trying to destroy a dynamically allocated object, it is possible that other threads may be trying to access it. We consider this a user error. Providing mutex locks in the destructor will not help because once one thread has successfully destroyed the object, it will no longer exist for the other threads to use. Destructors are not protected by mutex locks and are documented as unsafe. It is the user's responsibility to ensure that dynamically created objects are not destroyed while some thread is still using them.

2.7 Exported Interface of iostreams Library

For the purpose of making the library mt-safe, the exported interface of the iostreams library is considered to be the following:

- Public and protected member functions of the classes of libC.
- A set of base classes that can be extended and customized by the user.

Any changes to libC must ensure that the original exported interface continues to be available after the changes. *We propose that the class hierarchy structure not be considered part of the exported interface*. This proposal, we believe, is in keeping with the discussion of conformance in the iostreams specification (X3J16/93-0004, p 8), in particular, with the specification that a class member may be inherited from an implementation specific base class rather being declared directly. We have modified the iostreams class hierarchy in our implementation. The different design alternatives are discussed in section 3.4 below.

2.8 Other issues

Changes in libC sources to make it mt-safe should also meet the following requirements:

- The behavior of the iostreams library in a single threaded application should not change.
- For a single-threaded application, the performance penalty as a result of changes made to the library should be within acceptable limits, about < 5%.
- The changes should be consistent with the mechanism and style used in mt-safe libc as specified in the Posix standard.
- Thread-specific data is expensive and should be used sparingly.
- Recursive locking is expensive and should be avoided as far as possible, especially in inner loops.

exclusion member functions in each class. These routines can be used for locking and unlocking objects of the class. Another way, and the approach taken by us, is to define a separate locker class to perform the locking.

Section 3.2 below discusses implementation of the different user callable locking schemes. The locking routines in all the different schemes use the same member variable for the lock as the other public member functions of the class. We use recursive locking to accomplish this.

2.3.2 mt-safe virtual public functions

The class streambuf has several virtual member functions and some of these are called from other public member functions of that class. For instance, the virtual function streambuf::underflow() is called from the public member streambuf::sgetc(). Once both these member functions have been made mt-safe, they will each lock the streambuf object before performing any operation on it. The streambuf object may get locked twice if streambuf::sgetc() is called; once by sgetc() itself and the second time by underflow() if underflow() gets called by sgetc(). We use recursive locking to achieve this.

The virtual functions that are called by other public functions are: setbuf(), xsgetn(), underflow(), pbackfail(), doallocate(), xsputn() and overflow().

2.4 References to static data

Routines that maintain or return references to static data members must be modified to use thread-specific data instead. If such routines are not modified then they should be documented as mt-unsafe. In libC, the routines oct(), hex(), dec(), chr(), str(), and form() use static data. We provide a reentrant interface that uses user-specified buffers instead of static data by providing six additional functions oct_r(), hex_r(), dec_r(), chr_r(), str_r(), and form_r().

2.5 Static member variables

Static members of a class are like global variables and are shared by all objects of that class. Access to these variables cannot be protected by using the object's mutex lock. A static mutex is defined in each class that has static mutable member variables. Each member function of a class that accesses a static member variable uses the static mutex to serialize access to the static member. The static mutex is made protected so that derived classes can also use it. In the iostreams library, the only class that needs a static mutex lock is ios where the static mutex is used for acquiring and releasing locks in the member functions sync_with_stdio(), bitalloc(), xalloc(), iword(), and pword().

If a class has public static member variables, then access of these members in a program is inherently mt-unsafe because these members can be easily modified outside of the protection provided by the library. If these members are accessed directly in a user program, then it is the user's responsibility to protect access to these members by using appropriate locking in the program. This issue is discussed further in section 3.7 below.

2.0 MT-safe issues for C++

All the issues discussed in Posix standard 1003.4a, for making the library libc mt-safe, apply to the C++ library libC as well. However C++ has some additional requirements that are not relevant for C and are not addressed by the Posix standard. This section discusses requirements that are specific to C++ and libC.

2.1 Internal state of objects

Every class object has an internal state determined by the values of data members of the object. We define an object to be in a *consistent* state if all its data members have values that reflect its desired state. An object for which only some of the data members have the desired values is said to be in an *inconsistent* state.

A program running in a multi-threaded environment should ensure that if more than one thread tries to modify the internal state of an object then each thread gets the object in a consistent state. This requirement is needed for static and global objects. This requirement is also needed for local (stack based) objects whose addresses are passed between threads.

2.2 Reentrant member functions

Member functions in the original iostreams library do not provide any protection against multiple threads trying to modify the internal state of an object. As a result, most of the library functions in libC will not work correctly in a multi-threaded environment. To ensure reentrancy, the internal state of class objects must be protected by all public and protected member functions. The preferred way of providing this protection is to use mutex locks.

Another requirement for reentrancy is to ensure that compound operations on a stream object are atomic. Some member functions of the iostream classes perform compound operations on the underlying streambuf object. These operations must be made atomic. It must also be possible for the user to make compound operations atomic by explicitly locking iostream objects.

In our implementation, we have defined a class stream_MT that has a mutex lock as a data member. The classes ios and streambuf are derived from stream_MT. The member functions of all iostream classes use the mutex lock of the base class stream_MT for locking operations. This class is discussed in section 3.1 below.

2.3 Recursive locking

Our implementation uses recursive locking to satisfy two requirements: allowing a sequence of operations on an iostream object to be atomic, and, making virtual public member functions mt-safe. The implementation of recursive locking is discussed in section 3.6 below.

2.3.1 Sequence of operations on an object

A user should be able to perform a sequence of operations in an atomic fashion on any iostreams class object. One way of achieving this is to provide user callable mutual

Implementing iostreams in a multi-threaded environment

Mukesh Kapoor

mukesh.kapoor@Eng.sun.com
SunPro
A Sun Microsystems Inc. Business

1.0 Introduction

Multi-threading is a powerful facility that can speed up applications on multi-processor machines; it can also simplify the structuring of applications on both multi-processor and uni-processor machines. A library is said to be multi-threading safe (mt-safe) if all the public functions in its interface are reentrant. This implies that the library provides protection against multiple threads trying to modify the state of objects shared by more than one thread.

The current iostreams library, which is part of libC, was not designed to work in a multi-threaded environment. We have modified libC to make it mt-safe. Our implementation uses the threads library, libthread, which is based on the interface specified in the POSIX 1003.4a standard. This report discusses the issues involved and describes the interface changes needed for making libC mt-safe. Examples are provided from our experience with modifying libC. Wherever possible, the different choices are listed and a rationale given for the recommended choice. Though most of the issues discussed here are specific to the iostreams library, these issues apply for making any C++ class library mt-safe.

Section 2 addresses some general issues for making a C++ library mt-safe. Section 3 discusses implementation details for modifying the iostreams library, section 4 describes other functions of libC that were made mt-safe, and section 5 lists the interface changes that we made for making the iostreams library mt-safe.

The terms libC, C++ library, and iostreams library are used interchangeably in this document.