

# The Brown C++ Threads Package

Version 2.0

*Thomas W. Doeppner Jr.*  
*Department of Computer Science*  
*Brown University*  
*Providence, RI 02912*  
*twd@cs.brown.edu*

## 1. Introduction

The Brown C++ Threads Package is a portable library for the support of multithreaded programming in C++. It is built on top of the POSIX threads library and thus should be usable in any environment supporting POSIX threads.

## 2. Threads in C++

A thread in C++ is represented as a class derived from the abstract class *Thread*. That is, one defines one's own classes of threads, using the supplied *Thread* class as a basis. One creates a thread by allocating a new instance of a subclass of the class *Thread*. The thread itself starts execution in the method *startup*, which must be defined in the subclass.

For example, one might define one's own class as follows:

```
class MyThread : public Thread {
public:
    MyThread(int arg) {data = arg;}
private:
    void startup( );
    int data;
};

void startup( ) {
    threadbody(data);
}
```

One then might create two threads as follows:

```
main( ) {
    ...

    MyThread *t1 = new MyThread(1);
    t1->make_runnable( );
    MyThread *t2 = new MyThread(2);
    t2->make_runnable( );
    ...
}
```

This creates two threads. Note that two steps are required to create a thread: one first creates the thread object and then creates the thread (by calling *make\_runnable*). (The reason for the two steps is to ensure that all constructors have executed before a thread begins its execution.)

N.B.: A thread and its associated object are two different things. A *Thread* object is an important data structure that is used by the thread, but it is not the same as the thread. Deleting the object results in the deletion of data that is being used by the thread, and thus if it is deleted before the thread terminates, major runtime problems will likely ensue. When a thread terminates (except for *JoinableThreads*—see Section 2.4), its associated object is automatically deleted. This means that such threads should not be declared as local variables. For example, the following program has a bug:

```
parent() {
    MyThread t;           // thread t's object is initialized
    t->make_runnable();   // thread t is created

    ...                  // main body of parent
}                       // leaving scope of parent: the destructor for
                        // t is invoked, destroying the thread object,
                        // but the underlying thread continues to run.
                        // Moreover, when the thread terminates, it will
                        // call its own destructor, thereby deleting the thread's
                        // object a second time.
```

## 2.1 The Thread Object and Thread-Specific Data

To invoke a method of the current thread's *Thread* object, you must have a reference to that *Thread* object, which can be obtained via a call to the static method *thread::current()*.

An important concern is *thread-specific data*: data that is private to a thread, yet is referred to in each thread by the same name as in other threads. For example, system calls in single-threaded Unix processes supply failure codes in the global variable *errno*. This is not a good technique for multithreaded processes, since if two or more threads fail in system calls simultaneously, each will expect to find its error code in the same location. Thus we must arrange so that when a thread refers to *errno*, it refers to its own private location, different from the *errno*s of all other threads.

We have a straightforward approach for thread-specific data: subclasses of *thread* can be defined to contain instance variables. Thus to have a variable that is both private to a thread and accessible globally within the thread, we can do something like the following:

```
class MyNiftyThread : public Thread {
public:
    MyNiftyThread();
    int tsd;
private:
    void startup();
};
```

Each instance of *MyNiftyThread* comes along with its own copy of *tsd*. To access a thread's private copy of *tsd*, it merely accesses `((MyNiftyThread *)Thread::current())->tsd`.

## 2.2 The First Thread

The first thread of a program, i.e., the one that calls *main*, must have an associated thread object, just as all other threads do. So that this object (and the rest of the runtime) can be properly initialized, you must supply a declaration for the first thread. This declaration must have at least file scope—its constructors must be called before *main* is called. This object must be a subtype of *FirstThread*; any derived-class constructor you supply is executed before *main* is called and the destructor is executed after normal termination of the program.

The minimum declaration is the following:

```
FirstThread ft;
main(...) {
    ...
}
```

One can take advantage of using a derived class for the first thread as follows:

```
Class MyFirstThread : public FirstThread {
public:
    int tsd;
    MyFirstThread() {
        InitializationCode();
    }
} ft;
```

Thus one can provide thread-specific data for the first thread and supply an initialization procedure to be called before *main* is called. However, this simple use of thread-specific data is not as useful as one might hope. It should be possible for all threads to have a thread-specific data item of the same name. If all threads are of the same class, then we can use the technique shown in the example in Section 2.1 in which we use a simple expression to cast *Thread::current()* to the class of the invoking thread. However, since the first thread derived from *FirstThread* and other threads are instances of other subclasses of *Thread*, this method does not work for both the first thread *and* the other threads.

An alternative technique for providing thread-specific data is to encapsulate it in a separate class which can be inherited by each thread class using it, as in the following example:

```
class TSD {
public:
    int tsd;
};

class MyFirstThread: public Thread, public TSD {
    ...
};

class SecondThread: public Thread, public TSD{
    ...
};
```

However, this still forces one to refer to the thread-specific data using an expression dependent on the referring thread's type: either

```
((MyFirstThread *)Thread::current()->tsd
```

or

```
((SecondThread *)Thread::current()->tsd
```

We need a technique by which we can use one name (or expression) to refer to *tsd* from within threads of either derived thread class. What C++ has to offer here is the virtual function. This allows us to refer to the base class of an object and yet access a function defined in a derived class. So, to achieve our goal of

thread-class-transparent access to thread-specific data, we provide the virtual function *hook*, returning a *void \**, in the *Thread* class. The function has a declared value in the base class, so it is not necessary to redefine it, but one can redefine it as follows to achieve our goal:

```

class TSD {
public:
    int tsd;
};

class MyFirstThread: public Thread, public TSD {
public:
    void *hook() {return((void *)this);}
};

class SecondThread: public Thread, public TSD {
public:
    void *hook() {return((void *)this);}
    ...
};

```

With this approach, one can refer to the thread-specific storage, regardless of the thread class, as

```
((TSD *) (Thread::current()->hook()))->tsd
```

## 2.3 Termination

A thread terminates when it returns from its first procedure (*startup*) and when it calls the static method (of the *Thread* class) *exit* (i.e., *Thread::exit()*). Note that calls to the *exit* routine of the UNIX/C library result in the termination of the entire process, not just of the calling thread. Furthermore, if the first thread returns from *main*, the entire process is terminated. (Returning from *main* is equivalent to calling *exit*, according to POSIX semantics.) However, if the first thread calls *Thread::exit()*, then just the first thread terminates. In all cases, the process terminates according to the usual POSIX rules (a call to *exit* or through the actions of certain signals for which there is no handler) or when all threads in the process have terminated. Note again, as mentioned above, that destroying a *Thread* object (by invoking *delete*) does not necessarily terminate the thread. For information on how one thread can cause the safe termination of another, see the discussion of *alert*, below.

## 2.4 Joinable Threads

A subclass of *Thread* provided in the library is *JoinableThread*. This adds a *join* method to the *Thread* class, allowing one to wait until a thread terminates. Consider the following example:

```

class MyJoinableThread : public JoinableThread {
public:
    MyJoinableThread(int i1, int i2) {arg1 = i1; arg2 = i2;}
    int result() {return answer;}
private:
    void startup();
    int answer;
    int arg1;
    int arg2;
};

```

```

main() {
    ...
    MyJoinableThread *t = new MyJoinableThread(1, 2);
    ...
    t->join();
    int t_result = t->result();
    delete t;
    ...
}

void startup() {
    answer = compute_answer(arg1, arg2);
}

```

Note the independent existence of the Thread object and the thread itself. Our thread computes a value, which it stores in the object before terminating. The mainline thread waits until the thread it created terminates (indicating that the desired value has been computed), then retrieves the value by invoking the *result* method of the Thread object, and finally deletes the no-longer-needed *JoinableThread* object.

In the following example we provide a destructor that makes it safe to have a local declaration of a thread derived from *JoinableThread*:

```

class LocalThread : public JoinableThread {
public:
    LocalThread() {make_runnable();}
    ~LocalThread() {join();}
private:
    void startup() { ... }
};

main() {
    void sub();

    sub();
    ...
}

void sub() {
    LocalThread t;

    ...
}

```

The destructor for *LocalThread* calls *join*, thereby insuring that the thread has terminated before the destructors for the base classes (*JoinableThread* and *Thread*) are called.

### 3. Synchronization

Five synchronization-object classes are provided in the package: mutexes, conditions, semaphores, readers-writers locks, and barriers.

#### 3.1 Mutexes

Mutexes provide mutual exclusion and can be used as follows:

```

class SharedCounter {
public:
    SharedCounter(int i) {counter = i}
    SharedCounter &operator ++();
private:
    int counter;
    Mutex mut;
};

SharedCounter &SharedCounter::operator ++() {
    mut.lock();
    counter++;
    mut.unlock();
}

```

Multiple threads may invoke the ++ operator on the same SharedCounter concurrently; the lock and unlock methods on the mutex insure mutually exclusive access.

An important restriction on the use of mutexes is that only the thread that locked a mutex can unlock it. An attempt to do otherwise throws the exception *MutexProblem*.

Also provided with the mutex class is a *try\_lock* method, which returns EBUSY if the mutex was locked (and hence the caller has not locked the mutex) and returns 0 if the mutex was not locked, but is now locked by the caller. This is useful for avoiding deadlock situations, as the following example of a doubly linked list shows:

```

class ListElement {
public:
    ListElement(ListElement *head);
    ~ListElement();
private:
    ListElement *next;
    ListElement *prev;
    Mutex lock;
};

ListElement::ListElement(ListElement *head) {
    // link the new ListElement into the head of a doubly linked list

    // if head is null, we are creating a new head
    if (head == 0) {
        next = this;
        prev = this;
    } else {
        head->next->lock.lock();           // lock the next node
        if (head->next != head)
            head->lock.lock();           // lock the head
        next = head->next;
        prev = head;
        next->prev = this;
        head = this;
        head->lock.unlock();
        next->lock.unlock();
    }
}

```

```

ListElement::~ListElement() {
    // remove the ListElement from the list

    // is this the last thing on the list?
    if (next == this)
        return;
    next->lock.lock();           // lock the next node
    if (next != prev)
        prev->lock.lock();     // lock the previous node
    next->prev = prev;
    prev->next = next;
    if (next != prev)
        prev->lock.unlock();
    next->lock->unlock();
}

```

The strategy employed in this code is that, to insert or delete a node from the list, one must first lock the mutexes of the following and preceding nodes. However, since the list is circular, there is the chance that a deadlock may arise—for example, if a number of threads are simultaneously deleting all of the list elements. To avoid this problem we might use conditional requests for the mutex, as in the following revised version of the destructor for ListElement:

```

ListElement::~ListElement() {
    // remove the ListElement from the list

    // is this the last thing on the list?
    if (next == this)
        return;
    while (1) {
        next->lock.lock();       // lock the next node
        if (next != prev) {
            if (prev->lock.try_lock() == 0) {
                // we have obtained the lock
                break;
            }
            // give up the first lock and try again
            next->lock.unlock();
        }
    }
    next->prev = prev;
    prev->next = next;
    if (next != prev)
        prev->lock.unlock();
    next->lock->unlock();
}

```

This new version of the method avoids deadlock by not holding onto one lock while waiting indefinitely for the other to become available.

A very useful debugging technique, particularly for multithreaded programs, is the use of *assertions*. For example, suppose that we have written a procedure that assumes that a particular mutex has been locked by the caller. We can use an assertion to test this as follows:

```

procedure() {
    assert(mutex.is_locked());
    ...
}

```

The method *is\_locked* returns 1 if the mutex is currently locked by the calling thread, 0 otherwise. When one is debugging the program, the call to *assert* does nothing if its argument evaluates to a nonzero value, but it prints an error message and terminates the program with a core dump if the argument evaluates to zero. To use assertions, one must include the header file *assert.h*, i.e.,

```
#include <assert.h>
```

If you define the compile-time variable *NDEBUG*, then the *assert* statement does not evaluate its argument and does nothing—once you have debugged your program, you can leave the *assert* statements in the program as useful comments, but you “turn them off” by defining *NDEBUG*.

Be aware that in future versions of this package, *is\_locked* might be supported only as a debug option—an option turned on only when one is more concerned about locating bugs than speed. (It is somewhat expensive to support, thus it would be nice to be able to turn it off.)

### 3.2 Conditions

The *Condition* class is a more powerful synchronization primitive than the *Mutex* class. For example, we can use it to solve the producer-consumer problem, as shown below:

```

class PC {
public:
    PC(int);
    ~PC() {delete buf;}
    void produce(char);
    char consume();
private:
    char *buf;
    Mutex mut;
    Condition more;
    Condition less;
    int nextin;
    int nextout;
    int empty;
};

PC::PC(int arg_size) {
    size = arg_size;
    buf = new char[size];
    nextin = nextout = 0;
    empty = arg_size;
}

void PC::produce(char item) {
    mut.lock();
    while (empty <= 0)
        less.wait(&mut);
    buf[nextin] = item;
    if (++nextin >= size)
        nextin = 0;
}

```



```

    empty--;
    more.notify( );
    mut.unlock( );
}

char PC::consume( ) {
    mut.lock( );
    while (empty >= size)
        more.wait(&mut);
    char item = buf[nextout];
    if (++nextout >= size)
        nextout = 0;
    empty++;
    less.notify( );
    mut.unlock( );
    return(item);
}

```

Here we see two methods on conditions: *wait* and *notify*. The former is fairly complicated: the caller provides the address of a mutex. The mutex is unlocked and the caller thread is placed on a *wait queue* associated with the condition. When the caller thread is woken up (because some other thread has invoked the *notify* method), it first reobtains the lock on the mutex (waiting, if necessary) and finally returns from *wait*. The effect of calls to *notify* is to wake up the first thread waiting on the wait queue. If no threads are waiting on the queue, then nothing happens (and, in particular, the next thread to call *wait* joins the *wait queue* and must be woken up by some subsequent call to *notify*).

An alternative to *notify* is *broadcast*, which wakes up all threads on the queue. These threads compete for the mutex and return from *wait* one at a time.

### 3.3 Semaphores

The *Semaphore* class is based on Dijkstra's notion of semaphore. An instance of this class appears to be a nonnegative integer, to which one can apply two operations: *wait* (known as *P* in Dijkstra's terms) and *post* (Dijkstra's *V*). The effect of invoking the *wait* method is to reduce the value of the semaphore by one, once the semaphore is found to be positive. The effect of a call to *post* is to increase the value of the semaphore by one.

Semaphores are quite useful in some applications, extremely messy in others. Here is another solution to the producer-consumer problem, this time with semaphores.

```

class PCsem {
public:
    PCsem(int) {
        ~PCsem( ) {delete buf;}
        void produce(char);
        char consume( );
private:
    char *buf;
    Semaphore pmut;
    Semaphore cmut;
    Semaphore occupied;
    Semaphore empty;
    int nextin;
    int nextout;
    int size;
};

```

```

PCsem::PCsem(int arg_size) :
    pmut(1), cmut(1), occupied(0), empty(arg_size) {
    size = arg_size;
    buf = new char[size];
    nextin = nextout = 0;
}

void PCsem::produce(char item) {
    empty.wait();           // wait until there is space
    pmut.wait();           // get exclusive access to nextin
    buf[nextin] = item;
    if (++nextin >= size)
        nextin = 0;
    pmut.post();           // release mutual exclusion
    occupied.post();       // indicate one more occupied slot in buf
}

char PCsem::consume() {
    occupied.wait();       // wait until there is something in buf
    cmut.wait();           // get exclusive access to nextout
    char item = buf[nextout];
    if (++nextout >= size)
        nextout = 0;
    cmut.post();           // release mutual exclusion
    empty.post();          // indicate one more empty slot in buf
    return(item);
}

```

In this instance, the solution with semaphores is a bit simpler than the one with conditions. It even allows more parallelism. Note that we use semaphores in two ways. One way (*pmut* and *cmut*) is termed *binary semaphores*—the semaphores, as used, only take values of one and zero. They are used here for mutual exclusion, and could be replaced with mutexes. The other way (*occupied* and *empty*) is termed *counting semaphores*—this is the more general technique.

Note also that, unlike mutexes, there are no restrictions on which threads can invoke the *post* and *wait* methods.

There is an additional method, *try\_wait*, analogous to the *try\_lock* method on mutexes, that does not block, but returns *EBUSY* if the semaphore was zero and zero if the semaphore was positive (its value is now decreased by one).

### 3.4 Readers-Writers Locks

In many situations, strict mutual exclusion is more than is needed. What is often desired is to allow any number of threads to access an object if they are not modifying it, but to allow only one thread access to an object it will modify. This sort of synchronization is provided with the *RW* class. In the simple linked list below, some threads are doing lookups and others are doing additions.

```

class Node {
    friend List;
public:
    Node(int, Node * = 0);
private:

```

```

    Node *link;
    int value;
    RW lock;
};

Node::Node(int v, Node *n) {
    value = v;
    next = n;
}

class List{
    List() : head(0) {head = 0;}
    ~List() { ... }
    int search(int);
    int add(int);
private:
    Node head;
};

int List::search(int key) {
    Node *prev = &head;
    Node *cur;
    prev->lock.rlock();
    for (; prev->link != 0; prev = cur) {
        cur = prev->link;
        cur->lock.rlock();
        prev->lock.unlock();
        if (cur->value == key) {
            cur->lock.unlock();
            return(1);
        }
    }
    prev->lock.unlock();
    return(0);
}

void List::add(int v) {
    Node *prev = &head;
    Node *cur;
    prev->lock.wlock();
    for (; prev->link != 0; prev = cur) {
        cur = prev->link;
        cur->lock.wlock();
        if (v > cur.value) {
            prev->link = new Node(v, cur);
            cur->lock.unlock();
            prev->lock.unlock();
            return;
        }
        prev->lock.unlock();
    }
    prev->link = new Node(v);
    prev->lock.unlock();
}

```

The RW class provides the *rlock* and *wlock* methods, which allow one to take a *read lock*, meaning that writers are not allowed, and to take a *write lock*, meaning that no other threads are allowed. The *unlock* method releases both types of locks. In addition, there are *try\_rlock* and *try\_wlock* methods, which work analogously to the “try” methods of mutexes and semaphores.

As with mutexes, methods are provided for use within assertions for testing whether a lock is held. The method *is\_wlocked* returns whether the calling thread has the RW lock write-locked. The method *is\_rlocked* returns whether *any* thread has the RW lock read-locked. (Note the difference here: *Mutex::is\_locked* and *RW::is\_wlocked* return whether the calling thread holds the lock; *RW::is\_rlocked* merely returns whether the lock is locked or unlocked.)

As with *Mutex::is\_locked*, *RW::is\_wlocked* and *RW::is\_rlocked* might, in future versions of this threads package, be supported only as a debug option.

N.B.: *is\_rlocked* and *is\_wlocked* are not supported in the Solaris version of this package—RW locks are implemented using the readers-writer locks of the Solaris threads package, which precludes any implementation of these tests.

### 3.5 Barriers

A barrier is a generalization of the idea embodied in joinable threads. Barriers supply a *wait* method, which causes calling threads to block until a specified number of threads have called it; then all are released. A simple example is shown below.

```
class SyncStartThread : public Thread {
public:
    SyncStartThread(Barrier *b);
private:
    void startup( );
    Barrier *sync;
};

SyncStartThread::SyncStartThread(Barrier *b) {
    sync = b;
    make_runnable( );
}

void SyncStartThread::startup( ) {
    initialization_routine( );
    b->wait( );
    main_routine( );
}

mainline( ) {
    ...
    Barrier b(6);
    SyncStartThread *t[5];
    for (int i=0; i<5; i++)
        t[i] = new SyncStartThread(&b);
    b.wait( );
    rest_of_program( );
    Thread::exit( )
}
```

Here we are creating six threads of type `SyncStartThread`. Each of these threads executes some initialization code; we want to ensure that all threads have executed their initialization code before they go on with the rest of the program. So the five new threads, plus the mainline thread, call the `wait` method of the barrier. None of them return from this call until all have made the call, and thus all the new threads have executed the initialization code.

N.B.: Be very careful about deleting barriers. There is no reason to believe that all threads have exited a barrier just because they have all entered it. When you delete a barrier, you must be certain that no threads are still inside of it. To make this possible, the `wait` method on barriers returns a one if the calling thread is the last thread to exit from the barrier, otherwise zero. Thus a safe way to delete a barrier is to have all threads calling the `wait` method check the return value. The one and only thread for which `wait` returns zero should delete the barrier. Note that the following program has a bug:

```
xyzzy() {
    Barrier b(6);
    for (int i=0; i<5; i++)
        new SyncStartThread(&b);
    b.wait();
}
```

When the thread calling `xyzzy` returns from `xyzzy`, the destructors for all objects local to the `xyzzy` scope are invoked. Thus the destructor for the barrier `b` is invoked, even though there is no assurance that all threads are out of `b`. In the code for `mainline` above, we avoided this problem by having the thread calling `mainline` perform a `Thread::exit` rather than exit the `mainline` scope, thus insuring that the barrier's destructor is not called. Alternatively, the barrier could have been allocated from the heap (i.e., using `new`). Then each thread could have tested the value returned by `wait`, with the one thread for which `wait` returns zero deleting the barrier.

## 4. Alerts

An alert is a way to get the attention of a thread, often so as to force it to terminate. One thread invokes the `alert` method of another's Thread object; the effect is to cause an `Alerted` exception to be thrown in the target thread. If the target thread does not handle the exception, then it (but not the entire process) is terminated.

Alerts could be dangerous—if they were allowed to happen in an unrestricted fashion, a thread could be interrupted, and perhaps terminated, in the middle of important operations. To keep things safe, alerts are constrained to be acted upon by the target only at well defined, safe points known as *alert points*. That is, the target of an alert will be affected by it only when it executes a routine that is an alert point. The alert points supplied by default are: `Thread::alert_test` (whose affect is only to be an alert point), `Condition::wait`, `Semaphore::wait`, `Barrier::wait`, `Thread::join`, as well as renamed versions of the routines that must be *cancellation points* in POSIX 1003.1c. These latter routines are renamed by capitalizing their first letters. They are: `Aio_suspend`, `Close`, `Creat`, `Fcntl`, `Fsync`, `Mq_receive`, `Mq_send`, `Msync`, `Nanosleep`, `Open`, `Pause`, `Read`, `Sigwaitinfo`, `Sigsuspend`, `Sigtimedwait`, `Sigwait`, `Sleep`, `System`, `Tcdrain`, `Wait`, `Waitpid`, and `Write`. In addition, though not mentioned in 1003.1c, `Accept`, `Poll`, `Readv`, `Select`, and `Writev` are alertable if `accept`, `poll`, `readv`, `select`, and `writev` are cancellation points in the underlying POSIX-threads implementation. Alerts may be masked by use of the method (of the Thread object) `alert_disable` and unmasked via the method `alert_enable`. It is possible to make alert points out of other functions: see the release notes for instructions.

An example of the use of threads is given below in which two threads are searching a database. When one of them finds what they are after, it alerts the other so as to tell it not to search any further.

```
void thread_body() {
    try {
        search(database, per_thread_start_point);
        // the item has been found—notify the other thread that
        // it is no longer necessary for it to search
    }
```

```

        other_thread->alert();
    } catch (Alerted &a) {
        // we've been alerted; if we aren't going to rethrow the
        // exception object then we should delete it
        delete &a;
        // There's no point continuing to search, so
        // we continue with the rest of the program
    }
    rest_of_program();
}

```

Note that the *Alerted* exception has no parameters: the only information conveyed is that an alert has occurred. Also, alerts are not counted by the receiver: when one is received, an *alert-pending* bit is set, and when the *Alerted* exception is thrown, the *alert-pending* bit is cleared. Thus if several alerts are received before any are noticed, only one *Alerted* exception is thrown. Also note the distinction between the alert and the *Alerted* exception. An alert is the action of one thread upon another. The result of the action is to set the target thread's *alert-pending* bit. The target thread must notice that this bit is set (subject to whether alerts are masked off (via *alert\_disable*) or not (via *alert\_enable*)).

If a thread is alerted and throws the *Alerted* exception while blocked within a wait on a condition, the associated mutex will be locked before the exception is thrown. This is important so that one can depend on the state of the mutex when catching the exception. For example, consider the following code fragment:

```

try {
    mutex.lock();
    while(must_wait)
        cond.wait(&mutex);
    mutex.unlock();
} catch (Alerted &a) {
    assert(mutex.is_locked());
    mutex.unlock();
    throw(a);
}

```

We are guaranteed that the assertion within the catch clause is true, and thus can safely unlock the lock and propagate the exception.

## 5. The API

### 5.1 The *Thread* Class

<b>void</b> alert( )	Set the <i>alert-pending</i> bit of the <i>Thread</i> object.
<b>static int</b> alert_disable( )	Set this <i>Thread</i> object so that alerts are masked. It returns zero if <i>alert_disable</i> was already in effect for the thread, one otherwise.
<b>static int</b> alert_enable( )	Set this <i>Thread</i> object so that alerts are not masked. This routine is an alert point. It returns zero if <i>alert_enable</i> was already in effect for the thread, one otherwise.
<b>static void</b> alert_test( ) <b>throw</b> (Alerted)	An alert point—if an alert is pending and unmasked, an <i>Alerted</i> exception is thrown.
<b>int</b> Accept( <b>int</b> fd, <b>struct sockaddr</b> *s, <b>int</b> *len) <b>throw</b> (Alerted);	

```
int Aio_suspend(const struct aiocb *list[ ], int nent, const struct timespec *timeout)
    throw(Alerted);
int Close(int fd) throw(Alerted);
int Creat(const char *path, mode_t mode) throw(Alerted);
int Fcntl(int fd, int request, struct flock *arg) throw(Alerted);
int Fsync(int fd) throw(Alerted);
ssize_t Mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
    unsigned int *msg_prio) throw(Alerted);
int Mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned int msg_prio) throw(Alerted);
int Msync(void *addr, size_t len, int flags) throw(Alerted);
int Nanosleep(const struct timespec *rqtp, struct timespec *rmtp) throw(Alerted);
int Open(const char *path, int oflag) throw(Alerted);
int Open(const char *path, int oflag, mode_t mode) throw(Alerted);
int Pause( ) throw(Alerted);
int Poll(struct pollfd *fds, unsigned long nfd, int timeout) throw(Alerted);
int Read(int fd, void *buf, int size) throw(Alerted);
int Readv(int fd, struct iovec *vec, int size) throw(Alerted);
int Select(int nfd, fd_set *rv, fd_set *wv, fd_set *xv, struct timeval *to) throw(Alerted);
int Sigwaitinfo(const sigset_t *set, siginfo_t *info) throw(Alerted);
int Sigsuspend(const sigset_t *signal_mask) throw(Alerted);
int Sigtimedwait(const sigset_t *set, siginfo_t *info,
    const struct timespec *timeout) throw(Alerted);
int Sigwait(sigset_t *set, int *signal) throw(Alerted);
unsigned int Sleep(unsigned int seconds) throw(Alerted);
int System(const char *string) throw(Alerted);
int Tcdrain(int fd) throw(Alerted);
pid_t Wait(int *status_location) throw(Alerted);
pid_t Waitpid(pid_t process_id, int *status_location, int options) throw(Alerted);
int Write(int fd, void *buf, int size) throw(Alerted);
int Writev(int fd, const struct iovec *vec, int size) throw(Alerted);
```

These are alertable versions of the corresponding system calls: i.e., if alerts are enabled and either are pending at the beginning of the call or become pending during the call, the system call is terminated and the *Alerted* exception is thrown in the thread.

**static Thread \*current( )** This method returns a pointer to the Thread object of the caller.

**static void exit( )** Causes the calling thread to terminate.

**virtual void \*hook( )** The default definition of this function is to return 0. The intent is that programs can override it in derived classes, giving them a way to refer to properties of the derived class while dealing with the base class.

**void** make\_runnable( ) Threads are created in a *suspended* state. To cause them to start execution, this method must be invoked on their Thread objects.

Thread(stack\_size=0) The constructor for the Thread class. The new thread's stack has the size (in bytes) given by the argument (which defaults to zero). A zero stack size results in the default stack of the underlying POSIX-threads implementation. If the thread could not be created because of an error detected in the underlying POSIX-threads implementation, an exception of type *pthread\_failure* is thrown. The member *pthread\_failure::code* is an integer containing the error code (one of the standard values for *errno*—see the introduction to Section 2 of the Unix manual) that describes the problem.

## 5.2 The *JoinableThread* Class

**void** join( ) **throw**(Alerted) The calling thread blocks until the target thread (on whose Thread object this method has been invoked) terminates. The target thread's Thread object continues to exist and must be deleted explicitly. (It is automatically deleted on termination on nonjoinable (default) threads.) If an error occurs (e.g., the target thread has already been joined by some other thread) an exception of type *pthread\_failure* is thrown; *pthread\_failure::code* contains the error code (an integer); see Section 2 of the Unix manual for the meanings of the codes. This method is an alert point. If the method throws an *Alerted* exception, the thread whose object this is a method of becomes detached, i.e., it is no longer necessary for it to be joined.

JoinableThread(stack\_size=0) The constructor for the JoinableThread class. The *stack\_size* argument is as for the constructor for the *Thread* class. It throws the same exception for the same reasons as the constructor for the *Thread* class.

## 5.3 The *Barrier* Class

Barrier(int count) The constructor for the Barrier class. A barrier is initialized with the count given by the argument.

**int** wait( ) **throw**(Alerted) All threads calling this method block until the number of threads given by the count argument to the constructor have called the method. At that point, all threads are released (and the barrier can be used again). One should delete a barrier only after all threads have exited it. When the last thread of a given set of callers exits wait, wait returns 1 (and it is safe to delete the barrier, assuming that it is not being reused). Otherwise wait returns 0. This method is an alert point.

## 5.4 The *Condition* Class

**void** broadcast( ) Releases all threads currently in the wait queue of the Condition object on which this method was invoked. If there are no threads in the queue, then nothing happens.

**void** notify( ) Releases the first thread in the wait queue of the Condition object on which this method was invoked. If there are no threads in the queue, then nothing happens. Beware that occasionally more than one thread



may be released. It is also possible that a thread might be released even if neither `notify` nor `broadcast` is called. This is all done to encourage you to put calls to `Condition::wait` within `while` loops. (It also has something to do with the difficulty in making this work perfectly on all architectures.)

**void** wait(**Mutex** \*m) **throw**(Alerted)

The mutex is unlocked and the calling thread is put on the Condition object's wait queue. Once removed from the queue (by `notify`, `broadcast`, or through the action of an *exception*), the thread locks the mutex (possibly waiting for this to happen) and then returns. This method is an alert point; if alerted, the exception is thrown after the mutex is reacquired.

## 5.5 The *Mutex* Class

**int** is\_locked()

Returns 1 if the calling thread holds the mutex, 0 otherwise. This is intended to be a debugging aid and should not be used as an essential part of a program's logic.

**void** lock()

The calling thread blocks until it finds the mutex unlocked. It then locks the mutex. Only one thread may lock the mutex at a time. Deadlock is certainly possible. A *MutexProblem* exception is thrown if a thread attempts to lock a mutex that it already has locked. This exception carries a single parameter, of type *MutexProblem::problem* (which is an enumeration). The value in this case is *deadlock*. The value can be obtained by examining *MutexProblem::code*.

**int** try\_lock()

The thread attempts to lock the mutex. If immediately successful, the call returns 0. If the mutex is currently locked, the call returns *EBUSY* immediately (this is a standard UNIX error code whose value can be obtained by including *errno.h*).

**int** unlock()

Release the lock on a mutex. It is considered an egregious error if a thread calls `unlock` on a mutex for which it was not the most recent thread to call `lock`. A *MutexProblem* exception is thrown in this case, with a parameter of *not\_your\_mutex*. The value can be obtained by examining *MutexProblem::code*.

## 5.6 The *RW* Class

**int** is\_rlocked()

Returns 1 if the RW lock is read-locked, zero otherwise. This is intended to be a debugging aid and should not be used as an essential part of a program's logic. Not implemented in the Solaris implementation.

**int** is\_wlocked()

Returns 1 if the calling thread holds the write lock, zero otherwise. This is intended to be a debugging aid and should not be used as an essential part of a program's logic. Not implemented in the Solaris implementation.

**void** rlock()

Take a read lock on the object. Any number of threads may be holding a read lock on the object, as long as no threads are holding write locks. The locking strategy favors writers.

<b>int</b> try_rlock( )	Attempt to take a read lock on the object. If it cannot be done immediately, the call returns EBUSY. Otherwise a read lock is taken and the call returns 0.
<b>int</b> try_wlock( )	Attempt to take a write lock on the object. If it cannot be done immediately, the call returns EBUSY. Otherwise a write lock is taken and the call returns 0.
<b>void</b> unlock( )	Release whatever lock is held on the object.
<b>void</b> wlock( )	Take a write lock on the object. Only one thread may be holding a write lock, and only when no threads are holding a read lock on the object. The locking strategy favors writers.

## 5.7 The Semaphore Class

<b>void</b> post( )	Increment the value of the semaphore by one. If there is a queue of threads waiting for the semaphore to become positive, one waiting thread is released.
Semaphore( <b>int</b> count)	The constructor for the Semaphore class. A semaphore is created whose initial value (which must be specified) is <i>count</i> .
<b>int</b> try_wait( )	If the semaphore's value is positive, this decreases it by one and returns 0. Otherwise it returns EBUSY.
<b>int</b> wait( ) <b>throw</b> (Alerted)	This call causes the caller to block until the semaphore's value is positive. When found to be positive, the value is decreased by one and the caller returns. This method is an alert point.

## 6. Using the Package at Brown

The Brown C++ Threads library is in the directory `/pro/threads/C++/lib`. The static version is in `libc++thread.a` and the sharable version is in `libc++thread.so`. The header file is in `/pro/threads/C++/include/c++thread.h`. To compile and link a program on the Suns using the sharable version of the library, use a command similar to the following:

```
CC -mt -o your_prog your_prog.cc \
-I/pro/threads/C++/include \
-L/pro/threads/C++/lib -lc++thread \
-R /pro/threads/C++/lib -lsocket -lnsl -lposix4
```

When debugging an application with *debugger* or *dbx*, you must instruct the debugger to ignore the SIG-PWR signal, which is used by the C++ threads runtime library for support of alerts. To do this, give it the command:

```
ignore PWR
```

Note that there are some issues concerning the thread-safety of the standard IO-stream classes of C++. See the Solaris Answerbook on the C++ Library Reference Manual for a discussion. The `-mt` option to `CC` insures that the appropriate Solaris libraries are included and compile-time variables are defined; thus the Solaris-provided thread-safe versions of all C routines (such as *printf*) are used.

To compile and link a program on an Alpha running Digital Unix 4.0, use a command similar to the following:

Last modified on November 19, 1996 3:36 pm

```
cxx -o your_prog your_prog.cc -pthread \  
-I/pro/threads/C++/dec/include \  
-L/pro/thrads/C++/dec/lib -lc++thread -lrt -laio
```