Threads and Concurrency with C++

Threads and Concurrency with C++

- The Ice threading library provides the following thread-related abstractions:
  - mutexes
  - recursive mutexes
  - read-write recursive mutexes
  - monitors
  - a thread abstraction that allows you to create, control, and destroy threads

Mutexes

- The classes **`IceUtil::Mutex`** (**`IceUtil/Mutex.h`**) **`IceUtil::StaticMutex`** (**`IceUtil/StaticMutex.h`**) provide simple non-recursive mutual exclusion mechanisms

```
namespace IceUtil {
class Mutex {
  public:
    Mutex();
    ~Mutex();
    void lock() const;
    bool tryLock() const;
    void unlock() const;
    typedef LockT<Mutex> Lock;
    typedef TryLockT<Mutex> TryLock;
};

struct StaticMutex {
    void lock() const;
    bool tryLock() const;
    void unlock() const;
    typedef LockT<StaticMutex> Lock;
    typedef TryLockT<StaticMutex> TryLock;
  };
}
```

Mutexes

- **`IceUtil::StaticMutex`** is implemented as a simple data structure, so that instances can be declared statically and initialized during compilation:
  - **`static IceUtil::StaticMutex myStaticMutex = ICE_STATIC_MUTEX_INITIALIZER;`**
- Instances of **`IceUtil::StaticMutex`** are never destroyed.
- **`IceUtil::Mutex`** is implemented as a class and initialized by its constructor and destroyed by its destructor.
- **`IceUtil::Mutex`** and **`IceUtil::StaticMutex`** are non-recursive mutex implementations.
  - Do not call lock on the same mutex more than once from a thread. The mutex is not recursive so, if the owner of a mutex attempts to lock it a second time, the behavior is undefined.
  - Do not call unlock on a mutex unless the calling thread holds the lock. Calling unlock on a mutex that is not currently held by any thread, or calling unlock on a mutex that is held by a different thread, results in undefined behavior.

Thread-Safe File Access for the Filesystem Application

```
#include <IceUtil/Mutex.h>
// ...
namespace Filesystem {
  // ...
  class FileI : virtual public File,
      virtual public Filesystem::NodeI {
    public:
      // As before...
    private:
      Lines _lines;
      IceUtil::Mutex _fileMutex;
  };
  // ...
}
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
  _fileMutex.lock();
  Lines l = _lines;
  _fileMutex.unlock();
  return l;
}
void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
  _fileMutex.lock();
  _lines = text;
  _fileMutex.unlock();
}
```

Guaranteed Unlocking of Mutexes

- Using the raw lock and unlock operations on mutexes has an inherent problem: if you forget to unlock a mutex, your program will deadlock. Forgetting to unlock a mutex is easier than you might suspect:

```
Filesystem::Lines
Filesystem::File::read(const Ice::Current&) const
{
  _fileMutex.lock(); // Lock the mutex
   Lines l = readFileContents(); // Read from database
                                 // Can throw exception
  _fileMutex.unlock(); // Unlock the mutex
   return l;
}

void
SomeClass::someFunction(/* params here... */)
{
  _mutex.lock(); // Lock a mutex
  // Lots of complex code here...
  if (someCondition) {
    // More complex code here...
    return; // Oops!!!
  }
  // More code here...
  _mutex.unlock(); // Unlock the mutex
}
```

Guaranteed Unlocking of Mutexes

- In these examples the early return from the middle of the function leaves the mutex locked.
- Even though the examples makes the problem quite obvious, in large and complex pieces of code, both exceptions and early returns can cause hard-to-track deadlock problems.
- To avoid this, the **`Mutex`** class contains two type definitions for helper classes, called **`Lock`** and **`TryLock`**:

```
namespace IceUtil {
  class Mutex {
  // ...
    typedef LockT<Mutex> Lock;
    typedef TryLockT<Mutex> TryLock;
  };
}
```

- **`LockT`** and **`TryLockT`** are simple templates that primarily consist of a constructor and a destructor
  - the **`LockT`** constructor calls lock on its argument,
  - the **`TryLockT`** constructor calls **`tryLock`** on its argument.
  - The destructors call unlock if the mutex is lock when the template goes out of scope.
  - By instantiating a local variable of type **`Lock`** or **`TryLock`**, we can avoid the deadlock problem entirely
    - This is an example of the RAII (Resource Acquisition Is Initialization) idiom

Guaranteed Unlocking of Mutexes

- Using the **`Lock`** helper, we can rewrite the implementation of our read and write operations as follows:

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
  IceUtil::Mutex::Lock lock(_fileMutex);
  return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
  IceUtil::Mutex::Lock lock(_fileMutex);
  _lines = text;
}
```

- This also eliminates the need to make a copy of the **`_lines`** data member
  - The return value is initialized under protection of the mutex and cannot be modified by another thread once the destructor of **`lock`** unlocks the mutex.

Guaranteed Unlocking of Mutexes

- Both the **`Lock`** and **`TryLock`** templates have a few member functions:
  - **`void acquire() const;`**
    - This function attempts to acquire the lock and blocks the calling thread until the lock becomes available.
    - If the caller calls acquire on a mutex it has locked previously, the function throws **`ThreadLockedException`**.
  - **`bool tryAcquire() const;`**
    - This function attempts to acquire the mutex.
    - If the mutex can be acquired, it returns true with the mutex locked; if the mutex cannot be acquired, it returns false. If the caller calls **`tryAcquire`** on a mutex it has locked previously, the function throws **`ThreadLockedException`**.
  - **`void release() const;`**
    - This function releases a previously locked mutex.
    - If the caller calls release on a mutex it has unlocked previously, the function throws **`ThreadLockedException`**.
  - **`bool acquired() const;`**
    - This function returns true if the caller has locked the mutex previously and false, otherwise.
    - If you use the **`TryLock`** template, you must call acquired after instantiating the template to test whether the lock actually was acquired.

Recursive Mutexes

- A non-recursive mutex cannot be locked more than once, even by the thread that holds the lock.
- This frequently becomes a problem if a program contains a number of functions, each of which must acquire a mutex, and you want to call one function as part of the implementation of another function:

```
IceUtil::Mutex _mutex;
void f1()
{
  IceUtil::Mutex::Lock lock(_mutex);
  // ...
}

void f2()
{
  IceUtil::Mutex::Lock lock(_mutex);
  // Some code here...
  // Call f1 as a helper function
  f1(); // Deadlock!
  // More code here...
}
```

- **`f1`** and **`f2`** each correctly lock the mutex before manipulating data but **`f2`** calls **`f1`**.
  - At that point, the program deadlocks because **`f2`** already holds the lock that **`f1`** is trying to acquire.

Recursive Mutexes

- For this simple example, the problem is obvious.
  - In complex systems with many functions that acquire and release locks, it can get very difficult to track down this kind of situation:
    - The locking conventions are not manifest anywhere but in the source code
    - Each caller must know which locks to acquire (or not to acquire) before calling a function.
  - The resulting complexity can quickly get out of hand.

Recursive Mutexes

- Ice provides a recursive mutex class **`RecMutex`** (**`IceUtil/RecMutex.h`**) that avoids this problem:

```
namespace IceUtil {
  class RecMutex {
    public:
      void lock() const;
      bool tryLock() const;
      void unlock() const;
      typedef LockT<RecMutex> Lock;
      typedef TryLockT<RecMutex> TryLock;
  };
}
```

- The signatures of the operations are the same as for **`IceUtil::Mutex`**.
  - However, **`RecMutex`** implements a recursive mutex:
    - **`lock`**
      - The **`lock`** function attempts to acquire the mutex. If the mutex is already locked by another thread, it suspends the calling thread until the mutex becomes available. If the mutex is available or is already locked by the calling thread, the call returns immediately with the mutex locked.
    - **`tryLock`**
      - The **`tryLock`** function works like lock, but, instead of blocking the caller, it returns false if the mutex is locked by another thread. Otherwise, the return value is true.
    - **`unlock`**
      - The **`unlock`** function unlocks the mutex.

Recursive Mutexes

- As for non-recursive mutexes, you must adhere to a few simple rules for recursive mutexes:
  - Do not call unlock on a mutex unless the calling thread holds the lock.
  - You must call unlock as many times as you called lock for the mutex to become available to another thread.
- Using recursive mutexes, the code fragment from the previous slide works correctly:

```
#include <IceUtil/RecMutex.h>
// ...
IceUtil::RecMutex _mutex; // Recursive mutex
void f1()
{
  IceUtil::RecMutex::Lock lock(_mutex);
  // ...
}

void f2()
{
  IceUtil::RecMutex::Lock lock(_mutex);
  // Some code here...
  // Call f1 as a helper function
  f1(); // Fine
  // More code here...
}
```

Read-Write Recursive Mutexes

- Our implementation of read and write operations is more conservative in its locking than strictly necessary: only one thread can be in either the read or write operation at a time.
- However, we have problems with concurrent file access only if we have concurrent writers, or concurrent readers and writers for the same file.
- If we have only readers, there is no need to serialize access for all the reading threads because none of them updates the file contents.

Read-Write Recursive Mutexes

- Ice provides a read-write recursive mutex class **`RWRecMutex`** (**`IceUtil/RWRecMutex.h`**) that implements a reader-writer lock:

```
namespace IceUtil {
  class RWRecMutex {
    public:
      void readLock() const;
      bool tryReadLock() const;
      bool timedReadLock(const Time&) const;
      void writeLock() const;
      bool tryWriteLock() const;
      bool timedWriteLock(const Time&) const;
      void unlock() const;
      void upgrade() const;
      bool timedUpgrade(const Time&) const;
      void downgrade() const;
      typedef RLockT<RWRecMutex> RLock;
      typedef TryRLockT<RWRecMutex> TryRLock;
      typedef WLockT<RWRecMutex> WLock;
      typedef TryWLockT<RWRecMutex> TryWLock;
  };
}
```

Read-Write Recursive Mutexes

- A read-write recursive mutex splits the usual single lock operation into **`readLock`** and **`writeLock`** operations.
- Multiple readers can each acquire the mutex in parallel.
- Only a single writer can hold the mutex at any one time (with neither other readers nor other writers being present).
- A **`RWRecMutex`** is recursive, meaning that you can call **`readLock`** or **`writeLock`** multiple times from the same calling thread.

Read-Write Recursive Mutexes

- **`readLock`**
  - Acquires a read lock. If a writer currently holds the mutex or a thread is waiting for a lock upgrade, the caller is suspended until the mutex becomes available for reading. If the mutex is available, or only readers currently hold the mutex, the call returns immediately with the mutex locked.
- **`tryReadLock`**
  - Attempts to acquire a read lock. If the lock is currently held by a writer or a thread is waiting for a lock upgrade, the function returns false. Otherwise, it acquires the lock and returns true.
- **`timedReadLock`**
  - Attempts to acquire a read lock. If the lock is currently held by a writer or another thread is waiting for an upgrade, the function waits for the specified timeout. If the lock can be acquired within the timeout, the function returns true with the lock held. Otherwise, once the timeout expires, the function returns false.

Read-Write Recursive Mutexes

- **`writeLock`**
  - Acquires a write lock. If readers or a writer currently hold the mutex or another thread is waiting for an upgrade, the caller is suspended until the mutex becomes available for writing. If the mutex is available, the call returns immediately with the lock held.
- **`tryWriteLock`**
  - Attempts to acquire a write lock. If the lock is currently held by readers or a writer, or if another thread is waiting for an upgrade, the function returns false. Otherwise, it acquires the lock and returns true.
- **`timedWriteLock`**
  - Attempts to acquire a write lock. If the lock is currently held by readers or a writer, or if another thread is waiting for an upgrade, the function waits for the specified timeout. If the lock can be acquired within the timeout, the function returns true with the lock held. Otherwise, once the timeout expires, the function returns false.

Read-Write Recursive Mutexes

- **`unlock`**
  - Unlocks the mutex (whether currently held for reading or writing).
- **`upgrade`**
  - Upgrades a read lock to a write lock. If other readers currently hold the mutex, the caller is suspended until the mutex becomes available for writing. If the mutex is available, the call returns immediately with the lock held.
  - Only one reader can attempt to upgrade a lock at a time. If several threads call upgrade, all but the first thread receive a **`DeadlockException`**.
  - **`upgrade`** is non-recursive. Do not call it more than once from the same thread.
- **`timedUpgrade`**
  - Attempts to upgrade a read lock to a write lock. If the lock is currently held by other readers, the function waits for the specified timeout. If the lock can be acquired within the timeout, the function returns true with the lock held. Otherwise, once the timeout expires, the function returns false.
  - If another thread is waiting to upgrade the lock, **`timedUpgrade`** returns false immediately.
  - **`timedUpgrade`** is non-recursive. Do not call it more than once from the same thread.
- **`downgrade`**
  - Converts a write lock to a read lock.

Read-Write Recursive Mutexes

- You must adhere to a few rules for correct use of read-write locks:
  - Do not call unlock on a mutex unless the calling thread holds the lock.
  - You must call **`unlock`** as many times as you called **`readLock`** or **`writeLock`** (or **`upgrade`** or successful **`timedUpgrade`**) for the mutex to become available to another thread.
  - Do not call **`upgrade`** or **`timedUpgrade`** on a mutex for which you do not hold a read lock.
  - **`upgrade`** and **`timedUpgrade`** are non-recursive. Do not call these methods more than once from the same thread.
  - Do not call **`downgrade`** on a mutex unless the calling thread holds a write lock.
  - You must call **`downgrade`** (or **`unlock`**) as many times as you called **`writeLock`** and **`upgrade`** (or successfully called **`timedUpgrade`**) for the mutex to become available to another thread.

Read-Write Recursive Mutexes

- Using a **`RWRecMutex`**, we can implement our read and write operations to allow multiple readers in parallel, or a single writer:

```
#include <IceUtil/RWRecMutex.h>
// ...
namespace Filesystem {
  // ...
  class FileI : virtual public File,
    virtual public Filesystem::NodeI {
      public:
        // As before...
      private:
        Lines _lines;
        IceUtil::RWRecMutex _fileMutex; // Read-write mutex
  };
  // ...
}
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
  IceUtil::RWRecMutex::RLock lock(_fileMutex); // Read lock
  return _lines;
}
void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
  IceUtil::RWRecMutex::WLock lock(_fileMutex); // Write lock
  _lines = text;
}
```

Timed Locks

- Read-write locks provide member functions that operate with a timeout.
- The amount of time to wait is specified by an instance of the **`IceUtil::Time`** class (**`IceUtil/Time.h`**)
- We can get the timeout values using the following **`Time`** member functions
  - **`seconds`**
  - **`milliSeconds`**
  - **`microSeconds`**
- These functions construct **`Time`** objects from the argument in the specified units. For example, the following code fragment creates a time duration of one minute:
  - **`IceUtil::Time t = IceUtil::Time::seconds(60);`**

Timed Locks

```
#include <IceUtil/RWRecMutex.h>

// ...

IceUtil::RWRecMutex _mutex;

// ...
// Wait for up to two seconds to get a write lock...
//

IceUtil::RWRecMutex::TryWLock lock(_mutex, IceUtil::Time::seconds(2));
if (lock.acquired())
{
  // Got the lock -- destructor of lock will unlock
}
else
{
  // Waited for two seconds without getting the lock...
}
```

- **`TryRLock`** and **`TryWLock`** constructors are overloaded: if you supply only a mutex as the sole argument, the constructor calls **`tryReadLock`** or **`tryWriteLock`**; if you supply both a mutex and a timeout, the constructor calls **`timedReadLock`** or **`timedWriteLock`**.

Monitors

- Mutexes implement a simple mutual exclusion mechanism that allows only a single thread (or, in the case of read-write mutexes, a single writer thread or multiple reader threads) to be active in a critical region at a time.
- In particular, for another thread to enter the critical region, another thread must leave it.
  - It is impossible to suspend a thread inside a critical region and have that thread wake up again at a later time, for example, when a condition becomes true.

Monitors

- A monitor is a synchronization mechanism that protects a critical region:
  - As for a mutex, only one thread may be active at a time inside the critical region.
  - However, a monitor allows you to suspend a thread inside the critical region;
    - Doing so allows another thread to enter the critical region.
  - The second thread can either leave the monitor (thereby unlocking the monitor), or it can suspend itself inside the monitor;
    - Either way, the original thread is woken up and continues execution inside the monitor.

Monitors

- This extends to any number of threads, so several threads can be suspended inside a monitor
  - Monitors provide a more flexible mutual exclusion mechanism than mutexes because they allow a thread to check a condition and, if the condition is false, put itself to sleep;
  - The thread is woken up by some other thread that has changed the condition.
- The monitors provided by Ice have Mesa semantics, so called because they were first implemented by the Mesa programming language.
  - Mesa monitors are provided by a number of languages, including Java and Ada.
  - With Mesa semantics, the signalling thread continues to run and another thread gets to run only once the signalling thread suspends itself or leaves the monitor.

The `Monitor` Class

- Ice provides monitors with the **`IceUtil::Monitor`** class (**`IceUtil/Monitor.h`**)

```
namespace IceUtil {
  template <class T>
    class Monitor {
      public:
        void lock() const;
        void unlock() const;
        bool tryLock() const;
        void wait() const;
        bool timedWait(const Time&) const;
        void notify();
        void notifyAll();
        typedef LockT<Monitor<T> > Lock;
        typedef TryLockT<Monitor<T> > TryLock;
    };
}
```

- **`Monitor`** is a template class that requires either **`Mutex`** or **`RecMutex`** as its template parameter.
  - Instantiating a **`Monitor`** with a **`RecMutex`** makes the monitor recursive.

The `Monitor` Class

- **`lock`**
  - Attempts to lock the monitor. If the monitor is currently locked by another thread, the calling thread is suspended until the monitor becomes available. The call returns with the monitor locked.
- **`tryLock`**
  - Attempts to lock a monitor. If the monitor is available, the call returns true with the monitor locked. If the monitor is locked by another thread, the call returns false.
- **`unlock`**
  - Unlocks a monitor. If other threads are waiting to enter the monitor (are blocked inside a call to lock), one of the threads is woken up and locks the monitor.

The `Monitor` Class

- **`wait`**
  - Suspends the calling thread and, at the same time, releases the lock on the monitor. A thread suspended inside a call to wait can be woken up by another thread that calls notify or **`notifyAll`**. When the call returns, the suspended thread resumes execution with the monitor locked.
- **`timedWait`**
  - Suspends the calling thread for up to the specified timeout. If another thread calls **`notify`** or **`notifyAll`** and wakes up the suspended thread before the timeout expires, the call returns true and the suspended thread resumes execution with the monitor locked. Otherwise, if the timeout expires, the function returns false.

The `Monitor` Class

- **`notify`**
  - Wakes up a single thread that is currently suspended in a call to wait or **`timedWait`**. If no thread is suspended in a call to wait or **`timedWait`** at the time notify is called, the notification is lost (that is, calls to notify are not remembered if there is no thread to be woken up). Note that notifying does not run another thread immediately. Another thread gets to run only once the notifying thread either calls wait or **`timedWait`** or unlocks the monitor (Mesa semantics).
- **`notifyAll`**
  - Wakes up all threads that are currently suspended in a call to wait or **`timedWait`**. As for notify, calls to **`notifyAll`** are lost if no threads are suspended at the time. As for notify, **`notifyAll`** causes other threads to run only once the notifying thread has either called wait or **`timedWait`** or unlocked the monitor (Mesa semantics).

The `Monitor` Class

- You must adhere to a few rules for monitors to work correctly:
  - Do not call **`unlock`** unless you hold the lock. If you instantiate a monitor with a recursive mutex, you get recursive semantics, that is, you must call **`unlock`** as many times as you have called **`lock`** (or **`tryLock`**) for the monitor to become available.
  - Do not call **`wait`** or **`timedWait`** unless you hold the lock.
  - Do not call **`notify`** or **`notifyAll`** unless you hold the lock.
  - When returning from a wait call, you must re-test the condition before proceeding.

Using Monitors

- Consider a simple unbounded queue of items.
- A number of producer threads add items to the queue, and a number of consumer threads remove items from the queue.
- If the queue becomes empty, consumers must wait until a producer puts a new item on the queue.
- The queue itself is a critical region, that is, we cannot allow a producer to put an item on the queue while a consumer is removing an item.
- Producers call the put method to enqueue an item, and consumers call the get method to dequeue an item.
- This implementation of the queue is not thread-safe and there is nothing to stop a consumer from attempting to dequeue an item from an empty queue.

```
template<class T> class Queue {
  public:
    void put(const T& item) {
      _q.push_back(item);
    }
    T get() {
      T item = _q.front();
      _q.pop_front();
      return item;
    }
  private:
  list<T> _q;
};
```

Using Monitors

- Here is a version of the queue that uses a monitor to suspend a consumer if the queue is empty.
- Queue class now inherits from **`IceUtil::Monitor<IceUtil::Mutex>`**, that is, **`Queue`** is-a **`Monitor`**

```
#include <IceUtil/Monitor.h>
template<class T> class Queue
: public IceUtil::Monitor<IceUtil::Mutex> {
  public:
    void put(const T& item) {
      IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
      _q.push_back(item);
      notify();
    }
    T get() {
      IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
      while (_q.size() == 0)
        wait();
      T item = _q.front();
      _q.pop_front();
      return item;
    }
  private:
    list<T> _q;
};
```

Using Monitors

- For this machinery to work correctly, the implementation of **`get`** does two things:
  - **`get`** tests whether the queue is empty after acquiring the lock.
  - **`get`** re-tests the condition in a loop around the call to wait; if the queue is still empty after wait returns, the wait call is re-entered.
- You must always write your code to follow the same pattern:
  - Never test a condition unless you hold the lock.
  - Always re-test the condition in a loop around wait. If the test still shows the wrong outcome, call wait again.

Using Monitors

- Not adhering to these conditions will eventually result in a thread accessing shared data when it is not in its expected state, for the following reasons:
  - If you test a condition without holding the lock, there is nothing to prevent another thread from entering the monitor and changing its state before you can acquire the lock. This means that, by the time you get around to locking the monitor, the state of the monitor may no longer be in agreement with the result of the test.
  - Some thread implementations suffer from a problem known as spurious wake-up: occasionally, more than one thread may wake up in response to a call to notify, or a thread may wake up without any call to notify at all. As a result, each thread that returns from a call to wait must re-test the condition to ensure that the monitor is in its expected state: the fact that wait returns does not indicate that the condition has changed.

Efficient Notification

- The previous implementation of our thread-safe queue unconditionally notifies a waiting reader whenever a writer deposits an item into the queue.
- If no reader is waiting, the notification is lost and does no harm.
- Unless there is only a single reader and writer, many notifications will be sent unnecessarily, causing unwanted overhead.
- Here is a way to fix the problem:

Efficient Notification

- We keep track of the number of waiting readers and call notify only if a reader needs to be woken up:

```
#include <IceUtil/Monitor.h>
template<class T> class Queue
: public IceUtil::Monitor<IceUtil::Mutex> {
  public:
    Queue() : _waitingReaders(0) {}
    void put(const T& item) {
      IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
      _q.push_back(item);
      if (_waitingReaders)
        notify();
    }
    T get() {
      IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
      while (_q.size() == 0) {
        try {
          ++_waitingReaders;
          wait();
          --_waitingReaders;
        } catch (...) {
          --_waitingReaders;
          throw;
       }
      }
      T item = _q.front();
      _q.pop_front();
      return item;
    }
  private:
    list<T> _q;
    short _waitingReaders;
};
```

Threads

- The server-side Ice run time by default creates a thread pool for you and automatically dispatches each incoming request in its own thread.
- As a result, you usually only need to worry about synchronization among threads to protect critical regions when you implement a server.
- However, you may wish to create threads of your own.
  - You might need a dedicated thread that responds to input from a user interface.
  - If you have complex and long-running operations that can exploit parallelism, you might wish to use multiple threads for the implementation of that operation.
- Ice provides a simple thread abstraction that permits you to write portable source code regardless of the native threading platform.
  - This shields you from the native underlying thread APIs and guarantees uniform semantics regardless of your deployment platform.

The Thread Class

- The basic thread abstraction in Ice is provided by two classes, **ThreadControl** and **Thread** (**IceUtil/Thread.h**):

```
namespace IceUtil {
  class Time;
  class ThreadControl {
    public:
#ifdef _WIN32
      typedef DWORD ID;
#else
      typedef pthread_t ID;
#endif
      ThreadControl();
#ifdef _WIN32
      ThreadControl(HANDLE, DWORD);
#else
      ThreadControl(explicit pthread_t);
#endif
      ID id() const;
      void join();
      void detach();
      static void sleep(const Time&);
      static void yield();
      bool operator==(const ThreadControl&) const;
      bool operator!=(const ThreadControl&) const;
  };
```

```
  class Thread {
    public:
      virtual void run() = 0;
      ThreadControl start(size_t = 0);
      ThreadControl getThreadControl() const;
      bool isAlive() const;
      bool operator==(const Thread&) const;
      bool operator!=(const Thread&) const;
      bool operator<(const Thread&) const;
  };
  typedef Handle<Thread> ThreadPtr;
}
```

The `Thread` Class

- The **`Thread`** class is an abstract base class with a pure virtual **`run`** method.
- To create a thread, you must specialize the **`Thread`** class and implement the **`run`** method (which becomes the starting stack frame for the new thread).
- You must not allow any exceptions to escape from run.
  - The Ice run time installs an exception handler that calls **`::std::terminate`** if **`run`** terminates with an exception.

The `Thread` Class

- **`start`**
  - Starts a newly-created thread (that is, calls the **`run`** method).
  - The optional parameter specifies a stack size (in bytes) for the thread. The default value of zero creates the thread with a default stack size that is determined by the operating system.
  - The return value is a **`ThreadControl`** object for the new thread
  - You can start a thread only once; calling start on an already-started thread raises **`ThreadStartedException`**.
- **`getThreadControl`**
  - This member function returns a thread control object for the thread on which it is invoked.
  - Calling this method before calling start raises a **`ThreadNotStartedException`**.
- **`id`**
  - This method returns the underlying thread ID (**`DWORD`** for Windows and **`pthread_t`** for POSIX threads). This method is provided mainly for debugging purposes

The `Thread` Class

- **`isAlive`**
  - Returns false before a thread's start method has been called and after a thread's run method has completed; otherwise, while the thread is still running, it returns true.
  - **`isAlive`** is useful to implement a non-blocking join:

```
ThreadPtr p = new MyThread();
// ...
while(p->isAlive()) {
  // Do something else...
}
t.join(); // Will not block• operator==
```

- **`operator!=`**
- **`operator<`**
  - Compare the in-memory address of two threads.
  - They are provided so you can use sorted STL containers with **`Thread`** objects.

Implementing Threads

```
#include <IceUtil/Thread.h>
// ...
Queue q;
class ReaderThread : public IceUtil::Thread {
  virtual void run() {
    for (int i = 0; i < 100; ++i)
      cout << q.get() << endl;
    }
  };
class WriterThread : public IceUtil::Thread {
  virtual void run() {
    for (int i = 0; i < 100; ++i)
      q.put(i);
  }
};
```

Creating Threads

- To create a new thread, we simply instantiate the thread and call its **`start`** method:

```
IceUtil::ThreadPtr t = new ReaderThread;
t->start();
// ...
```

- We assign the return value from new to a smart pointer of type **`ThreadPtr`**. This ensures that we do not suffer a memory leak:
  - When the thread is created, its reference count is set to zero.
  - Prior to calling **`run`** (which is called by the **`start`** method), start increments the reference count of the thread to 1.
  - For each **`ThreadPtr`** for the thread, the reference count of the thread is incremented by 1, and for each **`ThreadPtr`** that is destroyed, the reference count is decremented by 1.
  - ThreadPtr is another example of an RAII class
  - When **`run`** completes, **`start`** decrements the reference count again and then checks its value: if the value is zero at this point, the **`Thread`** object deallocates itself by calling **`delete this`**; if the value is non-zero at this point, there are other smart pointers that reference this **`Thread`** object and deletion happens when the last smart pointer goes out of scope.

Creating Threads

- You must allocate your **`Thread`** objects on the heap - stack-allocated Thread objects will result in deallocation errors:

```
ReaderThread thread;
IceUtil::ThreadPtr t = &thread; // Bad news!!!
```

- This is wrong because the destructor of **`t`** will eventually call delete, which has undefined behavior for a stack-allocated object.
- Similarly, you must use a **`ThreadPtr`** for an allocated thread.
  - Do not attempt to explicitly delete a a thread:

```
Thread* t = new ReaderThread();
// ...
delete t; // Disaster!
```

- This will result in a double deallocation of the thread because the thread's destructor will call **`delete this`**.

The `ThreadControl` Class

- The **`start`** method returns an object of type **`ThreadControl`**
- The member functions of **`ThreadControl`** behave as follows:
- **`ThreadControl`**
  - The default constructor returns a **`ThreadControl`** object that refers to the calling thread. This allows you to get a handle to the current (calling) thread even if you do not have saved a handle to that thread previously. For example:

```
IceUtil::ThreadControl self; // Get handle to self
cout << self.id() << endl; // Print thread ID
```

  - This example also explains why we have two classes, **`Thread`** and **`Thread-Control`**: without a separate **`ThreadControl`**, it would not be possible to obtain a handle to an arbitrary thread.
  - This code works even if the calling thread was not created by the Ice run time; for example, you can create a **`ThreadControl`** object for a thread that was created by the operating system
  - The (implicit) copy constructor and assignment operator create a **`ThreadControl`** object that refers to the same underlying thread as the source **`ThreadControl`** object.

The `ThreadControl` Class

- **`join`**
  - Suspends the calling thread until the thread on which **`join`** is called has terminated. For example:
  - **`IceUtil::ThreadPtr t = new ReaderThread; // Create a thread`**
  - **`IceUtil::ThreadControl tc = t->start(); // Start it`**
  - **`tc.join(); // Wait for it`**
  - If the reader thread has finished by the time the creating thread calls **`join`**, the call to **`join`** returns immediately; otherwise, the creating thread is suspended until the reader thread terminates.
  - The **`join`** method of a thread must be called from only one other thread, that is, only one thread can wait for another thread to terminate.
  - Calling **`join`** on a thread from more than one other thread has undefined behavior.
  - Calling **`join`** on a thread that was previously joined with or calling join on a detached thread has undefined behavior.
  - You must **`join`** with each thread you create; failure to join with a thread has undefined behavior.

The `ThreadControl` Class

- **`detach`**
  - Detaches a thread. Once a thread is detached, it cannot be joined with.
  - Calling **`detach`** on an already detached thread, or calling detach on a thread that was previously joined with has undefined behavior.
  - If you have detached a thread, you must ensure that the detached thread has terminated before your program leaves its main function. This means that, because detached threads cannot be joined with, they must have a life time that is shorter than that of the main thread.
- **`sleep`**
  - This method suspends the calling thread for the amount of time specified by the **`Time`** parameter

The `ThreadControl` Class

- **`yield`**
  - This method causes the calling thread to relinquish the CPU, allowing another thread to run.
- **`operator==`**
- **`operator!=`**
  - These operators compare thread IDs.
  - **`operator<`** is not provided because it cannot be implemented portably.
  - These operators yield meaningful results only for threads that have not been detached or joined with.

The `ThreadControl` Class

- You must adhere to a few rules when using threads to avoid undefined behavior:
  - Do not allow run to throw an exception.
  - Do not join with or detach a thread that you have not created yourself.
  - For every thread you create, you must either join with that thread exactly once or detach it exactly once; failure to do so may cause resource leaks.
  - Do not call join on a thread from more than one other thread.
  - Do not leave main until all other threads you have created have terminated.
  - Do not leave main until after you have destroyed all Ice::Communicator objects you have created.
  - A common mistake is to call yield from within a critical region.
    - Doing so is usually pointless because the call to yield will look for another thread that can be run but, when that thread is run, it will most likely try to enter the critical region that is held by the yielding thread and go to sleep again.
    - At best, this achieves nothing and, at worst, it causes many additional context switches for no gain.
    - If you call yield, do so only in circumstances where there is at least a fair chance that another thread will actually be able to run and do something useful.

The ThreadControl Example

```
#include <vector>
#include <IceUtil/Thread.h>
// ...
Queue q;
class ReaderThread : public IceUtil::Thread {
  virtual void run() {
    for (int i = 0; i < 100; ++i)
      cout << q.get() << endl;
  }
};
class WriterThread : public IceUtil::Thread {
  virtual void run() {
    for (int i = 0; i < 100; ++i)
      q.put(i);
  }
};
```

```
int main()
{
  vector<IceUtil::ThreadControl> threads;
  int i;
  // Create five reader threads and start them
  //
  for (i = 0; i < 5; ++i) {
    IceUtil::ThreadPtr t = new ReaderThread;
    threads.push_back(t->start());
  }
  // Create five writer threads and start them
  //
  for (i = 0; i < 5; ++i) {
    IceUtil::ThreadPtr t = new WriterThread;
    threads.push_back(t->start());
  }
  // Wait for all threads to finish
  //
  for (vector<IceUtil::ThreadControl>::iterator i
         = threads.begin();
         i != threads.end(); ++i) {
    i->join();
  }
}
```

Portable Signal Handling

- The **`IceUtil::CtrlCHandler`** class provides a portable mechanism to handle Ctrl+C and similar signals sent to a C++ process.
  - On Windows, **`IceUtil::CtrlCHandler`** is a wrapper for **`SetConsoleCtrlHandler`**;
  - On POSIX platforms, it handles SIGHUP, SIGTERM and SIGINT with a dedicated thread that waits for these signals using **`sigwait`**.
  - Signals are handled by a callback function implemented and registered by the user.
  - The callback is a simple function that takes an **`int`** (the signal number) and returns **`void`**; it should not throw any exception:

Portable Signal Handling

```
namespace IceUtil {
  typedef void (*CtrlCHandlerCallback)(int);
  class CtrlCHandler {
    public:
      CtrlCHandler(CtrlCHandlerCallback = 0);
      ~CtrlCHandler();
      void setCallback(CtrlCHandlerCallback);
      CtrlCHandlerCallback getCallback() const;
};
}
```

Portable Signal Handling

- The member functions of **`CtrlCHandler`** behave as follows:
- Constructor
  - Constructs an instance with a callback function.
  - Only one instance of **`CtrlCHandler`** can exist in a process at a given moment in time.
  - On POSIX platforms, the constructor masks SIGHUP, SIGTERM and SIGINT, then starts a thread that waits for these signals using **`sigwait`**.
  - For signal masking to work properly, it is imperative that the **`CtrlCHandler`** instance be created before starting any thread, and in particular before initializing an Ice communicator.

Portable Signal Handling

Destructor

- Destroys the instance, after which the default signal processing behavior is restored on Windows (**`TerminateProcess`**).
- On POSIX platforms, the "**`sigwait`**" thread is cancelled and joined, but the signal mask remains unchanged, so subsequent signals are ignored.

- **`setCallback`**
  - Sets a new callback function.
- **`getCallback`**
  - Gets the current callback function.
- It is legal to specify a value of zero (0) for the callback function, in which case signals are caught and ignored until a non-zero callback function is set.