The C++ Thread Classes

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. For example, you might need a dedicated thread that responds to input from a user interface. And, 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.

On this page:

The C++ `Thread` Class

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

C++

```
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 :virtual public Shared {
    public:
        virtual void run() = 0;

        ThreadControl start(size_t stBytes = 0);
        ThreadControl start(size_t stBytes, int priority);
        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 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). Note that 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 remaining member functions behave as follows:

- `start(size_t stBytes = 0)`
  `start(size_t stBytes, int priority)`
  This member function starts a newly-created thread (that is, calls the `run` method). The `stBytes` 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.

  You can also specify a priority for the thread. (If you do not supply a priority, the thread is created with the system default priority.) The priority value is system-dependent; on POSIX systems, the value must be a legal value for the `SCHED_RR` real-time scheduling policy. (`SCHED_RR` requires root privileges.) On Windows systems, the priority value is passed through to the Windows `setThreadPriority` function. Priority Inversion in C++ provides information about how you can deal with priority inversion.

  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`.

If the calling thread passes an invalid priority or, on POSIX systems, does not have root privileges, `start` raises `ThreadSyscallException`.

- `getThreadControl`
  This member function returns a `ThreadControl` 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. Note also that `pthread_t` is, strictly-speaking, an opaque type, so you should not make any assumptions about what you can do with a thread ID.
- `isAlive`
  This method 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:

C++

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

- `operator==`
  `operator!=`
  `operator<`
  These member functions compare the in-memory address of two threads. They are provided so you can use `Thread` objects in sorted STL containers.

Note that `IceUtil` also defines the type `ThreadPtr`. This is the usual reference-counted smart pointer to guarantee automatic clean-up: the `Thread` destructor calls `delete this` once its reference count drops to zero.

Implementing Threads in C++

To illustrate how to implement threads, consider the following code fragment:

C++

```
#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);
    }
};
```

This code fragment defines two classes, `ReaderThread` and `WriterThread`, that inherit from `IceUtil::Thread`. Each class implements the pure virtual `run` method it inherits from its base class. For this simple example, a writer thread places the numbers from 1 to 100 into an instance of the thread-safe `Queue` class we defined in our discussion of monitors, and a reader thread retrieves 100 numbers from the queue and prints them to `stdout`.

Creating Threads in C++

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

C++

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

Note that we assign the return value from `new` to a smart pointer of type `ThreadPtr`. This ensures that we do not suffer a memory leak:

1. When the thread is created, its reference count is set to zero.
2. Prior to calling `run` (which is called by the `start` method), `start` increments the reference count of the thread to 1.
3. 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.
4. 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.

Note that, for all this to work, you *must* allocate your `Thread` objects on the heap — stack-allocated `Thread` objects will result in deallocation errors:

C++

```
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 thread:

C++

```
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`.

It is legal for a thread to call `start` on itself from within its own constructor. However, if so, the thread must not be (very) short lived:

C++

```
class ActiveObject : public Thread() {
public:
    ActiveObject() {
        start();
    }

    void done() {
        getThreadControl().join();
    }

    virtual void run() {
        // *Very* short lived...
    }
};
typedef Handle<ActiveObject> ActiveObjectPtr;

// ...

ActiveObjectPtr ao = new ActiveObject;
```

With this code, it is possible for `run` to complete before the assignment to the smart pointer `ao` completes; in that case, `start` will call `delete this;` before it returns and `ao` ends up deleting an already-deleted object. However, note that this problem can arise only if `run` is indeed very short-lived and moreover, the scheduler allows the newly-created thread to run to completion before the assignment of the return value of `operator new` to `ao` takes place. This is highly unlikely to happen — if you are concerned about this scenario, do not call `start` from within a thread's own constructor. That way, the smart pointer is assigned first, and the thread started second, so the problem cannot arise.

The C++ `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 had not previously saved a handle to that thread. For example:

  C++

  ```
  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 `ThreadControl`: without a separate `ThreadControl`, it would not be possible to obtain a handle to an arbitrary thread. (Note that 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.

  Note that the constructor is overloaded. For Windows, the signature is

  C++

  ```
  ThreadControl(HANDLE, DWORD);
  ```

  For Unix, the signature is

C++

```
ThreadControl(pthread_t);
```

These constructors allow you to create a `ThreadControl` object for the specified thread.

- `join`
  This method suspends the calling thread until the thread on which `join` is called has terminated. For example:

  C++

  ```
  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.

  Note that 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.

- `detach`
  This method 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.

  Note that, 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` class.

- `yield`
  This method causes the calling thread to relinquish the CPU, allowing another thread to run.

- `operator==`
  `operator!=`
  These operators compare thread IDs. (Note that `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.

As for all the synchronization primitives, 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 (or use the `Ice::Application` class).
- 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.

C++ Thread Example

Following is a small example that uses the `Queue` class we defined in our discussion of monitors. We create five writer and five reader threads. The writer threads each deposit 100 numbers into the queue, and the reader threads each retrieve 100 numbers and print them to `stdout`:

C++

```
#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();
    }
}
```

The code uses the `threads` variable, of type `vector<IceUtil::ThreadControl>`, to keep track of the created threads. The code creates five reader and five writer threads, storing the `ThreadControl` object for each thread in the `threads` vector. Once all the threads are created and running, the code joins with each thread before returning from `main`.

Note that you must not leave `main` without first joining with the threads you have created: many threading libraries crash if you return from `main` with other threads still running. (This is also the reason why you must not terminate a program without first calling `Communicator::destroy`; the `destroy` implementation joins with all outstanding threads before it returns.)

See Also

- Smart Pointers for Classes
- The Server-Side main Function in C++
- The C++ Monitor Class
- The Ice Threading Model
- The C++ Time Class