Thread Safety

The Ice run time itself is fully thread safe, meaning multiple application threads can safely call methods on objects such as communicators, object adapters, and proxies without synchronization problems. As a developer, you must also be concerned with thread safety because the Ice run time can dispatch multiple invocations concurrently in a server. In fact, it is possible for multiple requests to proceed in parallel within the same servant and within the same operation on that servant. It follows that, if the operation implementation manipulates non-stack storage (such as member variables of the servant or global or static data), you must interlock access to this data to avoid data corruption.

The need for thread safety in an application depends on its configuration. Using the default thread pool configuration typically makes synchronization unnecessary because at most one operation can be dispatched at a time. Thread safety becomes an issue once you increase the maximum size of a thread pool.

Ice uses the native synchronization and threading primitives of each platform. For C++ users, Ice provides a collection of convenient and portable wrapper classes for use by Ice applications.

On this page:

Threading Issues with Marshaling

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. In C++, the only relevant case is returning an instance of a Slice class, either directly or nested as a member of another type. In Java, .NET, and the scripting languages, Slice structures, sequences, and dictionaries are also affected.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following Java implementation:

Java

```
public class GridI extends _GridDisp
{
    GridI()
    {
        _grid = // ...
    }

    public int[][]
    getGrid(Ice.Current curr)
    {
        return _grid;
    }

    public void
    setValue(int x, int y, int val, Ice.Current curr)
    {
        _grid[x][y] = val;
    }

    private int[][] _grid;
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned array in preparation to send a reply message, it is possible for another thread to dispatch the `setValue` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `setValue` operations would not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `setValue` replaces `_grid` with a copy that contains the new element, leaving the previous contents of `_grid` unchanged:

Java

```
public class GridI extends _GridDisp
{
    ...

    public synchronized int[][]
    getGrid(Ice.Current curr)
    {
        return _grid;
    }

    public synchronized void
    setValue(int x, int y, int val, Ice.Current curr)
    {
        int[][] newGrid = // shallow copy...
        newGrid[x][y] = val;
        _grid = newGrid;
    }

    ...
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Finally, a third approach changes accessor operations to use AMD in order to regain control over marshaling. After annotating the `getGrid` operation with `amd` metadata, we can revise the servant as follows:

Java

```
public class GridI extends _GridDisp
{
    ...

    public synchronized void
    getGrid_async(AMD_Grid_getGrid cb, Ice.Current curr)
    {
        cb.ice_response(_grid);
    }

    public synchronized void
    setValue(int x, int y, int val, Ice.Current curr)
    {
        _grid[x][y] = val;
    }

    ...
}
```

Normally, AMD is used in situations where the servant needs to delay its response to the client without blocking the calling thread. For `getGrid`, that is not the goal; instead, as a side-effect, AMD provides the desired marshaling behavior. Specifically, the Ice run time marshals the reply to an asynchronous request at the time the servant invokes `ice_response` on the AMD callback object. Because `getGrid` and `setValue` are synchronized, this guarantees that the data remains in a consistent state during marshaling.

Thread Creation and Destruction Hooks

On occasion, it is necessary to intercept the creation and destruction of threads created by the Ice run time, for example, to interoperate with libraries that require applications to make thread-specific initialization and finalization calls (such as COM's `CoInitializeEx` and `CoUninitialize`). Ice provides callbacks to inform an application when each run-time thread is created and destroyed. For C++, the callback class looks as follows:

C++

```
class ThreadNotification : public IceUtil::Shared {
public:
    virtual void start() = 0;
    virtual void stop() = 0;
};
typedef IceUtil::Handle<ThreadNotification> ThreadNotificationPtr;
```

To receive notification of thread creation and destruction, you must derive a class from `ThreadNotification` and implement the `start` and `stop` member functions. These functions will be called by the Ice run by each thread as soon as it is created, and just before it exits. You must install your callback class in the Ice run time when you create a communicator by setting the `threadHook` member of the `InitializationData` structure.

For example, you could define a callback class and register it with the Ice run time as follows:

C++

```
class MyHook : public virtual Ice::ThreadNotification {
public:
    void start()
    {
        cout << "start: id = " << ThreadControl().id() << endl;
    }
    void stop()
    {
        cout << "stop: id = " << ThreadControl().id() << endl;
    }
};

int
main(int argc, char* argv[])
{
    // ...

    Ice::InitializationData id;
    id.threadHook = new MyHook;
    communicator = Ice::initialize(argc, argv, id);

    // ...
}
```

The implementation of your `start` and `stop` methods can make whatever thread-specific calls are required by your application.

For Java and C#, `Ice.ThreadNotification` is an interface:

Java/C#

```
public interface ThreadNotification {
    void start();
    void stop();
}
```

To receive the thread creation and destruction callbacks, you must derive a class from this interface that implements the `start` and `stop` methods, and register an instance of that class when you create the communicator. (The code to do this is analogous to the C++ version.)

For Python, the interface is:

Python

```
class ThreadNotification(object):
    def __init__(self):
        pass

    # def start():
    # def stop():
```

The Ice run time calls the `start` and `stop` methods of the class instance you provide to `Ice.initialize` when it creates and destroys threads.

Installing Thread Hooks with a Plug-in

The thread hook facility described above requires that you modify a program's source code in order to receive callbacks when threads in the Ice run time are created and destroyed. It is also possible to install thread hooks using the Ice plug-in facility, which is useful for adding thread hooks to an existing program that you cannot (or prefer not to) modify.

Ice provides a base class named `ThreadHookPlugin` for C++, Java, and C# that supplies the necessary functionality. The C++ class definition is shown below:

C++

```
namespace Ice {
class ThreadHookPlugin : public Ice::Plugin {
public:

    ThreadHookPlugin(const CommunicatorPtr& communicator, const ThreadNotificationPtr&);

    virtual void initialize();

    virtual void destroy();
};
}
```

The equivalent definitions for Java and C# are quite similar and therefore not presented here.

The `ThreadHookPlugin` constructor installs the given `ThreadNotification` object into the specified communicator. The `initialize` and `destroy` methods are empty, but you can subclass `ThreadHookPlugin` and override these methods if necessary.

In order to create a thread hook plug-in, you must do the following:

- Define and export a factory class (for Java and C#) or factory function (for C++) that returns an instance of `ThreadHookPlugin`, as described in the plug-in API.
- Implement the `ThreadNotification` object that you will pass to the `ThreadHookPlugin` constructor.
- Package your code into a format that is suitable for dynamic loading, such as a shared library or DLL for C++ or an assembly for C#.

To install your plug-in, use a configuration property like the one shown below:

```
Ice.Plugin.MyThreadHookPlugin=MyHooks:createPlugin ...
```

The first component of the property value represents the plug-in's entry point. For C++, this value includes the abbreviated name of the shared library or DLL (`MyHooks`) and the name of a factory function (`createPlugin`).

If your property value is language-specific and the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice run time for a certain language. For example, here is the C++-specific version:

```
Ice.Plugin.MyThreadHookPlugin.cpp=MyHooks:createPlugin ...
```

For more information, see Ice Plug-In Properties.

See Also

- Communicator Initialization
- Threads and Concurrency with C++
- Plug-in Facility
- Plug-in API
- Ice Plug-In Properties