Bionic Buffalo Tech Note #49

Thread Support in the `gibraltar` Object Request Broker

last revised Saturday 2003.05.24

Introduction

The `gibraltar` object request broker (ORB) is safe under limited circumstances for use in threaded environments. This Tech Note describes the use and limitations of threading support in `gibraltar`.

`gibraltar` may be built with or without thread support. If the symbol `TBT_THREADED_LIBRARY` is defined as non-zero, then `gibraltar` will be compiled to utilize primitives from a subset of pthreads. If `TBT_THREADED_LIBRARY` is zero, then `gibraltar` will assume a single-threaded environment.

In its single-threaded form, `gibraltar` routines will not take any steps to protect its data structures from improper access by simultaneously executing threads of control. Built this way, it may be used in non-threaded environments such as DOS, or in single threaded processes in environments such as Unix. In single-threaded mode, the *pthread.h* header isn't required, and it is not necessary to link with the pthread library. The rest of this Tech Note will ignore the single-threaded option, and instead will focus on code built with pthreads support.

When compiled with thread support, `gibraltar` is safe for use in threaded environments as follows:

> `gibraltar` procedures are reentrant and thread-safe, and may be called by any thread
>
> `gibraltar` procedures do not block; however, operation invocations on non-`gibraltar` objects made through `gibraltar` may block until the invocation completes or fails, or because the object adapter's thread policy causes the method invocation to block for other reasons
>
> threads using `gibraltar` procedures are *not* cancel safe; however, a thread may be terminated after a specified action (destruction of the thread's environment structure) taken by the thread itself
>
> `gibraltar` procedures are *not* fork safe

> `gibraltar` procedures are *not* async signal safe

If an application requires fewer restrictions (for instance, if it must be cancel safe), then a different ORB (such as `egypt` or `taiwan`) should be used.

The `gibraltar` ORB is available at *http://www.tatanka.com/prod/info/gibraltar.html.*

Threads and the `CORBA_Environment`

Each thread is associated with a unique copy of the `CORBA_Environment`. State information used by the ORB for a thread is associated with that thread's `CORBA_Environment`. A thread acquires a pointer to its `CORBA_Environment` by calling `tbt_get_environment()`.

The first time a thread calls `tbt_get_environment()`, a new environment will be created for that thread. Subsequent calls to `tbt_get_environment()` will return the same environment, without creating a new environment.

A thread can destroy its environment by calling `tbt_destroy_environment()`. This can be done at any time except from within a servant method. Once the thread's environment is destroyed, the ORB is oblivious to the thread, and it may be cancelled or suicide if necessary. A thread whose environment is destroyed also may call `tbt_get_environment()`, and a new environment will be created for that thread.

Operation Invocation and Connection Management

In operation invocations by local (same process) clients, threads are handled as follows:

> For local servants, the method is called in the same thread used to call the operation.
>
> For remote (different process or different network node) servants, the calling thread blocks within the ORB until the operation is completed or fails. (Each pending remote operation involves a separate remote connection; simultaneous connections to the same server are not multiplexed.)

When initialized, the `gibraltar` ORB creates a thread whose sole purpose is to accept incoming connections. The total number of simultaneous connections is limited by a configurable value, which determines the maximum number of remote clients which can be connected at the same time. A thread is also established for each potential simultaneous connection. (These are the *connection threads*.) Each connection might result in zero or more active requests. Another configurable value determines the maximum number of simultaneous requests, and a task is created for each. (These are the *remote request threads*.) Finally, a task is created whose purpose is to manage request and operation timeouts. Thus, in addition to the threads created by the application itself, the ORB creates *max connections* + *max requests* + 2

additional threads. (The possible total number of open sockets required can be

max connections + *number of application threads* + 1

in this design.)

Each local servant is associated with a Portable Object Adapter, or POA. When each POA is created, that POA is associated with a thread policy. There are three possible thread policy values:

`PortableServer::ORB_CTRL_MODEL`. In this model, the POA and its servants may simultaneously process requests from any or all application threads and from any or all remote request threads. Servants must be thread-safe.

`PortableServer::SINGLE_THREAD_MODEL`. In this model, for each POA, no more than one thread my invoke POA or thread operations at a given time. If a POA is in use by one thread, then any other thread attempting to use that POA will block until the first thread no longer uses that POA. Servants may not be thread-safe. However, because a servant may recursively invoke itself, or may invoke other servants of the same POA, servants may need to be reentrant. Under some circumstances, there also may be a potential for deadlock, if servants of different POAs are prevented from calling one another due to the single-thread restriction.

`PortableServer::MAIN_THREAD_MODEL`. In this model, all POAs with this policy, taken together, are allowed use by only one thread at a time. The result is similar to what would occur if all their servants were combined into one single-thread POA. The potential for deadlock among servants of POAs using this policy is eliminated, but reentrant code still may be necessary.

Although thread dispatch deadlocks might be prevented, there remains the possibility of resource deadlocks in servants which maintain or alter any sort of state information. Unless stateless methods are used, the programmer must still be careful about resource deadlocks. For example, if one method has exclusive access to a file needed by another method, any serialization or queueing of the second method's request is the responsibility of the servant or application programmer, not of the ORB or POA.

In `gibraltar`, each type of thread creates has the same priority, although the priorities are configurable at build time and may differ from type to type. Initializers for threads, mutexes, and conditions use default values (without attributes). Developers requiring more control over threads and resource allocation should use an ORB (such as `taiwan`) which supports the Real-Time CORBA specification.

Developers requiring more control over connections and over how messages and requests are processed should use an ORB (such as `taiwan`) which supports the CORBA Messaging specification.

The `gibraltar` Global Environment

Note: The information in this section is implementation specific, and portable applications should not rely on it.

All state information, data structures, tables, and run-time information owned by `gibraltar` are kept in, or linked from, the *global environment*. The global environment is a structure, so-named to distinguish it from the individual thread environments. By following pointers from the global environment, all of the thread environments can be found, as can the tables allowing lookup of object information from object references. The global environment is found by dereferencing a global pointer. A mutex protects the pointer and the linked global environment data structures.

A thread's `CORBA_Environment` is a substructure within the associated thread environment structure.

A key (from `pthread_key_create()`) is associated with the addresses of the thread environment structures. Each thread can locate its own thread environment structure (if any) by applying `pthread_getspecific()` to the key. This is cheaper than walking the data structures from the global environment, and doesn't require that the global environment be locked during the lookup operation.
