The OMNI Thread Abstraction

Tristan Richardson
AT&T Laboratories Cambridge

Revised November 2001

1 Introduction

The OMNI thread abstraction is designed to provide a common set of thread operations for use in programs written in C++. Programs written using the abstraction should be much easier to port between different architectures with different underlying threads primitives.

The programming interface is designed to be similar to the C language interface to POSIX threads (IEEE draft standard 1003.1c — previously 1003.4a, often known as "pthreads" [POSIX94]).

Much of the abstraction consists of simple C++ object wrappers around pthread calls. However for some features such as thread-specific data, a better interface can be offered because of the use of C++.

Some of the more complex features of pthreads are not supported because of the difficulty of ensuring the same features can be offered on top of other thread systems. Such features include thread cancellation and complex scheduling control (though simple thread priorities are supported).

The abstraction layer is currently implemented for the following architectures / thread systems:

- Solaris 2.x using pthreads draft 10
- Solaris 2.x using solaris threads (but pthreads version is now standard)
- Alpha OSF1 using pthreads draft 4
- Windows NT using NT threads
- Linux 2.x using Linuxthread 0.5 (which is based on pthreads draft 10)
- Linux 2.x using MIT pthreads (which is based on draft 8)
- ATMos using pthreads draft 6 (but not Virata ATMos)

See the `omnithread.h` header file for full details of the API. The descriptions below assume you have some previous knowledge of threads, mutexes, condition variables and semaphores. Also refer to other documentation ([Birrell89], [POSIX94]) for further explanation of these ideas (particularly condition variables, the use of which may not be particularly intuitive when first encountered).

2 Synchronisation objects

Synchronisation objects are used to synchronise threads within the same process. There is no inter-process synchronisation provided. The synchronisation objects provided are mutexes, condition variables and counting semaphores.

2.1 Mutex

An object of type `omni_mutex` is used for mutual exclusion. It provides two operations, `lock()` and `unlock()`. The alternative names `acquire()` and `release()` can be used if preferred. Behaviour is undefined when a thread attempts to lock the same mutex again or when a mutex is locked by one thread and unlocked by a different thread.

2.2 Condition Variable

A condition variable is represented by an `omni_condition` and is used for signalling between threads. A call to `wait()` causes a thread to wait on the condition variable. A call to `signal()` wakes up at least one thread if any are waiting. A call to `broadcast()` wakes up all threads waiting on the condition variable.

When constructed, a pointer to an `omni_mutex` must be given. A condition variable `wait()` has an implicit mutex `unlock()` and `lock()` around it. The link between condition variable and mutex lasts for the lifetime of the condition variable (unlike pthreads where the link is only for the duration of the wait). The same mutex may be used with several condition variables.

A wait with a timeout can be achieved by calling `timed_wait()`. This is given an absolute time to wait until. The routine `omni_thread::get_time()` can be used to turn a relative time into an absolute time. `timed_wait()` returns `true` if the condition was signalled, `false` if the time expired before the condition variable was signalled.

2.3 Counting semaphores

An `omni_semaphore` is a counting semaphore. When created it is given an initial unsigned integer value. When `wait()` is called, the value is decremented if nonzero. If the value is zero then the thread blocks instead. When `post()` is called, if any threads are blocked in `wait()`, exactly one thread is woken. If no threads were blocked then the value of the semaphore is incremented.

If a thread calls `try_wait()`, then the thread won't block if the semaphore's value is 0, returning `false` instead.

There is no way of querying the value of the semaphore.

3 Thread object

A thread is represented by an `omni_thread` object. There are broadly two different ways in which it can be used.

The first way is simply to create an `omni_thread` object, giving a particular function which the thread should execute. This is like the POSIX (or any other) C language interface.

The second method of use is to create a new class which inherits from `omni_thread`. In this case the thread will execute the `run()` member function of the new class. One advantage of this scheme is that thread-specific data can be implemented simply by having data members of the new class.

When constructed a thread is in the "new" state and has not actually started. A call to `start()` causes the thread to begin executing. A static member function `create()` is provided to construct and start a thread in a single call. A thread exits by calling `exit()` or by returning from the thread function.

Threads can be either detached or undetached. Detached threads are threads for which all state will be lost upon exit. Other threads cannot determine when a detached thread will disappear, and therefore should not attempt to access the thread object unless some explicit synchronisation with the detached thread guarantees that it still exists.

Undetached threads are threads for which storage is not reclaimed until another thread waits for its termination by calling `join()`. An exit value can be passed from an undetached thread to the thread which joins it.

Detached / undetached threads are distinguished on creation by the type of function they execute. Undetached threads execute a function which has a `void*` return type, whereas detached threads execute a function which has a `void` return type. Unfortunately C++ member functions are not allowed to be distinguished simply by their return type. Thus in the case of a derived class of `omni_thread` which needs an undetached thread, the member function executed by the thread is called `run_undetached()` rather than `run()`, and it is started by calling `start_undetached()` instead of `start()`.

The abstraction currently supports three priorities of thread, but no guarantee is made of how this will affect underlying thread scheduling. The three priorities are `PRIORITY_LOW`, `PRIORITY_NORMAL` and `PRIORITY_HIGH`. By default all threads run at `PRIORITY_NORMAL`. A different priority can be specified on thread creation, or while the thread is running using `set_priority().` A thread's current priority is returned by `priority()`.

Other functions provided are `self()` which returns the calling thread's `omni_thread` object, `yield()` which requests that other threads be allowed to run, `id()` which returns an integer id for the thread for use in debugging, `state()`, `sleep()` and `get_time()`.

4 Per-thread data

omnithread supports per-thread data, via member functions of the `omni_thread` object.

First, you must allocate a key for with the `omni_thread::allocate_key()` function. Then, any object whose class is derived from `omni_thread::value_t` can be stored using the `set_value()` function. Values are retrieved or removed with `get_value()` and `remove_value()` respectively.

When the thread exits, all per-thread data is deleted (hence the base class with virtual destructor).

Note that the per-thread data functions are **not** thread safe, so although you can access one thread's storage from another thread, there is no concurrency control. Unless you really know what you are doing, it is best to only access per-thread data from the thread it is attached to.

5 Using OMNI threads in your program

Obviously you need to include the `omnithread.h` header file in your source code, and link in the omnithread library with your executable. Because there is a single `omnithread.h` for all platforms, certain preprocessor defines must be given as compiler options. The easiest way to do this is to study the makefiles given in the examples provided with this distribution. If you are to include OMNI threads in your own development environment, these are the necessary preprocessor defines:

Platform	Preprocessor Defines
Sun Solaris 2.x	`-D__sunos__ -D__sparc__ -D__OSVERSION__=5 -DSVR4 -DUsePthread -D_REENTRANT`
x86 Linux 2.0 with linuxthreads 0.5	`-D__linux__ -D__i86__ -D__OSVERSION__=2 -D_REENTRANT`
Digital Unix 3.2	`-D__osf1__ -D__alpha__ -D__OSVERSION__=3 -D_REENTRANT`
Windows NT	`-D__NT__ -MD`

6 Threaded I/O shutdown for Unix

or, how one thread should tell another thread to shut down when it might be doing a blocking call on a socket.

If you are using omniORB, you don't need to worry about all this, since omniORB does it for you. This section is only relevant if you are using omnithread in your own socket-based programming. It is also seriously out of date.

Unfortunately there doesn't seem to be a standard way of doing this which works across all Unix systems. I have investigated the behaviour of Solaris 2.5 and Digital Unix 3.2. On Digital Unix everything is fine, as the obvious method using shutdown() seems to work OK. Unfortunately on Solaris shutdown can only be used on a connected socket, so we need devious means to get around this limitation. The details are summarised below:

6.1 read()

Thread A is in a loop, doing `read(sock)`, processing the data, then going back into the read.

Thread B comes along and wants to shut it down — it can't cancel thread A since (i) working out how to clean up according to where A is in its loop is a nightmare, and (ii) this isn't available in omnithread anyway.

On Solaris 2.5 and Digital Unix 3.2 the following strategy works:

Thread B does `shutdown(sock,2)`.

At this point thread A is either blocked inside `read(sock)`, or is elsewhere in the loop. If the former then read will return 0, indicating that the socket is closed. If the latter then eventually thread A will call `read(sock)` and then this will return 0. Thread A should `close(sock)`, do any other tidying up, and exit.

If there is another point in the loop that thread A can block then obviously thread B needs to be aware of this and be able to wake it up in the appropriate way from that point.

6.2 accept()

Again thread A is in a loop, this time doing an accept on listenSock, dealing with a new connection and going back into accept. Thread B wants to cancel it.

On Digital Unix 3.2 the strategy is identical to that for read:

Thread B does `shutdown(listenSock,2)`. Wherever thread A is in the loop, eventually it will return `ECONNABORTED` from the accept call. It should `close(listenSock)`, tidy up as necessary and exit.

On Solaris 2.5 thread B can't do `shutdown(listenSock,2)` — this returns `ENOTCONN`. Instead the following strategy can be used:

First thread B sets some sort of "shutdown flag" associated with listenSock. Then it does `getsockaddr(listenSock)` to find out which port listenSock is on (or knows already), sets up a socket dummySock, does `connect(dummySock, this host, port)` and finally does `close(dummySock)`.

Wherever thread A is in the loop, eventually it will call `accept(listenSock)`. This will return successfully with a new socket, say connSock. Thread A then checks to see if the "shutdown flag" is set. If not, then it's a normal connection. If it is set, then thread A closes listenSock and connSock, tidies up and exits.

6.3 write()

Thread A may be blocked in write, or about to go in to a potentially-blocking write. Thread B wants to shut it down.

On Solaris 2.5:

Thread B does `shutdown(sock,2)`.

If thread A is already in `write(sock)` then it will return with `ENXIO`. If thread A calls write after thread B calls shutdown this will return `EIO`.

On Digital Unix 3.2:

Thread B does `shutdown(sock,2)`.

If thread A is already in `write(sock)` then it will return the number of bytes written before it became blocked. A subsequent call to write will then generate `SIGPIPE` (or `EPIPE` will be returned if `SIGPIPE` is ignored by the thread).

6.4 connect()

Thread A may be blocked in connect, or about to go in to a potentially-blocking connect. Thread B wants to shut it down.

On Digital Unix 3.2:

Thread B does `shutdown(sock,2)`.

If thread A is already in `connect(sock)` then it will return a successful connection. Subsequent reading or writing will show that the socket has been shut down (i.e. read returns 0, write generates `SIGPIPE` or returns `EPIPE`). If thread A calls connect after thread B calls shutdown this will return `EINVAL`.

On Solaris 2.5:

There is no way to wake up a thread which is blocked in connect. Instead Solaris forces us through a ridiculous procedure whichever way we try it. One way is this:

First thread A creates a pipe in addition to the socket. Instead of shutting down the socket, thread B simply writes a byte to the pipe.

Thread A meanwhile sets the socket to non-blocking mode using `fcntl(sock, F_SETFL, O_NONBLOCK)`. Then it calls connect on the socket — this will return `EINPROGRESS`. Then it must call `select()`, waiting for either sock to become writable or for the pipe to become readable. If select returns that just sock is writable then the connection has succeeded. It then needs to set the socket back to blocking mode using `fcntl(sock, F_SETFL, 0)`. If instead select returns that the pipe is readable, thread A closes the socket, tidies up and exits.

An alternative method is similar but to use polling instead of the pipe. Thread B justs sets a flag and thread A calls select with a timeout, periodically waking up to see if the flag has been set.

References

[POSIX94] *Portable Operating System Interface (POSIX) Threads Extension*, P1003.1c Draft 10, IEEE, September 1994.

[Birrell89] *An Introduction to Programming with Threads*, Research Report 35, DEC Systems Research Center, Palo Alto, CA, January 1989.