Nxxx: Proposed changes to the C1X <threads.h>

Mr. Niall Douglas BSc MA MBS MCollT <http://www.nedproductions.biz/>

July 2011

My thanks in particular to Hans Boehm without whose detailed feedback this proposal would look completely different. My thanks are also due to John Benito for his seemingly never tiring efforts.

Table of Contents

[List of issues and proposed solutions: 1](#_Toc298682969)

[The proposed C1X permit object 2](#_Toc298682970)

[Proposed C1X permit object specification: 3](#_Toc298682971)

[Appendix A: Example C1X permit object implementation 6](#_Toc298682972)

# List of issues and proposed solutions:

The latest iteration of the C1X draft fixes most of the remaining problems in the proposed C1X threading library. However, just a few more remain in my opinion:

1. **There is no explicit way of initialising a thrd\_t to a null value**

Why this is a problem: Firstly there is the problem of undefined initialisation. Secondly, one often wishes to record which thread owns something, and equally if *no* thread owns something. Without a null value one cannot legally record no ownership.

Proposed solution: Adding a THRD\_T\_INIT would solve the problem.

Cost of proposed solution: Negligible.

1. **There is no way of performing a timed join to a thread**

Why this is a problem: Certainly in my own coding, one wants to trap deadlocks during thread shut down and dump state to a debug file which can be sent to me for analysis. Right now under POSIX threads I typically have a wait object which a thread signals just before exit – this allows me to emulate a timed join, so if the thread does not exit in say ten seconds I can take remedial action. This is a lot of unnecessary complexity though when timed joins ought to be available. Furthermore, they can be useful for polling to see if an asynchronous job has completed, and out of simple consistency where every other waitable function has a timed variant we ought to have this here too.

Proposed solution: Add a thrd\_timedjoin().

Cost of proposed solution: Microsoft Windows already can do timed joins, as can Linux and QNX via pthread\_timedjoin\_np(). Looking at the source code for the implementation of pthread\_join() on BSD suggests that adding support on BSD derived platforms would be trivial. Therefore the cost ought to be between negligible and trivial.

1. **There is no standard defined asynchronous job completion notification object (a.k.a. a “permit object”)**

Why this is a problem: There are many occasions in threaded programming when a third party library goes off and does something asynchronous in the background. In the meantime, the foreground thread may do other tasks, occasionally polling a notification object to see if the background job has completed, or indeed if it runs out of foreground things to do, it may simply sleep until the completion of the background job or jobs. Put simply, the foreground threads polls or waits for *permission* to continue.

**NOTE: Naive programmers think a wait condition is suitable for this purpose. This is highly incorrect due to the problem of spurious and lost wakeups inherent to wait conditions. Despite the documentation saying this, wait conditions are frequently proposed as the “correct” solution in many “expert advice” internet sites including stackflow.com among others.**

A completely safe notification object can be built from atomics and wait conditions – indeed, what is proposed below is entirely built this way. The problem is rather one of **standardisation** because of two reasons:

* 1. If third party libraries provide asynchronous functionality, they need some way of telling client code that the job is done. That implies that without the C1X threading library supplying such an object, each third party library will supply its own notification object *none of which will interoperate between libraries* or worse, use the C1X condition variable despite it being unsuitable. For example, if client code has multiple threads waiting on the third party library (a many reader, one writer configuration) but the third party library was only written to notify a single thread of job completion, this introduces unnecessary extra complexity in the client code of having to create a thread whose sole purpose is to notify all other waiting threads – and no doubt introduces an extra notification object implementation.
  2. Even the most able programmers and the brightest minds inadvertently introduce deadlocking bugs into their implementations of multithreading primitives. C1X ought to provide safe and reliable versions of the most primitive multithreading functionality on top of which programmers can construct more complex functionality if desired.

Proposed solution: Add a permit\_t object which is described below.

Cost of proposed solution: A full example implementation written using existing C1X facilities is supplied. There are however costs of validation etc. plus added complexity.

# The proposed C1X permit object

The proposed object allows client code to set whether the permit should wake a single or all waiting threads (i.e. whether permits are consumed or not consumed respectively). Client code can then supply this preinitialised permit object to third party libraries for notification usage.

The example implementation supplied in Appendix A is built entirely from C1X threading library primitives. It is (hopefully) deadlock free, suffers from **no spurious or lost wake up** issues, and has no unbounded execution times. It executes all operations in less than 150 CPU cycles (uncontended) on an Intel Core 2 CPU despite that processor having a particularly slow compare-exchange atomic instruction.

If waiters do not consume permits, the example implementation is additionally completely lock free. If waiters do consume permits, a lock is used to serialise waiting and granting to prevent permit\_grant having unbounded execution times. However it only mutually excludes permit\_grant so only one permit\_grant can execute at any given time.

# Proposed C1X permit object specification:

permit\_t, which is a complete object type that holds an identifier for a permit

*[Note: I would recommend that this ought to be defined to something (e.g. an int) even if \_\_STDC\_NO\_THREADS\_\_ is defined because it is quite common to pass a preinitialised instance of a permit object into third party library code i.e. one passes a permit\_t \* in API parameters. Lacking the type would force ABI breakage where\_\_STDC\_NO\_THREADS\_\_ does not have the same definition across libraries]*

int permit\_init(permit\_t \*permit, \_Bool waitersDontConsume, \_Bool initialstate)

**Description**

The permit\_init function creates a permit object. If it succeeds it sets the variable pointed to by permit to a value that uniquely identifies the newly created permit. If initialstate is zero, a thread that calls permit\_wait on a newly created permit object will block. If waitersDontConsume is one, a successful permit\_wait does not reset the permit’s state to blocking.

**Returns**

The permit\_init function returns thrd\_success on success, or thrd\_nomem if no memory could be allocated for the newly created object, or thrd\_error if the request could not be honored.

void permit\_destroy(permit\_t \*permit)

**Description**

The permit\_destroy function releases all resources used by the permit object pointed to by permit. The permit\_destroy function requires that no threads be blocked waiting for the permit object pointed to by permit.

**Returns**

The permit\_destroy function returns no value.

int permit\_grant(permit\_t \*permit)

**Description**

The permit\_grant function sets the state of the object pointed to by permit to granted.

If this object was initialised with waitersDontConsume set to zero, and if there is a thread blocked on the same object at that time, it shall consume the granted state and become unblocked, thus resetting the state of the object to non-granted. If there is no thread blocked on the same object at that time, the object remains with granted state until either a thread consumes this state by waiting on this object, or the object’s state is set to non-granted using permit\_revoke.

If this object was initialised with waitersDontConsume set to one, then any threads blocked by this same object shall become unblocked and no thread shall block on this same object until its state is set to non-granted using permit\_revoke.

**Returns**

The permit\_grant function returns thrd\_success on success or thrd\_error if the request could not be honored.

void permit\_revoke(permit\_t \*permit)

**Description**

The permit\_revoke function sets the state of the object pointed to by permit to non-granted.

**Returns**

The permit\_revoke function returns no value.

int permit\_wait(permit\_t \*permit, mtx\_t \*mtx)

**Description**

If this object was initialised with waitersDontConsume set to zero, the permit\_wait function returns immediately if the permit object pointed to by permit has the state of granted, atomically resetting its state to non-granted in the process.

If this object was initialised with waitersDontConsume set to one, the permit\_wait function returns immediately if the permit object pointed to by permit has the state of granted. State is not changed in this case.

If the permit object pointed to by permit has the state of non-granted, the permit\_wait function atomically unlocks the mutex pointed to by mtx and blocks until the permit object pointed to by permit gains the state of granted, wherein this state becomes consumed if this object was initialised with waitersDontConsume set to zero. When the calling thread becomes unblocked it locks the mutex pointed to by mtx before it returns. The permit\_wait function requires that the mutex pointed to by mtx be locked by the calling thread.

At no stage will any thread ever be permitted through permit\_wait without this object having gained a state of granted. Additionally, if this object was initialised with waitersDontConsume set to zero, it is explicitly guaranteed that no more than exactly one thread shall be permitted through permit\_wait for each invocation of permit\_grant.

If mtx is null on entry, the permit\_wait function will still block as detailed above, but will not sleep the thread. This is useful for high performance/low contention scenarios, or indeed if the operating system is presently in a state where it cannot sleep a thread (e.g. during bootstrap or during interrupt handling).

**Returns**

The permit\_wait function returns thrd\_success on success or thrd\_error if the request could not be honored.

int permit\_timedwait(permit\_t \*permit, mtx\_t \*mtx, const struct timespec \*ts)

**Description**

Same as permit\_wait, except that the function will return after the TIME\_UTC-based calendar time pointed to by ts.

**Returns**

The permit\_timedwait function returns thrd\_success upon success, or thrd\_timedout if the time specified in the call was reached without acquiring the requested resource, or thrd\_error if the request could not be honored.

# Appendix A: Example C1X permit object implementation

<see attached zip file. Full source will be pasted here when proposal document is submitted>