WG21/N1940 **J16/06-0010** **2006-02-14**

Nick Maclaren
University of Cambridge Computing Service,
New Museums Site, Pembroke Street,
Cambridge CB2 3QH, England.
Email: nmm1@cam.ac.uk
Tel.: +44 1223 334761
Fax: +44 1223 334679

Why POSIX Threads Are Unsuitable for C++

1.0. Introduction

This attempts to explain why POSIX threads (also called pthreads) are unsuitable to use as a basis for adding threading to C++. The problem is not with the intent, syntax or interface (though there are problems with those, too), but with the model, concepts, semantics and specification.

Please note that it does **not** make a case against standardising a simple syntactic binding of POSIX function calls, combined with stating that all areas of semantic conflict are implementation-defined or undefined. That is quite feasible, though of limited use.

1.1. Assertions

There are several major problems:

1. POSIX threads and C++ have very different and incompatible models and concepts, and forcing C++ to use POSIX's model will harm the latter badly. Attempting to change the POSIX model, even in the simplest and most reasonable ways, is a political minefield, as many people can witness.
2. The same applies to much of the semantics and most of the constraints; while this is more soluble, the result is likely to be compatible with neither serial C++ nor POSIX. Subtle semantic incompatibilities between standards already cause a great deal of grief to implementors and programmers.
3. The actual POSIX threads specification is partial, often ambiguous and sometimes internally inconsistent. Attempting to improve this has already caused major political problems, and C++ should avoid getting involved with those.
4. As the number of cores on a system gets above 2–4, it becomes necessary to improve the performance of a single task (flow of control), rather than just by separating serial tasks into their own threads, and a more fine-grained approach is needed. POSIX threads have not proved successful for this.

1.2. References etc.

Unless stated otherwise, all references are to IEEE Std 1003.1 2004 (POSIX), ISO/IEC 14882:1998 (C++), ISO/IEC 9899:1999 (C99) or ISO/IEC 1539-1:2004 (Fortran). MPI 1.1 (2.0 is essentially extensions to 1.1) can be found on `http://www.mpi.org`. OpenMP 2.0 can be found on `http://www.openmp.org`, and it is the C and C++ version being referred to here.

I shall make no attempt to describe all of the problems with POSIX threads, even in the above areas, and the document explains only a selection of some of the more fundamental ones that I know are likely to affect C++. Note that they refer to fundamental (and often pervasive) problems, and a solution that solves the particular example is not enough.

1.3. A Common Myth

It is essential to correct the frequently made claim that POSIX threads work because they are used today. As those of us in High Performance Computing (HPC) can witness, they cause immense problems as soon

as anyone stresses them hard enough to expose their flaws. Let us consider just race conditions, where the reasons are easy to analyse, and use the term "race conditions" for unsafe constructions and "race events" for ones that get exposed and cause occasional failure.

- Most uses of POSIX threads are **not** to parallelise tasks at all, but to isolate tasks (e.g. separate requests to an Internet application server) and to introduce non-blocking behaviour when using blocking interfaces. Their rate of race conditions is very low indeed.
- The density of race conditions typically varies by factors of thousands or more, between tightly coupled HPC codes and Internet application servers, leading to factors of millions in the race events (see below). This is why many HPC people have trouble with this one and few Internet application server people do.
- Many race conditions will be exposed only when two threads can run in parallel, and not when they have to be serialised on a single CPU, because most systems synchronise memory and other state when switching context. Also, the number of race conditions is often proportional to the number of CPUs, and the uselessness of a single CPU system to test multithreaded code is well known in HPC.
- Race events are proportional to the square of the number of race conditions (in obscure cases, higher powers), because a failure will occur only when two or more threads have a race condition on a single object (obviously).

So a flawed design might cause failure once a century on an Internet application server with 2 CPUs, but might show up in minutes on a tightly coupled HPC code on 32 CPUs. This is one reason why almost all HPC programs use the Message Passing Interface (MPI), a few use Fortran OpenMP, and almost none use POSIX threads in C or C++ (except for Monte-Carlo and similar embarrassingly parallel tasks). The extra tedium of sending explicit messages is more than countered by the ability to debug the program.

Some people will now say that HPC is unimportant, and POSIX threads are good enough for most uses. But what about control programs for chemical plants, nuclear reactors and other such demanding uses? They need the power of parallelism, and an unpredictable, uncontrollable failure once a decade is not acceptable.

2.0. Background and Abstract Models

This section is precisely **not** about the memory model. That is a prerequisite for any kind of shared-memory parallelism, asynchronous actions (whether I/O as in POSIX or message passing as in MPI) and even reliable signal handling (not just of asynchronous signals). Given an adequate memory model, all of those become supportable; without one, all are necessarily undefined behaviour.

It is describing structural models of threading, and why the one assumed by POSIX conflicts with C++.

2.2. Explicit versus Implicit Threading

Hoare's seminal paper "Communicating Sequential Processes" assumes that the programmer writes the code to control the program's data and control flow but, for the past 20 years, most parallel extensions to serial languages have used an implicit model where the programmer describes the linkages (and obeys constraints), and the implementation controls the program's flow.

Both MPI and POSIX threads use the explicit model, with the former using entirely separate processes and explicit, library-based message passing. POSIX threads use explicit communication, but implicit data sharing, which is part of the reason that they have inconsistencies of design. A similar approach that was once used in HPC is separate processes with some sort of shared memory (often called SHMEM, of which there are many variants); if readers are not familiar with examples, please ignore this reference.

Almost all recent parallel extensions of the C, C++ or Fortran languages (as distinct from libraries) have used an implicit model, often designed so that a clean program can be executed efficiently in serial or in parallel, depending on the implementation. This is stated explicitly as a target of OpenMP (e.g. in 1.3 Execution Model) and Fortran (e.g. in C.12.8 Parallel computation).

In the context of using multiple CPUs to increase performance, it is claimed (correctly) that it is much easier to develop parallel applications and to add parallelism to serial ones using the implicit model, but it

is also true that almost all modern, scalable, parallel applications use MPI. Experience is that debugging and tuning OpenMP programs is very hard, but experience with the array features of Fortran is much more positive.

For example, an implementation could parallelise the C++ class `valarray` in a comparable way to that Fortran implementations can and do parallelise Fortran's array features. In fact, it probably has been done.

2.3. Hierarchical versus Unstructured Threading

Over the years, there have been many ways in which threading has been added to serial models, but we can ignore most of them as unfashionable (at least in the context of C++) and hence politically unacceptable. Dataflow is one such approach. There are two main models that are relevant to C++, which will be described in simplistic terms.

The first is a hierarchy, where a parallel construct spawns a number of threads and waits for them all to finish; this can be done recursively, leading to an execution tree (with no connexion to the parse tree). A thread's lifetime is the scope of its creation, in some sense. OpenMP (see section 1.3) and virtually every other parallel extension of C, C++ or Fortran use some variation of it, because it adds implicit parallelisation cleanly into their abstract machine models. The systems MVS, VMS and CMS support this, in its explicit form.

[*Note that the threads referred to above may be virtual, and may be mapped onto a fixed number of permanently executing physical threads; in fact, they usually are.*]

The second is a fixed set of parallel, independent processes, optionally with the ability to share data. This has been most successful in its distributed memory (i.e. message passing), explicit form, and almost all scalable HPC programs use MPI. Models like Bulk Synchronous Parallel Computation (BSP, see `http://www.bsp-worldwide.org`) use it in its implicit form.

For some reason, POSIX threads adopted the second model, with explicit control of communication and implicit passing of data. This is generally adequate for the sort of application that could equally well be written to use multiple unthreaded processes with shared memory segments (including embarrassingly parallel HPC applications) but, as demonstrated below, causes major trouble for many other types of parallel codes.

2.4. Model Incompatibilities: C++ Exceptions

OpenMP is intended to be transparent (see above), but it defines C++ exceptions to be handled entirely within the current thread, to simplify integration with POSIX. Unfortunately, it does not specify how the C++ abstract machine maps to threads, and its restrictions on exception handling are so draconian that C++ programs need major changes to maintain semantics.

OpenMP 2.3 `parallel` *Construct:*

A `throw` *executed inside a parallel region must cause execution to be resumed within the dynamic extent of the same structured block, and it must be caught by the same thread that threw the exception.*

Please ignore simplifications and syntactic errors in the following, and consider parallelising code like:

```
void fred (void) {
    <some use of valarray, a for-loop or independent function calls>
}

try {
    fred();
} catch (some exception) {
    <some action>
}
```

The following is required by OpenMP to maintain the semantics:

```
void fred (void) {
    <construct an exception object A>
    #pragma omp parallel
    try {
        #pragma omp for
        <some use of valarray, a for-loop or independent function calls>
    } catch () {
        <copy the current exception object into A>
    }
    throw(A);
}

try {
    fred();
} catch (some exception) {
    <some action>
}
```

This obviously cannot be done if the function `fred` cannot copy the exception object between threads, or if the program calls a parallel class which calls user code which raises an exception to be caught outside the library call. There are doubtless other such restrictions.

Note that this specification is unnecessary, and the exception could perfectly well pass back over the parallel clause. This would not be entirely trivial to specify, as there would have to be rules for how multiple exceptions in separate threads would be merged into one, but it could be done. And, despite common belief, it is no harder to implement than allowing a call to `exit` from any thread.

2.5. Model Incompatibilities: Process Termination

Thread cancellation is problematic in any language with either pointers into thread-local data or destructors, let alone both. POSIX requires the former and C++ the latter. But POSIX is confusing enough even with normal termination and without bringing in destructors. The specification of `exit` says:

POSIX System Interfaces 3. exit():

The exit() function shall first call all functions registered by atexit(), in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. Each function is called as many times as it was registered. If, during the call to any such function, a call to the longjmp() function is made that would terminate the call to the registered function, the behavior is undefined.

The exit() function shall then flush all open streams with unwritten buffered data, close all open streams, and remove all files created by tmpfile(). Finally, control shall be terminated with the consequences described below.

There is precisely one reference to threads, it refers to `_exit`, not `exit`, and so is irrelevant to C++, so what **does** happen in a threaded program?

The above specification implies that `atexit` is called in the thread that calls `exit`; if that thread is the only one that is active, the intent is clear, though it may cause the implementor a headache. We need to think what happens to the other active threads.

Are `atexit` functions allowed or required to tidy up such threads and, if so, how? If they can or have to use `pthread_join` and `pthread_cancel`, are they allowed to use the other facilities, including `pthread_create`? Even more horribly, are those threads allowed to use threading facilities on the thread that is running the `atexit` functions? The current wording does not even forbid them from adding `atexit` functions, asynchronously, while an `atexit` function is running in another thread.

If the `atexit` functions do not clean up, "C++ 18.3 Start and termination" states that `exit` destroys non-local objects with static storage duration — but in which thread or threads are destructors called? And

do the threads continue to run while that is going on, possibly initialising more objects that need destroying?

That sounds a bad idea, but C++ entangles the calling of `atexit` functions and the calling of destructors, so the only practical place to stop the threads is before calling any `atexit` functions, which means that destructors will necessarily be called in the thread that called `exit`, and not their 'owning' thread. I do not know C++ well enough to know if that is soluble, or even a significant issue.

The threads must clearly not be destroyed until the very end, because their stacks may contain I/O buffers and the closing of `<cstdio.h>` `FILE` objects comes last. So stopping the threads must be a suspension operation — which POSIX does not have.

But we have forgotten something. Memory is only synchronised when the threads call some specific functions, so we cannot just suspend them, but have to persuade them to call a function and then suspend themselves. Even if there were a suitable signal to demand this, POSIX's approach to signal safety (and thread cancellation) is to mask signals off over critical sections, so that would not work, either.

Resolving this problem alone will not be quick, easy or clean.

2.6. Choice of Abstract Model

Questions raised by this section include:

- Should C++ specify a single threading model, or permit extensions to use different ones?
- Exactly what variation in threading models should be required, or permitted, and what properties of permitted models should it require/assume?

This document makes no attempt to answer these questions, but merely points out that the POSIX threading model conflicts very seriously with C++, and that adopting it will mean some quite major changes to C++ in harmful ways. It will also be very hard to do without introducing inconsistencies with the existing language, and may take a long time.

MPI is heavily used, but is a 'pure' library interface for message passing between separate processes, and needs nothing from C++ beyond an adequate memory model. So it may be ignored in this respect.

All of BSP, OpenMP (with fixed exception handling) and a parallelised `valarray` (etc.) library fit well with C++. The latter two are the most plausible, politically, and are compatible with each other.

And, of course, those are not the only models. Some people have added dataflow to C, though not very successfully, and I don't claim to know everything that is going on even in "my" areas.

3.0. Inconsistencies in POSIX Threads

This section is very messy, but that is because it is describing a messy situation. It attempts to explain why the C99 and POSIX standards are internally inconsistent in ways that matter very seriously to real parallel programs. The issues occur in serial ones (especially with signal handling), but are much more exposed by parallelism (including all of POSIX threads, OpenMP and MPI).

3.1. The C99 Synchronisation Model

POSIX states clearly that it is currently based on C99 (see 2.2 Application Conformance and elsewhere), so any problems with C99's memory model will affect POSIX.

Let us omit describing the C99 sequence point problem in detail, on the grounds of not restarting old and unproductive arguments, and merely note that there is no consensus on the precise synchronisation rules for serial code.

A far worse problem is the object model problem, where C99 is seriously ambiguous, and often inconsistent, when it comes to specifying the exact type of an object referred to by an lvalue. C++ has closed many of the loopholes, but by no means all. As far as parallelism is concerned, the critical aspect is knowing how much data an lvalue refers to, and therefore whether two lvalues refer to overlapping objects. I have a paper describing this, which has been widely circulated. It is temporarily on `http://www.hpcf.cam.ac.uk/export/Objects`. As a trivial example, consider:

```
static char p[50] = "ABCDE", *q = "FGHIJ";
void fred (void) {strcat(p,q);}
int joe (void) {return strcmp(p,"XYZ");}
```

Can functions `fred` and `joe` be called in parallel? If not, why not? If so, why? There is no consensus on this for either POSIX(1996)/C90, or POSIX(2001)/C99, or even whether there has been a change.

Please note that this is merely an example, and that some of the ambiguities affect constructions that are in use in most programs. I did not write that paper in the abstract, but as a result of experience with real problems in real programs on real implementations.

There are also equally bad problems elsewhere.

C99 5.1.2.3 Program execution, paragraph 5:

The least requirements on a conforming implementation are:

— At sequence points, volatile objects are stable in the sense that previous accesses are complete and subsequent accesses have not yet occurred.

— At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

— The input and output dynamics of interactive devices shall take place as specified in 7.19.3. ...

C99 6.7.3 Type qualifiers, footnote 114:

A volatile declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions.

The above makes it very clear that a C99 compiler may optimise memory accesses *ad lib.*, except for those to `volatile` objects and across an implementation-defined set of I/O calls, and the most aggressive optimisation levels in most compilers do something very like that.

Equally importantly for practical POSIX programs, C99 specifies nothing about the synchronisation of non-interactive I/O and other external actions, or signals and other exceptional conditions. In C99, only signals raised by a call to the raise function result in defined behaviour, though the proof is somewhat messy and the references are omitted. C++ is very similar, both with regard to C++ exceptions and signals.

Similar remarks can be made about the floating-point flags and modes defined in <`fenv.h`>, though with completely different (and almost unbelievable) properties. As there seem to be no practical applications that use them, or compilers that support them correctly, let us skip the details. This is merely being flagged as a potential issue, with the recommendation that C++ avoid it.

In cases anyone thinks that these issues do not arise in practice, I can witness that they do — but they are rarely even identified because so few people have the skills to track them down, and identify whether they are due to programmer error, implementor error, a defect in a standard or an incompatibility between standards.

3.2. The POSIX Synchronisation Model

POSIX is based on C99 (above), but there is no clear statement of what extensions to the C99 language (as distinct from the library) it specifies. Attempting to get this placed on a task list for POSIX 2001 failed, on the grounds that it was not within the remit of the Austin Group — and SC22WG15 has now been disbanded.

POSIX Base Definitions 4.10 Memory Synchronization:

Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. Such access is restricted using functions that synchronize thread

execution and also synchronize memory with respect to other threads. The following functions synchronize memory with respect to other threads:

fork() pthread_barrier_wait() pthread_cond_broadcast() pthread_cond_signal() pthread_cond_timedwait() pthread_cond_wait() pthread_create() pthread_join() pthread_mutex_lock() pthread_mutex_timedlock() pthread_mutex_trylock() pthread_mutex_unlock() pthread_spin_lock() pthread_spin_trylock() pthread_spin_unlock() pthread_rwlock_rdlock() pthread_rwlock_timedrdlock() pthread_rwlock_timedwrlock() pthread_rwlock_tryrdlock() pthread_rwlock_trywrlock() pthread_rwlock_unlock() pthread_rwlock_wrlock() sem_post() sem_trywait() sem_wait() wait() waitpid()

Unfortunately, that brings in the object model problem, because an obvious prerequisite to avoiding overlap is to know how big objects are. And that is what C99 (and hence POSIX) does not specify (see the Objects paper referred to above). This problem with C and threading is precisely why I wrote the document in the first place!

It also makes no reference to sequence points, `volatile` or I/O calls, and so has not a single point of contact with C99's rules. It is unclear what constraints this places on the C compiler beyond those placed by C99, if any. This confusion is one reason that getting threaded C99 code to 'work' in HPC involves dropping the optimisation level for no good reason, changing source code in irrelevant ways, adding arbitrary delays and null system calls, and similar voodoo.

As implied above, POSIX relies heavily on non-interactive I/O and signals, yet the above specifies nothing about them. Consider thread A signalling thread B, or thread A writing on a FIFO that is read by thread B. If they are not synchronised by the above, what **are** they synchronised by? Are they allowed to breach causality? This is not a joke, and really happens — some implementations will pre-execute code that they know will be needed later, as permitted by the C99 standard, and neither C99 nor POSIX require them to worry about either I/O (except to interactive devices) or signal synchronisation.

Many Internet application servers use threads to serve different requests, but use a connexionless model. When the server sets up some state, it sends a message to some external entity, that entity responds, the server receives the response, and another thread handles the response, they assume that it can use the state that was set up in the first place. But POSIX does not require that to be supported.

As above, I can witness that these problems arise in practice.

3.3. Other Model Issues with POSIX

POSIX Base Definitions 3.393 Thread:

A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, errno value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via malloc(), directly addressable storage obtained through implementation-defined functions, and automatic variables, are accessible to all threads in the same process.

It looks very reasonable, until one thinks a bit harder. What about a `va_list` object created in a thread; can it really be used in another? That is what the wording says. Is it **really** reasonable to require implementations to support that? As an implementor, it sends shivers up my spine.

And would one really want to allow C++ exception objects to be accessed from threads other than the one handling them? The above wording states that it is permitted, but it is a potential implementation nightmare, especially for systems that are not based on Unix, as I can witness from personal experience.

POSIX System Interfaces 3. longjmp():

The longjmp() function shall restore the environment saved by the most recent invocation of setjmp() in the same thread, with the corresponding jmp_buf argument. If there is no such invocation, or if the function containing the invocation of setjmp() has terminated execution in the interim, or if the invocation of setjmp() was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

Did they really mean to say that? It certainly states that a single `jmp_buf` can be used by two threads in parallel, and it will select the right data. It is very doubtful that any implementations do that, or it is what was meant.

3.4. POSIX Thread Safety

POSIX Base Definitions 3.396 Thread-Safe:

A function that may be safely invoked concurrently by multiple threads. Each function defined in the System Interfaces volume of IEEE Std 1003.1-2001 is thread-safe unless explicitly stated otherwise. Examples are any "pure" function, a function which holds a mutex locked while it is accessing static storage, or objects shared among threads.

That seems to conflict with POSIX Base Definitions 4.10 Memory Synchronization, which says that only some functions synchronise memory. The obvious interpretation is that 3.396 is referring to the hidden objects used to implement a function, and 4.10 to the objects visible to the programmer. For some functions, that makes sense, though it gets rather less rational on a closer inspection.

POSIX's list of not thread-safe functions (System Interfaces 2.9.1 Thread-Safety) is erroneous, in that it does not include `mblen, mbtowc, srand` and `tmpnam`, and bizarre, in that it does not include `abort, atexit, exit, raise, setlocale` and `signal`. It is unclear what thread-safety means for `setlocale` and `signal`, and truly baffling what it might mean for `abort` and `exit`.

The macros of the single-char I/O functions (e.g. `putc`) are clearly stated as not sequence-point safe (C99 7.1.4 footnote 156), yet are thread-safe in POSIX. Many vendors have responded by eliminating the macros, which is a performance penalty for the programs that relied on them being very efficient and optimisable. But the question remains: what on earth does thread-safe but not sequence-point safe mean?

More seriously, the penalty of requiring all I/O functions to be thread-safe is considerable. In a highly parallel program written so that only one thread does I/O to any one file (which is what all experienced parallel programmers do), there is the penalty of a lock on **every** I/O operation. Does C++ really want to impose that on the programmers who write sane code?

3.5. Specification Summary

As a long-standing expert of last resort on many areas of programming (including Fortran, C, portability, error handling and threading), I would classify the causes of the problems encountered in tested programs written by experienced programmers as follows (**very** roughly):

Ambiguity in the standard	Fortran	C	pthreads
Clear programmer error	45%	25%	15%
Clear implementor error	40%	20%	10%
Ambiguity in the standard	15%	55%	30%
Inconsistency between standards	—	—	45%

Of course, I am excluding the much larger number of obvious user errors, as they get filtered out at an earlier stage. Also, to be fair, the sample size for the last category is very small, but the results are very disturbing. That is another reason that I say that the POSIX threads specification is unsuitable to use as a basis for C++ threading.

4.0. Implementation Aspects

The implementation problems of using a different model are usually exaggerated. If C++ were to use a cleaner, simpler and more suitable model, implementing that **using POSIX threads** might well be quicker, easier and more reliable than attempting to implement a 'simple' integration of POSIX threads into C++. The reason is that it would enable the implementor to avoid many of the undefined or unreliable areas of POSIX threads. The process termination issues above are a good example. This benefit should not be underestimated.

As far as exception handling is concerned, a proper hierarchical model would be much simpler than the POSIX one, and would allow an exception to pass from a child thread back to its parent when the child thread fails to handle it and dies. This is no harder than allowing `exit` to be called from an arbitrary thread, as I can witness from actual implementation experience.

4.1. Efficiency

Up until recently, most systems made use of multiple, semi-independent CPUs by running a large pool of serial processes in parallel, but each process was definitely serial. There were a fair number of older systems which used parallelism for performance, and they generally tended to use inter-dependent CPUs with restrictive, lightweight threads which ran within the context of a process. To some extent, the two models are still with us in applications, exemplified by the Internet application servers and HPC programs, and there are some signs that the second type of system may be making a comeback.

Originally, threads were called lightweight processes, and were designed to be very cheap. However, POSIX threads took a different approach, and have added so many of the features of processes that they are very similar to separate processes with a common virtual address map. In fact, most systems either implement them as a derivative of full processes, or implement both as derivatives of a system process/thread.

In particular, POSIX more-or-less requires them to be scheduled as independent processes. Because this is precisely what is not wanted for tightly coupled parallel applications, several systems have created their own, more genuinely lightweight threads. There were, of course, several alternatives in use before about 1990, but most of them have faded from memory, though they may return.

This is not a minor point. My experience (on several systems) with attempting to tune OpenMP and POSIX threads codes is that the majority of tightly-coupled programs fail to scale because of obscure interactions with the scheduler, usually in aspects that are neither documented nor controllable (not even by the system administrator). I have seen degradations of factors of 100 and more, and believes that this may be why so many people have given up.

To keep this document within bounds, and to avoid delving into areas that are so arcane that most programmers do not know they exist, I shall not discuss scheduling models.

5.0. Conclusion

The conclusion was stated in the first section — POSIX threads are not a suitable basis for adding threading to C++, either conceptually or as a specification. Designing a form of threading that fits more naturally into C++ would lead to a more reliable result, would almost certainly be quicker, and might well be easier to implement using POSIX threads than integrating POSIX threads as they stand.

If there is a need to provide a POSIX threads binding (as part of a full POSIX binding or otherwise), my suggestion is to define the syntactic binding, to require implementation-definition of the major problem areas (like process termination and exception handling), and to say that all other areas of conflict are undefined behaviour. That is what will be the case in practice, anyway, as it is at present when POSIX is called from C++.