Document: WG 14 N1341
Date: 2008-08-12

Austin Group Comments on N1287

This is simply a collection of comments received on the Austin Group mail list in response to a request for review and comment on N1287, threads API. It does not represent any official Austin Group position at this time.

From Ted Baker:

Here we go again, with more proliferation of similar-but-not-identical standards... I skimmed through the document at the cited URL. It is so close to POSIX threads that I wondered what is the justification for introducing yet another set of names. It seems that the semantics are more loosely specified than POSIX threads, so if their objective is to be able to support a single API as a layer over other thread implementations then this might be of some value.

However, I anticipate that some users will not be satisfied with such a feature-poor API, and will start demanding more, inevitably recapitulating the history of POSIX threads. For example, I guess it is inevitable that we will soon see a "real time" version of this and the proposed C++ interfaces. --Ted

From David Butenhof:

I have to agree with Ted. Perhaps the weakest part of this pseudo-generic wrapper package is that it lacks even an extension mechanism to change any semantic attributes of threads before starting them; something absolutely critical if any future foray into realtime scheduling support might come to be in the cards. Or, for that matter, any realistic use of the "portable" C facility on real systems for real programs, where extensions to access base thread implementation capabilities are going to be needed (e.g., realtime, stack allocation control, etc.)

Some APIs like UI threads allow creating suspended threads, setting attributes via additional APIs, and then starting it; others, like pthreads, tried a more structured approach with attributes objects on the create operation (which also has the advantage of being easily layered on a base implementation that works the other way, while the converse is possible but much messier). This has neither.

Of course the "xtime" thing and thrd_sleep() is just POSIX timespec and nanosleep(). Adding a C language facility equivalent to 'struct timespec' will, inevitably, end up with a similar but semantically unrelated data type that needs to be converted and interpreted in any real application, causing pain for innumer-

able developers for many years to come. While I see the motivation that drives messes like this, I don't like it.

C can't grow to encompass all of the differences between various operating systems to enable 100% portable programs, no matter what. Frankly I'd much prefer that C focus on a good, strong, portable memory model that would enable fully portable use and creative/useful "abuse" (lock-free, wait-free) of the various threading APIs already widely deployed and understood. Especially when the "standard" C thread interface is so bare-bones and non-extensible that it essentially prevents most developers from using it anyway.

The memory model is critical. Having a C runtime threading API isn't, unless it's going to allow full portable use of at least most common posix and win32 mechanisms and attributes; so that you can actually write a useful portable (or at least "mostly portable") threaded program. This API, as it stands now, doesn't come close to fitting that bill.

From Hallvard B Furuseth:

(In response to Ted Baker):

While I don't know why they are doing it this way, I can see one advantage: The C language and memory model will finally acknowledge the existence of threads. It could do that without providing an API, and maybe suggest these primitives as a theoretical way to implement threading. However I'd suspicious of such a purely theoretical model. If it doesn't get some real-life exposure in its own right, design bugs and other problems don't get detected as easily.

OTOH - I found a strong criticism of Posix-like threading in: "Why POSIX Threads Are Unsuitable for C++" by Nick Mclaren: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1940.pdf I didn't understand all of it, and I don't know if the current proposal is loose enough to accommodate his objections or if he lost the debate. Statements like "almost all modern, scalable, parallel applications use MPI" (as in OpenMP rather than pthreads) looked a bit damning...

(In response to Dave Butenhof):

Another extension mechanism I missed is that pthread functions return errno values (which the implementation can invent), while this one only returns a few predefined error values. If none of the three defined error values are suitable, why not let the functions return 'any other (positive?) value' instead of 'thrd_error'? That'd also allow - but not require - an implementation to return errno values, and to define the thrd_* error values to 0, ENOMEM, ETIMEDOUT and EBUSY. Incidentally, it looks to me like these functions as written must not return EINTR — they must instead loop.

I guess the functions could set errno when returning thrd_error, but it seems messy to both return an error value beyond success/failure and set errno. One should be enough.

[re xtime]

Couldn't xtime be defined so that POSIX can state that xtime must be a typedef for struct timespec?

The current xtime spec is wrong for C: It says sec and nsec are ints. But if int is 16-bit, it can hold neither nanoseconds nor the current time in seconds. It's not necessarily a Dinkum bug - maybe their implementation does not attempt to support 16-bit int.

Other notes about n1287:

The type names 'once_flag' and 'xtime' lack a '_t' suffix. I can well imagine at least the former could conflict with variable names. For that matter, depending on how widely this API is expected to be used, I wonder if the functions and types should all get a 'thrd_' prefix, to reduce namespace pollution in programs that do use them.

Why do thrd_abort() and thrd_sleep() have thrd_ prefixes? What is thread-related about those functions?

There is no equivalent of PTHREAD_<COND/MUTEX>_INITIALIZER. Maybe they could be made optional - so programs that want them can do

```
#ifdef MTX_INIT/CND_INIT?
```

Then implementations where cnd_init/mtx_init use resources, can choose to not #define these flags - and thus ensure that the first call to mtx_lock() on the mutex will not fail due to resource limits in the implicit "if (mutex == MTX_INIT) { mtx_init() for real }".

> Memory Visibility
> "There are three times at which changes made to memory by one thread
> are guaranteed to be visible in another thread:

> [...thread creation and mutex use...]"

By itself this is misleading, since other parts of the document goes on to say:

> "The condition variable functions use a mutex internally; when a
> thread returns from a wait function any changes made to memory by
> threads that called a wait function or a notify function before the
> return will be visible to the caller."
> (...)
> "The function call_once uses a mutex internally; when it returns any
> changes made to memory by the once function will be visible to the
> caller."

I see no reason why the document should state how operations works internally. It'd be cleaner to list the actual operations that affect memory visibility up front.

> Once Functions

This section (as well as the POSIX spec) should specify that if several threads call call_once/pthread_once in parallel with the same once_flag, the ones not calling the once function will wait - so that the once function call will have been completed when the threads proceed.

> mtx_init
> (...)
> The function creates a mutex object with properties indicated by type,
> which must have one of the six values

This and the function specs would be more readable if described in terms of a bitmask mtx_something (mtx_plain|mtx_timed|m-tx_try) | mtx_recursive.

For that matter, if mtx_timed is defined as (mtx_try | some-thing) != 0, then mtx_trylock would only need to mention that it needs the mtx_try flag - it need not mention mtx_timed.

>   * mtx_timed -- for a non-recursive mutex that supports time-out

That should be "timeout as well as test-and-return", according to the mtx_trylock() spec. Also fix the 'mtx_timed' header.

Incidentally, I suggest to spell test-and-return with hyphens. That "and" can be a little confusing in the middle of a sen-tence, if printed on a device which does not preserve underlin-ing of "test and return".

> #define TSS_DTOR_ITERATIONS <integer constant expression>

Maybe this should be a allowed to expand to a non-constant ex-pression, so the implementation is free to make it a user-settable resource limit.

Document: WG 14 N1341
Date: 2008-08-12

From Hans Boehm:

I don't have any strong opinions on the pthreads vs. Dinkumware issue. However, there are some details in this proposal, that I think should be adjusted. In particular, the memory visibility discussion needs to be consistent with whatever the final memory model ends up being. It's important how this interacts with atomics, which I hope will also be in the standard. There are other issues, such as with call_once.

I think it's crucial that the standard clearly guarantee sequential consistency for data-race-free programs (low-level atomics excepted), since I think that's the only programming rule that I think we can actually explain to most programmers. And it's the one the one that's now supported by both Java and the C++ WP.

This model (sequential-consistency for data-race-free programs) effectively requires that mtx_trylock() and mutex_timedlock() be allowed to spuriously fail to acquire the lock. Otherwise the client can tell whether mtx_lock() includes a memory fence on both sides, which makes it significantly more expensive to implement on many architectures. I think the pthreads specification should eventually be correspondingly relaxed. (This is not intended to encourage implementation changes, and hence does not have to break code, even that small amount of really obscure code that relies on this. But it does cause many existing implementations to actually meet the letter of their specification.)

This proposal also raises the question of whether mutexes should be statically initializable. Requiring this may pose some implementation challenges on some platforms, but I believe statically initialized mutexes are very useful to the client. The alternative is often an (easily forgotten and rather messy) call_once call before acquiring a mutex.

Hans

From yodaiken@fsmlabs.com

How do you do:

```
thread A: while(1){wait for event; process data; sem_post;}
thread B1 ... BN: while(1) { sem_get; collect data; ... }
```

?

Gotta implement semaphores with conditions?