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

Asynchronous Exceptions for Threads

1.0 The Problem

Thread cancellation is a horrible issue, but the problem is that many users want it and will use facilities like `pthread_cancel()` and `pthread_kill()` if C++ does not provide anything. We discussed this in Redmond, and here is what I suggest should be provided, based on mainframe designs (including but not limited to ones that I implemented) – note that the syntax is all wrong, and this is describing solely the semantics.

1.1 Primitives with Defined Behaviour

These can be implemented reliably with defined semantics; the first is the one we discussed in Redmond. The second may seem a bit pointless, but is a hook for implementations that want to provide such a facility; it is trivial to implement by always saying "no", and no portable code should rely on it doing anything.

- `typedef enum {failure, normal, flagged, terminating;} state_t;`
- `state_t thread_interrupt_delayed (thread_t thread);`

Unless the thread is already terminating, this sets a flag for the target thread to request an exception. If is is already terminating, it leaves the state unchanged. In both cases, it returns what the state was immediately before the call.

The thread checks the flag in certain library calls; if it is set, it clears it and throws the `asynchronous_termination` exception. The set of checks should include both entry to and exit from a defined set of thread library synchronisation calls. An implementation may define other places where it is checked, and the wording should allow anything from no other checking up to checking between every statement.

- `state_t thread_interrupt_immediate (thread_t thread);`

This behaves like `thread_interrupt_delayed()` but requests the implementation to behave as if a `throw asynchronous_termination;` statement had been inserted between two statements in the target thread's code "effective immediately". If an implementation cannot do this and honour C++ semantics (as will be the case for most optimised code), it shall do nothing and return `failure`.

1.2 Primitives with Undefined Behaviour

Unfortunately, there **are** requirements for more draconian interruption mechanisms, as shown by the existence of `pthread_cancel()` with `PTHREAD_CANCEL_ASYNCHRONOUS` and `pthread_kill()`. Either of `pthread_cancel()` with `PTHREAD_CANCEL_DEFERRED` or `pthread_kill()` combined with signal masking can be used fairly safely and portably, and provide essentially the same functionality as `thread_interrupt_delayed()` above, but I am referring to the other cases.

Because of the C language specifications and modern implementation strategies, an unexpectedly cancelled or interrupted thread can do almost anything. The only case where this is not true is for non-portable codes, where the program relies on implementation-dependent behaviour. Very few implementations guarantee such behaviour, so even such codes are very often reliable only for a particular release of the implementation.

The specification of the following functions needs to make it clear that calling these functions will result in undefined behaviour, though an implementation **may** be kind and define what will happen, and that the answer "no" should always be expected. The correct use of these primitives is cleaning up after something has already gone wrong, when the only realistic alternatives also cause undefined behaviour.

Please note that I have missed functions like these, as an implementor, and provided them as an extension, because they are important when cleaning up after a fatal error. There is no good reason not to expose them to the user, as their specification is portable, even though their behaviour is largely unpredictable.

- `state_t thread_interrupt_abort (thread_t thread);`

This behaves as if the currently executing statement in the target thread were forced to call the `abort()` function immediately, but where the effect of the `abort()` would apply only to the target thread. If an implementation is unable to cause this, for any reason, it should return `failure`.

- `bool thread_interrupt_abort_all ();`

This calls the `thread_interrupt_abort()` function on all active threads, including disconnected ones, and returns `true` only if it succeeded on all of them.