WG21/N1963 **J16/06-0033** **2006-02-23**

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

Generic Support for Threading Models

1.0. Introduction

This is orthogonal to the memory model question, but arose from a discussion on my POSIX document. It is a proposal for how a wide range of threading models (obviously not all) could be supported by a single mechanism within C++. As before, please ignore mistakes in the wording, as it is the concepts that are important.

To summarise, there are currently two main threading models in use in a C++ context, which I shall call the flat model and the hierarchical model. POSIX threading is close to the first and OpenMP is close to the second. There are also other models that are out of fashion, which could well come back — and, of course, there are doubtless models that I don't know about or are yet to be invented.

As in my other documents, the base documents are IEEE Std 1003.1 2004 (POSIX), ISO/IEC 14882:1998 (C++) and `http://www.openmp.org` (OpenMP). Other relevant references are `http://www.mpi.org` (MPI), `http://www.bsp-worldwide.org` (BSP), plus numerous ones for coroutines and dataflow.

1.1. The Abstract Models

The flat model is essentially the one described in Hoare's "Communicating Sequential Processes", and is the basis for both distributed memory parallelisation (e.g. MPI) and many forms of shared-memory parallelisation (e.g. POSIX threads). The essence is that there are a number of separate, equivalent processes, which run independently and communicate with each other by explicit actions. POSIX threads are not completely flat, in that they can be spawned hierarchically, but thereafter run independently — this leads to a lot of problems with scoping.

The hierarchical model is essentially the one used by OpenMP. The essence is that a basic action spawns a number of threads and suspends itself; its children run to completion and the results are collated; the parent thread then continues. This process is recursive, leading to a tree of threads. In general, the threads do not communicate with each other directly, though OpenMP does include a few such facilities. Virtually all automatic parallelisation uses the hierarchical model, because it is the one that makes most sense when increasing the performance of an initially serial program by adding parallelism.

Coroutines do not run independently but communicate explicitly, and BSP processes run semi-independently but communicate implicitly. Dataflow models divide the problem into independent units, which run according to the data dependencies. Another fairly common model is to bind threads to objects, so that a class instance has some associated data and one or more threads that last only as long as the object does; this is particularly useful for advanced I/O. All these (and many others) are not described here, but it seems desirable that C++ extensions (and perhaps even C++) should be able to support the main ones.

1.2. Specific Problems with the Flat Model

It is often claimed that the flat model is the most flexible, but there is a sense in which most of the models are equivalent. However, the flat model is definitely the flavour of the decade, so it is worth considering as a basis. The problems with integrating it into C++ show up most clearly in the termination question.

When one thread calls `exit` or `abort`, what should happen to the other threads?

There are essentially three choices:

- They complete normally, and the program stops only when the last thread stops. There is then a sub-question of which of the return statuses is used.
- They are sent an asynchronous signal, which begs the question of whether and how C++ should support such things.
- The program's behaviour is undefined.

POSIX is completely silent on this one.

There is also the issue of what to do with threads that terminate with exceptions.

There are some very nasty issues to do with scheduling (coroutines are an extreme example), where POSIX does not help, either. The problems with scheduling are arcane to a degree, and are not described here, but the proposal attempts to bypass them by proving some flexible mechanism.

2.0. A Possible Generic Solution

The approach taken in this document is to define some mechanism that is sufficiently flexible to allow a programmer to implement most of the main models, including coroutines and dataflow. It is based around the concept of the initial thread being a master thread, which is provided with enough power to control the other threads.

The solution is described in terms of POSIX threads, because that seems to be the viewpoint that most people have taken. That is a bit sad, because it is one of the least "conceptually parallel" of all the models. No assumption is made about which of the POSIX facilities should be provided, except where stated.

2.1. Generally Available Facilities

Condition variables, mutexes, read/write locks, spin locks and semaphores can be set up in either the master thread or slave threads, but can be used only in the latter. If used in the former, they fail in some suitable way. Exactly why POSIX wants all of those is left as an exercise for the reader.

Thread attributes, scheduling parameters etc. are similar, but are generally passed back as messages by a slave thread, requesting action by the master thread.

2.2. Master Thread Functions

These include all of the functions by which threads are created and controlled, including cancellation, concurrency, scheduling and barriers. If POSIX-like function calls are provided for use by slave threads, they map into messages passed back to the master thread. They fail in some suitable way if called from a slave thread.

`void threadset_create (int` *count*`);`

This may be called only if there is no active threadset, and creates one. *count* is merely a limit on the number of threads (like `PTHREAD_THREADS_MAX`, and all threads start in a destroyed state. All other functions may be called only when there is an active threadset.

`void threadset_destroy (void);`

This destroys the active threadset, and fails if any of the threads are not yet destroyed. It is undefined behaviour if the master thread terminates with an active threadset.

`void threadset_halt (bitset` *mask*`);`

This halts the threads specified in the mask at the next synchronisation point. Synchronisation points are implementation-defined, but must include any specified by C++; I refer to the latter in my document on the library and non-memory actions.

`void threadset_continue (bitset` *mask*`);`

This restarts the halted threads specified in the mask.

`bitset threadset_select (bitset` *mask*`, double` *delay*`);`

This waits until at least one of the threads specified in the mask raises an event (see below), or the number of seconds of delay have passed, and returns which have done so.

`void threadset_throw (bitset` *mask*`);`

This throws a `THREAD_FAIL` exception in each of the halted threads specified in the mask.

`void threadset_terminate (bitset` *mask*`);`

This forces a call to `terminate` in each of the halted threads specified in the mask.

`void threadset_abort (bitset` *mask*`);`

This attempts to kill each of the threads specified in the mask, dead. The exact behaviour is implementation-defined, and any memory being updated by any threads that are running is left in an undefined state.

`void thread_create (int` *index*`);`

This starts the specified thread, which must be in a destroyed state.

`thread_state thread_condition (int` *index*`);`

This returns data on the state of the specified thread; the possible states and other available data are:

- `thread_destroyed`
- `thread_running`
- `thread_halted`
- `thread_exited, int` *code*
- `thread_terminated`
- `thread_aborted`
- `thread_exception, void *` *exception_ptr*
- `thread_message, void *` *message*

A thread is halted in states `thread_halted` and `thread_message`.

`void thread_destroy (int` *index*`);`

This destroys the specified thread, which must not be in state `thread_running`, `thread_halted` or `thread_message`.

2.2. Slave Thread Functions

All of these are callable only from a slave thread, and fail in some suitable way otherwise.

`void exit (int` *code*`);`

`void terminate (void);`

`void abort (void);`

These stop and inactivate the thread (alone), as if it were a program.

`void thread_message (void *` *message*`);`

This passes a message back to the master thread, and halts the calling thread. There is no mechanism for returning a value, but that can be done by storing something into an address passed back.

2.3. Excluded POSIX Threads Functionality

Cleanup handlers and thread-specific keys are already in C++. `pthread_atfork` and `pthread_sigmask` are outside C++'s model. `pthread_exit` is provided by `exit` in a thread, and POSIX `exit` is provided by passing a message back to the master thread.

2.4. POSIX Emulation

Essentially, all that the main thread needs to do is to start a threadset, and enter an event loop (much as in GUIs), dealing with the various actions requested by one thread to apply to another.

There would need to be a large number of messages defined, corresponding to most of the POSIX facilities I have ignored, but the approaches are clearly equivalent (i.e. client-server versus subroutine).

The real advantage of the approach proposed here is that it gives the programmer complete control, and allows the implementation of almost any scheduling model.