Threads Package

Sean Reilly - February 27, 2015

# Thread Pools

Threads allow the developer to spread computation tasks across processors on multi-core computers. All threads in a process share access to the processes memory. Instead of creating a new thread for each task, tasks can also be run in the thread pool. The thread pool associates a queue of tasks to be completed with a set of threads that execute those tasks. This process is illustrated in Figure 1. The application creates tasks and adds them to the queue. The execution threads are creates as part of the pool, and tasks are automatically executed by the next available thread. USML uses thread pools to control the relationship between the number of hardware cores and the maximum number of simultaneous executing tasks.

completed tasks

execution threads

task queue

Application

create and add

Figure – Thread pool concept

This design uses Boost's Asynchronous I/O (ASIO) library to implement C++ thread pools. Our internet research suggests that many of the people using the Boost Threads library also use the ASIO Recipe for a thread pool (<http://think-async.com/Asio/Recipes>). ASIO handles all of the work of scheduling and concurrency management. Our implementation differs from the ASIO Recipe in several aspects:

* Tasks are passed to the thread pool using a shared\_ptr. This allows the creator of the task to continue to communicate with the task without fear that the thread pool has deleted it. The task is automatically deleted when both the thread\_pool and the creator of the task stop referencing it.
* The ASIO Recipe allows any function or functor to be passed into the pool. Our implementation implements a base class for thread tasks so that some additional behaviors can be standardized.
* Standard behaviors for identifying and canceling tasks are included in the task base class.



Figure – Thread pools classes

The classes for this implementation are illustrated in Figure 2. The typical use is:

* The developer defines a new class that inherits from thread\_task.
* The sub-class overloads the run() method to execute the new class, and then adds any other methods needed by the task.
* The developer dynamically creates a thread\_task::reference for a new instance of the sub-class, and then calls any configure methods needed on the new instance.
* The developer passes the thread\_task::reference to thread\_pool::run().
* The the thread\_task::reference is used to invoke the abort() method if a need arrises to pre-maturely abort this task. The subclass code monitors the \_abort flag to detect when abort() has been invoked.

Developers can create multiple thread\_pool object directly, or use the thread\_controller singleton to share a common thread\_pool across the entire process. The thread\_controller singleton automatically creates one thread for each hardware core.

# Read/Write Locks

The classes illustrated in Figure 4 provide a simple mechanism for implementing a "multiple read/single write" lock internal to class methods. The goal is to create a scheme that blocks during writing, but allows concurrent access by many threads when just reading.

The example below shows these types being used in concert to control multi-threaded access to set/get accessors on a simple class. During writing, the class uses the write\_lock\_guard to gain exclusive access to the mutex that locks this whole object. During reading, read\_lock\_guard establishes a shared lock. Multiple read\_lock\_guard locks can be active without blocking each other. But, the write\_lock\_guard block with all other read\_lock\_guard and write\_lock\_guard locks. Both types of guard unlock when the guard object goes out of scope. Additional read\_write\_lock objects can be added to make the locking more granular.

class Thing {

private:

read\_write\_lock \_mutex ;

int \_value ;

public:

// constructor

Thing( int v ) {

\_value = v ;

}

// Retrieve current value with locking.

int value() const {

read\_lock\_guard(\_mutex) ;

return \_value ;

}

// Define new value with locking.

void value( int v ) {

write\_lock\_guard(\_mutex) ;

\_value = v ;

}

};

Figure – Read/write lock example

Defining these as typedefs allow us to easily migrate from using std:: threads and locks when shared locks become available in the C++14 standard.



Figure – Read/write lock typedefs