

Threads in NMM

Motama GmbH, Saarbruecken, Germany
(<http://www.motama.com>)

April 2010

Copyright (C) 2005-2010
Motama GmbH, Saarbruecken, Germany
<http://www.motama.com>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being all sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found in the file COPYING.FDL.

THE DOCUMENT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE DOCUMENT BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENT OR THE USE OR OTHER DEALINGS IN THE DOCUMENT.

1. Class Thread

This Class Thread is a wrapper for the correspond platform dependent threading library. The start-method creates a new thread of control that executes concurrently with the calling thread. The methods and the arguments are similar to the corresponding low-level methods. The following example shows how to create a new Thread.

```
void* thread_method(void* argument) {  
    ....  
    return 0;  
}
```

```

class Foo::foo() {
    Thread new_thread;
    new_thread.start(thread_method,0); //runs the method thread_method in a new thread,
                                     //with no arguments.
    ...

    new_thread.join()                // blocks until the thread terminates
}

```

2. Class ThreadMutex

This class is a wrapper for mutex variables. All resources are allocated and deallocated in the constructor/destructor. It provides the same methods like a typical mutex, i.e. lock(), unlock(), trylock(). The ThreadMutex implements two kinds of Mutexes, 'Fast' (default value) and 'Recursive'. The kind of a mutex determines whether it can be locked again by a thread that already owns it or not. Here is a little example, how to use this class.

```

class Foo {
    ....
private:
    ThreadMutex mutex; //the new Mutex.
}

int Foo::foo() {
    mutex.lock();
    ....
    mutex.unlock();
}

```

If the same thread should lock the ThreadMutex again, you must use a ThreadMutex with kind 'Recursive' or you cause a deadlock.

```

class Foo{
    ....
private:
    ThreadMutex* mutex;
}

Foo::Foo() {
    mutex = new ThreadMutex(ThreadMutex::Recursive); //Now the mutex can be locked again.
}

int Foo::foo() {
    mutex->lock(); //locks the mutex
    mutex->lock(); //locks it again.
    ....
}

```

```

mutex->unlock(); //unlocks the second lock
mutex->unlock(); //unlocks the first lock
}

```

3. Class ThreadCondition

A ThreadCondition is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. The basic operations on conditions are: notify the condition and wait for the condition. A ThreadCondition must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it. The following example shows how to use a ThreadCondition.

```

class Foo {
private:
    ThreadCondition* condition;
    ThreadMutex mutex;
};

Foo::Foo() {
    condition = new ThreadCondition(mutex);
}

int Foo::foo_1() {
    if ( !data_available)    // if no data available
        condition->wait();  //wait for a condition.
}

int Foo::foo_2() {
    condition->notify();      //if this method is called, new data are available and
                             //waiting conditions are notified.
}

```

4. Class MutexGuard

If you write threadsafe code you must ensure that locked mutex-variables are also unlocked. This class can be used to ensure that a locked mutex is automatic unlocked at the end of a method. The constructor of this class locks the mutex and the destructor unlocks it. So you can create a MutexGuard, when you enter

a critical section and the `MutexGuard` unlocks the `Mutex` if it leaves the scope. A little example shows how to use this class.

```
int Foo::foo() {
    MutexGuard m(&mutex); //mutex is the internal ThreadMutex to ensure mutual exclusion.
                          //The constructor locks the mutex.

    .....
    .....

    return 0;
} //Leave the scope, which unlocks the mutex
```