AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture & Design of Embedded Real-Time Systems (TI-AREM)

POSA2:
Scoped Locking C++ Idiom

Version: 17-2-2015

Abstract

The Scoped Locking C++ idiom ensures that **a lock is acquired** when control enters a scope and **released automatically** when control leaves the scope, regardless of the return path form the scope

Context

- A concurrent application containing shared resources that are manipulated by multiple threads concurrently

Problem

- Every public monitor operation accessed by clients should start with acquiring the lock (**enter** monitor or **acquire**) and end with releasing the lock (**exit** monitor or **release**)
- The programmer could forget it in some return path
- The solution is not exception-safe

Example: Web Server Hit Counter

```
class Hit_Counter {
public:
                         // Increment the hit count for a URL <path> name
   bool increment(const string &path) {
      lock_.acquire();
      Table_entry *entry = lookup_or_create(path);
      if (entry == 0) {     // something's gone wrong
         lock_.release();
         return false;
      }
      else {       // increment hit count for <path> name
            entry->increment_hit_count();
            lock_.release()
            return true;
      }
   }
private:
   Table_entry *lookup_or_create(const string &path);
   Thread_Mutex lock_;
};
```

Solution: Scoped Locking C++ Idiom

- Define a **guard class** whose **constructor** acquires a lock when control enters a scope and whose **destructor** automatically releases the lock when control leaves the scope

```
void MonitorClassX::operationX()
{
   Guard myGuard(lock_);        // lock_ pointer to a lock object
    // …  Critical section code
    // …
    return;
}   // destructor called automatically
```

Implementation

```
class Thread_Mutex_Guard {
public:
   Thread_Mutex_Guard( Thread_Mutex &lock) :
         lock_(&lock), owner_(false) {
         lock_->acquire();
         owner_ = true;
   }
   ~ Thread_Mutex_Guard() {
         if (owner_)
            lock_->release();
   }
private:
   Thread_Mutex *lock_;         // Thread_Mutex wrapper facade
   bool owner_;
   // disallow copying and assignment
   Thread_Mutex_Guard(const Thread_Mutex_Guard &);
   void operator= (const Thread_Mutex_Guard &);
};
```

Example: Hit Counter with Guard object

```
class Hit_Counter {
public:
                        // Increment the hit count for a URL <path> name
   bool increment(const string &path)
   {
      Thread_Mutex_Guard guard(lock_);
      Table_entry *entry = lookup_or_create(path);
      if (entry == 0) {    // something's gone wrong
           return false;
      }
      else {      // increment hit count for <path> name
           entry->increment_hit_count();
           return true;
      }
   }
private:
   Table_entry *lookup_or_create(const string &path);
   Thread_Mutex lock_;
};
```

Variants: Explicit Assessors

- Some situations requires a possibility to release the lock explicitly

Example problem where the lock will be released twice

```
{
    Thread_Mutex_Guard guard(lock);
    // do some work
    if (a certain condition holds)
            lock->release();
    // do some more work
    // leave the scope, which releases the lock again
}
```

Implementation with explicit accessors

```
class Thread_Mutex_Guard {
public:
    Thread_Mutex_Guard( Thread_Mutex &lock) :
            lock_(&lock), owner_(false) { acquire(); }
    ~ Thread_Mutex_Guard() {  release(); }
    void release() {
            if (owner_) { owner_= false; lock_->release(); }
    }
protected:
    void acquire() {
            lock_->acquire();
            owner_ = true;
    }
private:
    Thread_Mutex *lock_;
    bool owner_;
    // disallow copying and assignment …..
};
```

Consequences

- Benefits:
  - Increased robustness by eliminating common programming errors
- Liabilities:
  - Potential for deadlock when used recursively
  - Limitations with language-specific semantics
  - Excessive compiler warnings

Known Uses

- Booch Components
  - C++ class libraries
- ACE
  - Adaptive Communication Environment has an ACE_GUARD implementation
- Thread.h++
  - The Rogue Wave Threads.h++ library has a set of guard classes
- Java
  - has a programming feature called a synchronized block (compiler generates monitorenter, monitorexit and an exception handler)

Relation to other POSA2 Patterns