Computer Science • Vol. 8 Special Edition • 2007

Michał Korzycki*

A COMPILE-TIME DEADLOCK DETECTION PATTERN

The paper presents the application of the trait technique in generic programming for compile-time deadlock detection and prevention in multithreaded applications.

Keywords: *generic programming, multithreading, design patterns*

WZORZEC CZASU KOMPILACJI DLA DETEKCJI ZAKLESZCZEŃ

W artykule zaprezentowano przykład zastosowania techniki trejtów *z C++ do wykrywania potencjalnych zakleszczeń w programie wielowątkowym.*

Słowa kluczowe: *programowanie uogólnione, wielowątkowść, wzorce projektowe*

1. Introduction

Writing multithreaded programs became an unavoidable neccessity. The reasons for that are ranging from performance issues to program structure simplification. Not to mention the simple fact that multi-CPU and multicore-CPUs hardware is much more available for common usage nowadays than it was a decade ago.

Multithreading introduces a wide new range of technical issues that we must deal with. Typically those problems are of an order of magnitude harder to track and eliminate than those typically found in classical sequential programs. It shows clearly that our focus should be oriented on finding solutions that could prevent those problems, rather than trying to solve them after that they have occured.

I would like to demonstrate in this article the possibility of using generic programming techniques to help building multithreaded code. The non-intrusive pattern presented permits to restructure some algorithms and detect at compile time potential deadlock points. This pattern bases on one of the basic techniques of C++ metaprogramming: type traits.

The presented pattern does not rely on any specific thread or monitor model. All thread implementation specific issues are assumed to be hidden in the `Mutex`[1] class, that provides the interface for a classical readers-writer lock. It and can be easily replaced to suit any required environment.

* Institute of Computer Sciences, AGH University of Science and Technology, Kraków, Poland, `korzycki@agh.edu.pl`

[1] For the sake of brevity I present here only the key code fragments. The full source code is available under `http://winnie.ics.agh.edu.pl/deadlockdetection/src.tgz`

Listing 1. The `Mutex` interface

```
class Mutex {
public:
    void readLock();
    void writeLock();
    void releaseReadLock();
    void releaseWriteLock();
};
```

2. The problem

One of the problems that we must deal with in multithreaded applications are deadlocks. The algorithms that could suffer from such deficiency are those that deal with nested critical sections. A deadlock can occur when two threads of execution are separately inside two critical sections protected by their corresponding locks, and try simultanously to acquire locks already held by the opposite thread. This nesting of critical sections leads to the situation when each of the threads waits forever for the other one to release its first acquired lock. The problem could becomes even more complicated if many different threads are involved, causing an apperance of a cycle in the monitor nesting sequence.

One of the solutions usually found is to have a try-and-see approach to locking. When a certain lock attempt takes too much time – we simply abandon it. That often is followed by throwing an exception, causing the exception handler to rollback all acquired locks - possibly releasing the lock cause. That still is a runtime solution. I will present now an attempt to find a solution for deadlock prevention at compile-time.

3. The solution

We can represent locks as nodes in a directed graph, and the possibility of locking the monitor, call it A, after having acquired the lock for a different one, call it B – as an edge going from A to B. Potential deadlocks are corresponding to cycles in such a graph. Thus, the mentioned graph should be an acyclic directed one. Such graphs can be represented by means of a relation. Two nodes, A and B, will be in relation if there is a path in the graph leading from A to B. This relation is of a partial ordering type, so we can use such terms as “less” and “smaller than” when comparing two locks in relation. A formal proof of this property is unfortunately beyond the scope of this article.

If we track all the locks that we have acquired in the current thread, and make sure that each time that we lock a new one it is always “smaller” than all the others that we have already locked, we will never encounter a deadlock in my program. This can be done quite straightforward on run-time. We can create a comparing function that checks the lock to be acquired. If the check procedure fails, we can abandon the lock attempt of the monitor avoiding the dreaded deadlock.

There is, however, a better way than detecting and solving the problem at runtime. What if we could use an architecture that simply disallows to create deadlock-prone code? Here generic programming comes to help. We can represent the mentioned above relation in C++ terms: the relation between types "being a parent of" is of a partial ordering type. If the sequentially locked monitors are of different types, and their ordering breaks the inheritance relation of some related helping classes– that–in turn–will cause the compilation to fail. We will use for that behaviour for defining traits for specific monitor types. A trait in generic programming is an external, non-intrusive class containing information describing a specific type. Traits are implemented in C++ usually using template specializations.

Compile-time deals with types and not instances. That restricts at first sight the described pattern to the cases where sequentially locked monitors are of different types. We will see that by taking some additional assumptions on the interfaces of the monitors we are able to lift this restriction, expanding this pattern also to monitors of the same type.

4. Implementation

We based the implementation on several elements. First, a smart pointer template `LockProxy` with a member field being a trait. We use that template to reference all the resources that we want to lock and protect from deadlocks.

We provide this proxy in two flavors. One gives a read-only access to the protected object, the other one gives full access to it. Read-only access is easily expressed in C++ terms through const-correctness, declaring the pointer to the accessed resource as a pointer to a const. That way only methods declared as const can be called on such an object. That allows me in turn to take opportunity of the `Mutex` device that enables many readers to access the object, but allows only one writer. Such discernation helps to reduce contention on monitors that are only "read", and helps to avoid dirty reads (reading a value while it is being written), which could lead to race conditions.

Listing 2. Differences between the two types of the `LockProxy` interface

```
template <class T> class WriteLockProxy : public LockProxyTmpl<T> {
    T * resource;
public:
    T * operator-> () {
        ...
        return resource;
    }
    ...
};

template <class T> class ReadLockProxy : public LockProxyTmpl<T> {
    const T * resource;
public:
```

```
    const T * operator-> () {
        ...
        return resource;
    }
    ...
};
```

With each resource type we connect a trait being a class whose only characteristic is its inheritance chain. The user of a specific resource has the responsibility to provide a specialization of the generic trait (that has no ancestors or inheritants) that inherits from another specialization.

Listing 3. The `LockProxyTrait` template, and an example of its specialization

```
template<class T> class LockProxyTrait {
};

template<> class LockProxyTrait<Node> : public LockProxyTrait<Root>
};
```

There is a risk that the user may track back on lock ordering and start a new chain of locks from a point being a predecessor of already held proxies. To avoid this, the `LockProxy` has a constructor that requires the passing of a "parent" proxy. Such parent should possess a unique per-thread token, that in turn is passed to the created child. The only proxy allowed to be used to create children is the one actually possessing the lock token. This is always passed down the ordering relation, that clearly shows that no cyclic locks can occur. In the same constructor we assign the children trait to the parent trait. If this assignation fails at compile time that shows that we have taken the wrong order of proxy creation that could lead to deadlocks and have to restructure my algorithm. The `LockProxy` is a stack-based, uncoppiable object. As a result, it cannot be passed between threads. So, its definition guarantees us that by itself it is thread-safe.

Listing 4. The `LockProxy` constructor

```
template<class T> class LockProxy {
protected:
    //guarantee that it is stack based and thread-safe
    static void * operator new ( size_t n ){}
    LockProxy & operator= ( const LockProxy& lp ){}
    LockProxy ( const LockProxy& lp ){}
public:
    bool per_thread_locktoken;
    LockProxyTrait<T> trait;
};

template <class T> class WriteLockProxy : public LockProxy<T> {
...
public:
```

```
    template class <T1> WriteLockProxy (LockProxyTmpl<T1> &parent,
          ResourceId id ) :
          LockProxy<T>(parent,id) {
          if(!parent.per_thread_locktoken)
            throw new InvalidLockingPathException();
          parent.trait = this->trait;// test the relation in compile time
          resource=ObjectRepository::getResource( id );
          ObjectRepository::writeLockResource( id );
          parent.per_thread_locktoken=false;
          this->per_thread_locktoken=true;
          ...
    }
};
```

The specializations inheritance chain starts from a `LockProxyTrait<Root>` that keeps the top of the hierarchy. It is a template specialization of a `Root` class with no functionality, in order to maintain a consistent notation for all proxies. There is an issue that we have to deal here with – the fact that the root proxy has no parent that can provide him a lock token. That issue is solved by a thread-based Singleton – the `RootFactory` that is the generator of the unique per thread lock tokens. As the implementation of such a facility is thread-model specific, I will not present it here.

From this `LockProxyTrait<Root>` should inherit directly the trait corresponding to the resource type that ought to be locked first, then from it the next one in locking sequence, and so on.

We use the `LockProxy` templates to traverse through the structure of resources. Listing 5 presents a simple example of traversal of a hierarchy of classes, after declaring the proper locking order through the `LockProxyTrait` template specialization.

Listing 5. Traversing a class hierarchy

```
void traversal() {
...
ReadLockProxy<Root> root;//required to acquire the lock below
ReadLockProxy<Node> (root,nodeid) nodeproxy;
//a~Node has children accessible through getChild(int)
    for (int i=0 ; i~< nodeproxy->getChildrenNum() ; i++ ) {
        ReadLockProxy<Leaf> (nodeproxy,nodeproxy->getChild(i)) leafproxy;
        ...
        //do something with leafproxy
        cout << leafproxy->getValue() << endl;
        //block end causes the release of the ReadLockProxy
    }
}
```

The next element of the implementation is the `ResourceRepository`. This class has two purposes. One is `Mutex` handling, the other one is object referencing by a unique ID. As the pattern is to be non-intrusive – ResourceRepository is the place where we connect `Mutex`-es with specific resources. Each resource that we want to use

has to be registered first in the repository. Registration causes a `Mutex` to be assigned to a resource. As a result we obtain a unique ResourceId that is used to reference the objects by the proxies. Adding this facility helps to avoid the cases where we could access a resource without the intermediary of a LockProxy, or with no Mutex associated. As a unique ID, we use here for simplicity, a pointer cast to a long. For a more elaborated application reference counting could be much more appropriate.

Listing 6. ObjectRepository for a non-intrusive Mutex handling

```
typedef long ResourceId;

class ObjectRepository {
    static map< ResourceId , Mutex*> mutexRepository;
    static set< ResourceId > repository;
public:
    ResourceId getIdForObject(void * p) {
        ResourceId id = reinterpret_cast<ResourceId> (p);
        if(repository.find(id)==repository.end()) {
            repository.insert(id);
            mutexRepository[id]=new Mutex();
        }

    void * getResource() { ... }
    void getWriteLock(ResourceId id) {
        mutexRepository[id]->getWriteLock();
    }
...
}
```

5. Extensions of the pattern to same type monitors

Till now – the pattern that we have seen controls deadlocks at compile-time only for monitors of different types. To expand the control to monitors of the same type, we have to define the locking order and the resulting relation for instances of the same type. That can be done straightforward as the comparision between corresponding objects IDs. To use it at compile-time we must be able to obtain from a higher level type a vector of his children. That way the first created child keeps that vector and iterates over it – always in the direction of growing ids. Such a `LockIterator` has a `next()` operation causing him to release the current object and lock the next one following the kept child vector. As we maintain a partial ordering of monitors, we don't risk a deadlock. And the sequential access using `next()` does not allow us to create a deadlocking code. This Iterator class is strongly dependent on the interface of the monitors and as such must be a specific specialization of the templates. The iteration of Listing 5 when based on a LockProxyIterator as described above could be changed into:

Listing 7. Extending the pattern to same type monitors

```
ReadLockIterator<Leaf> leafproxy (nodeproxy);
while ( !leafproxy.end() );  ) {
    ...
    //do something with leafproxy
    cout << leafproxy.getValue() << endl;
    leafproxy.next();
}
```

6. Safe places to discard locking

In the special case when for a chosen resource there is always a specific parent monitor that exists on every path of leading to our resource - we can safely relax the discipline of checking for locking conditions after obtaining a write lock on this parent monitor. A write lock obtained on such a parent guarantees that no other thread can access the resource that are only accessible through that specific monitor. That way we can safely use a `RandomAccessLockProxy` that does not lock any mutexes. `RandomAccessLockIterators` can become bidirectional iterators. The order of creating additional LockProxies can also be simplified – as long as the lock token does not wander above the mentioned parent in hierarchy. Such a gatekeeper parent that always can be found is the root `LockProxy`. Obtaining a write lock on it denies to any other thread the access to any monitor. As the possibility of having only one way of accessing a resource is stricly connected with the interfaces of the monitors and the way we access a specific resource ID, such proxies should be realized by specific specialization of the templates presented.

7. Usage tips

The first step in using this technique is to plan thoroughly the nesting order that will be used. As a rule of thumb, the longer we need to keep a lock, the higher in hierarchy it should be. But this is not an absolute rule as in the case of a breadth-first search through a structure of monitors. Sometimes we would like to start to lock the search queue, get a queue entry, lock objects corresponding to the entry, create an entry, place it on the queue. That describes a cyclic access – or at least a high contention of the search queue. The solution is to break this cycle by giving a very low rank to the search queue and a high one to queue entries. That will force us to release the queue immediatelly after usage. So a proper lock hierarchy, can enforce policies for lock management in our project.

A place where it is safe to discard this pattern are resources that do not nest other critical section in their critical sections. That observation is used in the code of such classes as the `RootDirectory` or the `ResourceRepository`. Both have members that need to be protected from race conditions, and we have achieved that by the

usage of mutexes. But as they do not create nested locks of any addtional mutexes, this is a safe place to avoid the usage of the `LockProxy` pattern.

The ordering relation is also optimal when it is a strong relation (any two types are comparable). Optimal in this case means that we have the best coverage of the resource space, and at each step the amount of locks that we can acquire is as big as possible. That characteristic translates into the requirement that, if possible, the relation between traits should be that of a single base inheritance.

8. Final remarks

This pattern can be sometimes cumbersome if the amount of different monitors is low. In this case a simpler solution based on the observations made in this article could be an option. The system in which this pattern is used deals with over two dozens of monitor types, and with very different tasks running simultanously. Any pattern relieving the programmer from taking directly care of all the issues is of a big help.

References

[1] Stroustrup B.: *The C++ Programming Language*. 3rd Edition, Addison Wesley Longman, Inc., 2000

[2] Stroustrup B.: *Wrapping C++ Member Function Calls*. The C++ Report. Vol 12, No 6, June 2000

[3] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc. 1994

[4] Karlsson, B.: *Beyond the C++ Standard Library: An Introduction to Boost*. Addison Wesley Longman, Inc. 2005

[5] Lea D.: *Concurrent Programming in Java*. 2nd Edition, The Java Series 1999

[6] Ben-Ari M.: *Principles of Concurrent Programming*. Prentice-Hall International 1982