Generic Synchronization Policies in C++

Ciaran McHale
www.CiaranMcHale.com

Contents

1 Introduction **3**

2 Scoped Locks **3**

3 Generic Synchronization Policies **5**
3.1 The Mutex and Readers-writer Policies 6
3.2 The Producer-consumer Policy 7
3.3 The Bounded Producer-consumer Policy 7

4 Generic Synchronization Policies in C++ **8**

5 Support for Generic Synchronization Policies in Other Languages **11**

6 A Critique of Generic Synchronization Policies **12**
6.1 Strengths of GSPs 12
6.2 Potential Criticisms of GSPs 12

7 Issues Not Addressed by GSPs **13**
7.1 Thread Cancellation 13
7.2 Timeouts 14
7.3 Lock Hierarchies 14

8 GSP Class Library **15**

9 Acknowledgments **15**

10 About the Author **16**

This software and documentation are distributed under an MIT-style license (shown below), which basically means you must not remove the copyright notice but, aside from that, you can use or modify this software and documentation as you want. For example, you can use it in both open-source and closed-source projects, and you can give away the software and documentation for free or you can sell it. You can find information about open-source licenses from the *Open Source Initiative* (www.opensource.org).

Copyright © 2006–2008 Ciaran McHale.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

- The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
- THE SOFTWARE 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 AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1 Introduction

Writing synchronization code is usually both difficult and non-portable.

Much of the difficulty in writing synchronization code is due to the use of low-level synchronization APIs.

The portability problem arises because neither C nor C++ provide a standard library for synchronization. As a result, many operating systems provide proprietary APIs for synchronization. Some people write a portability-layer library that hides the proprietary APIs of the underlying operating systems. Unfortunately, it is common for such libraries to provide a low-level, and hence difficult to use, API.

In my experience, most uses of synchronization code in multi-threaded applications fall into a small number of high-level "usage patterns", or what I call *generic synchronization policies* (GSPs). This paper illustrates how the use of such GSPs simplify the writing of thread-safe classes. In addition, this paper presents a C++ class library that implements commonly-used GSPs.

2 Scoped Locks

Threading libraries provide functions that acquire and release mutual exclusion (mutex) locks. In this paper, I use `getLock()` and `releaseLock()` to denote such functions. A critical section is code that is bracketed with calls to `getLock()` and `releaseLock()`. For example, the following is a critical section:

```
getLock(mutex);
...
releaseLock(mutex);
```

The above critical section looks simple. However, if the code within a critical section has conditional `return` statements or conditionally throws exceptions then care must be taken to release the mutex lock at *all* exit points of the critical section. Figure 1 shows an example of this.

In general, adding calls to `releaseLock()` at every potential exit point of a critical section clutters up the code, thus making the code more difficult to read and maintain. Indeed, a common source of bugs in multi-threaded applications is forgetting to add a call of `releaseLock()` at some of the possible exit points of a critical section.

There is a useful technique that removes the need for the cluttering, error-prone calls to `releaseLock()`. This technique involves writing a class—let's call it `ScopedMutex`—that calls `getLock()` and `releaseLock()`

```
void foo()
{
    getLock(mutex);
    ...
    if (...) {
        releaseLock(mutex);
        return;
    }
    if (...) {
        releaseLock(mutex);
        throw anException;
    }
    ...
    releaseLock(mutex);
}
```

Figure 1: An operation with a critical section

in its constructor and destructor. You can see a pseudocode implementation of this class in Figure 2.

```
class ScopedMutex {
public:
    ScopedMutex(Mutex & mutex)
        : m_mutex(mutex)
    {
        getLock(m_mutex);
    }
    ~ScopedMutex() {
        releaseLock(m_mutex);
    }
protected:
    Mutex &    m_mutex;
};
```

Figure 2: A pseudocode `ScopedMutex` class

Now, instead of explicitly calling `getLock()` and `releaseLock()` in the body of an operation, you can just declare a `ScopedMutex` variable local to the function. Figure 3 shows an example of this. When the function is called, the constructor of the `ScopedMutex` variable is invoked and this

calls `getLock()`. Then, when the function terminates, the destructor of the `ScopedMutex` variable is invoked and this calls `releaseLock()`. This happens regardless of whether the function returns early (line 1), throws an exception (line 2) or runs to completion (line 3).

```
void foo()
{
    ScopedMutex    scopedLock(mutex);
    ...
    if (...) return;              // line 1
    if (...) throw anException;   // line 2
    ...
}                                 // line 3
```

Figure 3: An operation with a scoped mutex lock

If you compare the code in Figures 1 and 3 then you will see that the latter code (which uses the `ScopedMutex` class) is shorter and easier to read than the former.

This technique of using a constructor/destructor class for synchronization is *partially* well-known within the C++ community. I say this for two reasons.

First, my experience as a consultant and trainer has given me the opportunity to work with many C++ programmers in many different organization. I have found that about half the programmers I work with are familiar with this technique and view it as being a basic C++ idiom, while the same technique is new to the other half.

Second, among programmers who have used this constructor/destructor technique for synchronization, usage of this technique invariably is confined to mutual exclusion (and *occasionally* readers-writer locks). However, this technique is applicable to other, more complex synchronization code too (which is the focus of this paper).

Before discussing how to apply this technique to other synchronization code, it is necessary to take a slight detour. In particular, I have to introduce a new concept: that of *generic synchronization policies*.

3 Generic Synchronization Policies

C++ supports *template* types. For example, a list class might be written as:

```
template<T> class List { ... };
```

Once implemented, this template type can be instantiated multiple times to obtain, say, a list of `int`, a list of `double` and a list of `Widget`:

```
List<int>       myIntList;
List<double>    myDoubleList;
List<Widget>    myWidgetList;
```

The ability to define template types is not unique to C++. Several other languages provide similar functionality, although often there are differences in terminology and syntax. For example, some languages use the term *generic types* rather than *template types*, and the type parameters might be enclosed within `[` and `]` instead of `<` and `>`.

The concept of genericity is not restricted to types. It can be applied to synchronization too, as I now discuss.

3.1 The Mutex and Readers-writer Policies

Using a pseudocode notation, I can declare some well-known synchronization policies as follows:

```
Mutex[Op]
RW[ReadOp, WriteOp]
```

In this notation, the name of the generic synchronization policy is given first, and is then followed by a parameter list enclosed in square brackets. Each parameter denotes a set of operation names. For example, the `Mutex` policy is instantiated upon a set of operations (`Op`), while the `RW` (readers-writer) policy is instantiated upon a set of read-style operations (`ReadOp`) and a set of write-style operations (`WriteOp`).

Consider a class that has two read-style operations called `Op1` and `Op2`, and a write-style operation called `Op3`. I instantiate the `RW` policy upon these operations as follows:

```
RW[{Op1, Op2}, {Op3}]
```

Likewise, an instantiation of the `Mutex` policy upon these three operations is written as follows:

```
Mutex[{Op1, Op2, Op3}]
```

3.2 The Producer-consumer Policy

The producer-consumer policy is useful when a buffer is used to transfer data from one thread (the producer) to another thread (the consumer). The producer thread *puts* items into the buffer and then, sometime later, the consumer thread *gets* these items. If the consumer thread tries to get an item from an empty buffer then it will be blocked until the buffer is not empty. Furthermore, the put-style and get-style operations execute in mutual exclusion; this is to prevent the buffer from becoming corrupted due to concurrent access. This policy is written as follows:

```
ProdCons[PutOp, GetOp, OtherOp]
```

`OtherOp` denotes any other (non put-style and non get-style) operations on the buffer class. For example, perhaps there is an operation on the buffer that returns a count of how many items are currently in the buffer. Such an operation might need to run in mutual exclusion with the put-style and get-style operations to ensure its correct operation. If a buffer-style class has operations called `insert()`, `remove()` and `count()` then you can instantiate the `ProdCons` policy on the class as follows:

```
ProdCons[{insert}, {remove}, {count}]
```

If the class does not have a `count()` operation then you can instantiate the `ProdCons` policy on it as follows:

```
ProdCons[{insert}, {remove}, {}]
```

In this case, the `OtherOp` parameter of the policy is instantiated upon an *empty* set of operations names.

3.3 The Bounded Producer-consumer Policy

A common variation of the producer-consumer policy is the *bounded* producer-consumer policy, in which the buffer has a fixed size. This prevents the buffer from growing infinitely large if one thread puts items into the buffer faster than the other thread can get them. In this policy, if the producer thread tries to put an item into an already-full buffer then it will be blocked until the buffer is not full. This policy is written as follows:

```
BoundedProdCons(int size)[PutOp, GetOp, OtherOp]
```

Notice that the size of the buffer is specified as a parameter to the name of the policy. Such parameters are usually instantiated upon a corresponding parameter to the constructor of the buffer; an example of this will be shown later (in Figure 7 on page 11).

4 Generic Synchronization Policies in C++

The discussion in Section 3 focussed on the *concept* of GSPs. I now explain how to implement GSPs in C++.

Figure 4 shows the mapping of the `Mutex[Op]` policy into a C++ class using POSIX threads. Note that, in order to keep the code concise, error checks on the return values of the POSIX threads library calls have been omitted.

```
 1 class GSP_Mutex {
 1 public:
 3     GSP_Mutex()  { pthread_mutex_init(m_mutex, 0); }
 4     ~GSP_Mutex() { pthread_mutex_destroy(&m_mutex); }
 5
 6     class Op {
 7     public:
 8         Op(GSP_Mutex &) : m_data(data)
 9               { pthread_mutex_lock(&m_data.m_mutex); }
10         ~Op() { pthread_mutex_unlock(&m_data.m_mutex); }
11     protected:
12         GSP_Mutex &    m_data;
13     };
14
15     protected:
17         pthread_mutex_t    m_mutex;
16         friend class ::GSP_Mutex::Op;
18 };
```

Figure 4: Mapping of `Mutex[Op]` into a C++ class

The mapping from a GSP into a C++ class is performed as follows:

1. The name of the C++ class is the same as the name of the GSP, but with a `"GSP_"` prefix. The prefix is used to prevent name-space pollution. So, in Figure 4 the `Mutex` GSP is implemented by the `GSP_Mutex` class.

2. The class has one or more instance variables (line 17) that provide storage for the mutex. The constructor and destructor of the class (lines 3 and 4) initialize and destroy the instance variable(s).

3. The `Mutex[Op]` GSP has a parameter called `Op`. This translates into a nested class (lines 6–13) with the same name. If a GSP has several parameters then each parameter translates into a separate nested class; an example of this will be shown later.

4. Each nested class has one instance variable (line 12), which is a reference to the outer class. This instance variable is initialized from a parameter to the constructor of the inner class (line 8).

5. The constructor and destructor of the nested class get and release the lock (lines 9 and 10) stored in the instance of the outer class.

As another example, Figure 5 shows how the `RW[ReadOp, WriteOp]` GSP maps into a C++ class. Notice that because this GSP takes two parameters, there are two nested classes.

```
class GSP_RW {
public:
    GSP_RW()  { /* initialize the readers-writer lock */ }
    ~GSP_RW() { /* destroy the readers-writer lock */ }

    class ReadOp {
    public:
        ReadOp(GSP_RW & data) : m_data(data)
                        { /* acquire read lock */ }
        ~ReadOp()       { /* release read lock */ }
    protected:
        GSP_RW &    m_data;
    };

    class WriteOp {
    public:
        WriteOp(GSP_RW & data) : m_data(data)
                        { /* acquire write lock */ }
        ~WriteOp()      { /* release write lock */ }
    protected:
        GSP_RW &    m_data;
    };

protected:
    ...  // Instance variables required to implement a
         // readers-writer lock
    friend class ::GSP_RW::ReadOp;
    friend class ::GSP_RW::WriteOp;
};
```

Figure 5: The `GSP_RW` class with nested `ReadOp` and `WriteOp` classes

Instantiating a GSP upon the operations of a C++ class involves the following three steps:

1. `#include` the header file for the GSP. The name of the header file is the same as name of the GSP class, but written in lowercase letters. For example, the header file for the `GSP_RW` class is `"gsp_rw.h"`.

2. Add an instance variable to the C++ class that is to be synchronized. The instance variable's type is that of the GSP's outer class.

3. Inside the body of each operation that is to be synchronized, declare a local variable, the type of which is that of a nested class of the GSP.

The instantiation of `RW[{Op1, Op2}, {Op3}]` in Figure 6 illustrates these steps.

```
#include "gsp_rw.h"
class Foo {
public:
    Foo()  { ... }
    ~Foo() { ... }
    void Op1(...) {
        GSP_RW::ReadOp     scopedLock(m_sync);
        ... // normal body of operation
    }
    void Op2(...) {
        GSP_RW::ReadOp     scopedLock(m_sync);
        ... // normal body of operation
    }
    void Op3(...) {
        GSP_RW::WriteOp    scopedLock(m_sync);
        ... // normal body of operation
    }
protected:
    GSP_RW    m_sync;
    ... // normal instance variables of class
};
```

Figure 6: Instantiation of `GSP_RW`

As a final example, Figure 7 shows a class that is instantiated with:

```
BoundedProdCons(int size)[PutOp, GetOp, OtherOp]
```

This policy takes a parameter that indicates the size of the buffer. This parameter is obtained from the `bufSize` parameter of the class's constructor.

```
#include "gsp_boundedprodcons.h"
class WidgetBuffer {
public:
    WidgetBuffer(int bufSize) : m_sync(bufSize) { ... }
    ~WidgetBuffer()                             { ... }

    void insert(Widget * item) {
        GSP_BoundedProdCons::PutOp scoped_lock(m_sync);
        ... // normal body of operation
    }

    Widget * remove() {
        GSP_BoundedProdCons::GetOp scoped_lock(m_sync);
        ... // normal body of operation
    }
protected:
    GSP_BoundedProdCons    m_sync;
    ... // normal instance variables of class
};
```

Figure 7: Instantiation of `GSP_BoundedProdCons`

5 Support for Generic Synchronization Policies in Other Languages

The implementation of GSPs shown in this paper relies upon constructors and destructors to automate the execution of synchronization code. Although object-oriented languages usually provide constructors, not all object-oriented languages provide destructors, especially languages that have built-in garbage collectors. This may lead readers to conclude that GSPs cannot be implemented in existing languages that do not provide destructors. While this may be so, it would be possible for designers of *future* languages to incorporate support for GSPs into their language design. For example, in my Ph.D. thesis [McH94] I show how to add support for GSPs to the compiler of an object-oriented language that uses garbage collection instead of destructors.

6 A Critique of Generic Synchronization Policies

I now point out some benefits and potential drawbacks of GSPs.

6.1 Strengths of GSPs

First, GSPs provide a good form of skills reuse. In particular, it is a lot easier to *use* a GSP than it is to *implement* one. Thus, a programmer skilled in synchronization programming can implement whatever GSPs are needed for a project, and then other, lesser skilled, programmers can use these pre-written GSPs.

Second, GSPs aid code readability and maintainability by separating synchronization code from the "normal", functional code of a class.

Third, as I discussed in Section 2, placing synchronization code in the constructor and destructor of the GSP classes means that locks are released even if an operation terminates by returning early or throwing an exception. This eliminates a common source of bugs in multi-threaded programs.

Fourth, GSPs provide not only ease of use; they also provide a portability layer around the underlying synchronization primitives. Of course, some companies have developed in-house, portability libraries that hide the differences between synchronization primitives on various platforms, and some other companies make use of third-party portability libraries, such as the Threads.h++ library from RogueWave. The use of GSPs is compatible with such existing libraries: GSPs can be implemented just as easily on top of Threads.h++ (or some other portability library) as they can be implemented directly on top of operating-system specific synchronization primitives.

Finally, the implementation of a GSP can be inlined. Thus, the use of GSPs need not impose a performance overhead.

6.2 Potential Criticisms of GSPs

Some readers might be thinking: "GSPs are limited; they cannot handle *all* the synchronization needs I might have." However, in many activities, a disproportionately large amount of results come from a relatively small amount of investment. This is generally known as the 80/20 principle [Koc00]. In my experience, this applies to the synchronization requirements of applications. A small set of GSPs is likely to suffice for most of the synchronization needs of programmers. So, even if a small set of pre-written GSPs cannot handle *all* the synchronization needs that a programmer will face, the 80/20 principle suggests that the use of GSPs would be useful *often enough* to justify their use.

Of course, people are *not* restricted to just a small set of pre-written GSPs. People can define new GSPs. For example, perhaps a programmer needs to write some project-specific synchronization code. Even if this synchronization code will be used in just one place in the project, it is hardly any additional work to implement this as a GSP and then instantiate it, rather than to implement it "in-line" in the operations of a class. Doing do offers several benefits:

1. Implementing the synchronization code as a GSP is likely to improve readability and maintainability of the synchronization code *and* the sequential code of the project.

2. If the programmer later discovers another place that needs to use the same policy then the GSP can be re-used directly, rather than having to re-use in-lined code via copy-n-paste.

Some other readers might be thinking: "GSPs are not new; 'GSP' is just a new name for an existing C++ programming idiom". The claim that GSPs are based on an already-known C++ idiom (the `ScopedMutex` class discussed in Section 2) is entirely true. Indeed, the `ScopedMutex` class is a GSP in all but name. However, as discussed in Section 2, the C++ idiom that underlies GSPs was previously used only for mutex and *occasionally* the readers-writer policies. A significant contribution of GSPs is in pointing out that the same technique can be used for most, if not all, synchronization policies.

7 Issues Not Addressed by GSPs

GSPs illustrate the 80/20 principle: most of the synchronization requirements of programmers can be satisfied by a small collection of GSPs. However, there are some synchronization issues that are *not* tackled by GSPs. I now briefly discuss these issues below, so that readers can be forewarned about when the use of GSPs is not suitable.

7.1 Thread Cancellation

The POSIX threads specification provides a mechanism for a thread to be *cancelled*, that is, terminated gracefully. When a thread is cancelled, it is important that the thread has a chance to do some tidying up before it is terminated, for example, the thread may wish to release locks that it holds. This is achieved by having the programmer register callback functions that will be invoked in the event of the thread being cancelled. The current implementation of GSPs does not provide support for the thread cancellation capability of POSIX threads.

This is not due to any intrinsic incompatibility between GSPs and thread cancellation. Rather it is simply due to the author never having needed to make use of thread cancellation. Integrating GSPs with thread cancellation is left as an exercise to interested readers.

7.2 Timeouts

Some thread packages provide a timeout capability on synchronization primitives. By using this, a programmer can specify an upper time limit on how long a thread should wait to, say, get a lock. The current implementation of GSPs does *not* provide a timeout capability. There are two reasons for this.

First, timeouts are rarely needed and hence, by following the 80/20 principle, I decided to not bother supporting them.

Second, implementing a timeout capability is relatively complex with the APIs of some threads packages. For example, a `Mutex` policy *without* a timeout capability can be implemented trivially in POSIX threads by invoking functions upon an underlying mutex type. In contrast, implementing a `Mutex` policy *with* a timeout capability in POSIX threads necessitates the use of a mutex variable *and* a condition variable; the resulting algorithm is more complex to write and maintain, and it incurs *at least* twice as much performance overhead as a `Mutex` without a timeout capability. This additional performance overhead suggests that if some programmers decide they require a `Mutex` policy with a timeout capability then they should implement it as a *new* GSP, say, `TimeoutMutex`, rather than add the timeout capability to the existing `Mutex` policy. In this way, programmers can use the `TimeoutMutex` policy on the few occasions when they need to, and can use the more efficient `Mutex` policy on all other occasions.

7.3 Lock Hierarchies

GSPs are useful for classes or objects that have self-contained synchronization. However, sometimes the synchronization requirements of *several* classes are closely intertwined, and a programmer needs to acquire locks on two (or more) objects before carrying out a task. The need to acquire locks on several objects at the same time is commonly referred to as a *lock hierarchy*. Attempting to acquire a lock hierarchy can result in deadlock if done incorrectly. Algorithms for acquiring lock hierarchies safely are outside the scope of this paper, but can be found elsewhere [But97]. The point to note is that algorithms for acquiring lock hierarchies safely require unhindered access to the locking primitives.

This is in opposition to GSPs, which completely encapsulate the underlying synchronization primitives.

8 GSP Class Library

This paper accompanies a library of GSP classes. You can download this paper, and its library from www.CiaranMcHale.com/download. The library implements all the GSPs discussed in this paper, that is, `GSP_Mutex`, `GSP_RW`, `GSP_ProdCons` and `GSP_BoundedProdCons`. The library implements these GSPs for the following threads packages: Solaris threads, DCE Threads, POSIX threads and Windows threads. Dummy implementations of these GSPs for non-threaded systems are also provided; this makes it possible to write a class that can be used in both sequential and multi-threaded applications.

All the GSP classes are implemented with inline code in header files. Because of this, to make use of a GSP you need only `#include` the corresponding header file; there is no GSP library to link into your application. The name of the header file for a GSP is the same as the name of the GSP class, but written in lowercase letters. For example, the header file for the `GSP_RW` class is `"gsp_rw.h"`.

You should use the `-D<symbol_name>` option on your C++ compiler to define one of the following symbols:

- `P_USE_WIN32_THREADS`
- `P_USE_POSIX_THREADS`
- `P_USE_DCE_THREADS`
- `P_USE_SOLARIS_THREADS`
- `P_USE_NO_THREADS`

The symbol instructs the GSP class which underlying threading package it should use.

9 Acknowledgments

The concept of GSPs has its roots in my Ph.D. thesis [McH94]. Research for this Ph.D. was partially funded by the Comandos ESPRIT project and was carried out while I was a member of the Distributed Systems Group (DSG) in the Department of Computer Science, Trinity College, Dublin, Ireland. I

wish to thank DSG for their support. I also wish to thank colleagues in IONA Technologies for their comments on earlier drafts of this paper.

10 About the Author

Ciaran McHale works for IONA Technologies (www.iona.com), which specializes in standards-based, distributed middleware. He has worked there since 1995 and is a principal consultant. He has a Ph.D. in computer science from Trinity College, Dublin, Ireland. Interested readers can find more information about Ciaran McHale at his personal web-site: www.CiaranMcHale.com.

References

[But97] David Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[Koc00] Richard Koch. *The 80/20 Principle: The Secret of Achieving More With Less*. Nicholas Breealey Publishing Ltd., May 2000. ISBN: 187881680. 312 pages.

[McH94] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity College, Dublin 2, Ireland, October 1994. Available for download at www.CiaranMcHale.com/download/phd-thesis.pdf.