

Generic Synchronization Policies in C++

Ciaran McHale

CiaranMcHale.com
Complexity explained simply

1

License

Copyright (c) 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.

Generic Synchronization Policies in C++

2

Introduction

- Most people know that writing synchronization code is:
 - Difficult: APIs are low-level
 - Non-portable: many threading APIs: POSIX, Windows, Solaris, DCE, ...
- In practice, most synchronization code implement a small number of high-level "usage patterns":
 - Let's call these *generic synchronization policies* (GSPs)
 - The most common GSPs can be implemented as a C++ library
- Using GSPs in applications:
 - Is much easier than using low-level APIs
 - Encapsulates the underlying threading package → provides portability

Generic Synchronization Policies in C++

3

1. Scoped Locks

4

Critical section

- The following (pseudocode) function uses a critical section:

```
void foo()
{
    getLock(mutex);
    ...
    releaseLock(mutex);
}
```

- The above code is very simple. However...
- Complexity increases if the function has several exit points:
 - Because `releaseLock()` must be called at each exit point
 - Examples of extra exit points:
 - Conditional `return` statements
 - Conditionally throwing an exception

Generic Synchronization Policies in C++

5

Critical section with multiple exit points

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

Have to call `releaseLock()` at every exit point from the function

Generic Synchronization Policies in C++

6

Critique

- Needing to call `releaseLock()` at every exit point:
 - Clutters up the “business logic” code with synchronization code
 - This clutter makes code harder to read and maintain
- Forgetting to call `releaseLock()` at an exit point is a common source of bugs

- There is a better way...

Generic Synchronization Policies in C++

7

Solution: ScopedMutex class

- Define a class called, say, `ScopedMutex`:
 - This class has no operations! Just a constructor and destructor
 - Constructor calls `getLock()`
 - Destructor calls `releaseLock()`
- Declare a `ScopedMutex` variable local to a function
 - At entry to function → constructor is called → calls `getLock()`
 - At exit from function → destructor is called → calls `releaseLock()`
- The following two slides show:
 - Pseudocode implementation of `ScopedMutex` class
 - Use of `ScopedMutex` in a function

Generic Synchronization Policies in C++

8

The ScopedMutex class

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

    ~ScopedMutex ()
    { releaseLock(m_mutex); }
private:
    Mutex & m_mutex;
};
```

Generic Synchronization Policies in C++

9

Use of ScopedMutex

```
void foo()
{
    ScopedMutex    scopedLock (mutex) ;
    ...
    if (...) { return; }
    if (...) { throw anException; }
    ...
}
```

No need to call `releaseLock()` at every exit point from the function!

Generic Synchronization Policies in C++

10

Comments on ScopedMutex

- This technique is *partially* well known in the C++ community:
 - 50% of developers the author worked with already knew this technique
 - They considered it to be a “basic” C++ coding idiom
 - Other 50% of developers had not seen the technique before
- Of the developers who already knew this technique:
 - They all used it for mutex locks
 - Only a few knew it could be used for readers-writer locks too
 - Nobody knew it could be used for almost any type of synchronization code
- Contribution of this presentation:
 - Generalize the technique so it can be used much more widely
- To explain how to do this, I need to take a slight detour:
 - Have to introduce the concept of *generic synchronization policies*

Generic Synchronization Policies in C++

11

2. The Concept of Generic Synchronization Policies

12

Genericity for types

- C++ provides template types
- Example of a template type definition:

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

- Examples of template type instantiation:

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

- Some other languages provide a similar capability, often with different terminology and syntax
 - Perhaps called *generic types* instead of *template types*
 - Perhaps surround type parameters with [] instead of <>

Generic Synchronization Policies in C++

13

Genericity for synchronization policies

- Using a pseudocode notation, here are declarations of mutual exclusion and readers-writer policies

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

- In above examples, each parameter is a set of operations
- Example instantiations on operations Op1, Op2 and Op3

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

Generic Synchronization Policies in C++

14

Producer-consumer policy

- Useful when:
 - A buffer is used to transfer data between threads
 - A producer thread *puts* items into the buffer
 - A consumer thread *gets* items from the buffer
 - If the buffer is empty when the consumer tries to get an item then the consumer thread blocks
 - The buffer might have *other* operations that examine the state of the buffer

- In pseudocode notation, the policy declaration is:

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

- Example instantiations:

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

Generic Synchronization Policies in C++

15

Bounded producer-consumer policy

- Variation of the producer-consumer policy:
 - Buffer has a fixed size
 - If the buffer is full when the producer tries to put in an item then the producer thread blocks

- In pseudocode notation, policy is:

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

- Typically, the *size* parameter is instantiated on a parameter to the constructor of the buffer class
 - An example instantiation will be shown later

Generic Synchronization Policies in C++

16

3. Generic Synchronization Policies in C++

17

Mapping Mutex[Op] into C++

```
class GSP_Mutex {
public:
    GSP_Mutex() { /* initialize m_mutex */ }
    ~GSP_Mutex() { /* destroy m_mutex */ }

    class Op {
    public:
        Op(GSP_Mutex & data) : m_data(data)
        { getLock(m_data.m_mutex); }
        ~Op()
        { releaseLock(m_data.m_mutex); }
    private:
        GSP_Mutex & m_data;
    };

private:
    friend class ::GSP_Mutex::Op;
    OS-specific-type m_mutex;
};
```

Class name = "GSP_" + name of policy

Constructor & destructor of outer class initialize and destroy locks

A nested class for each policy parameter

Constructor & destructor of nested class get and release locks stored in the outer class

Generic Synchronization Policies in C++

18

Mapping RW[ReadOp, WriteOp] into C++

```
class GSP_RW {
public:
    GSP_RW();
    ~GSP_RW();

    class ReadOp {
    public:
        ReadOp(GSP_RW & data);
        ~ReadOp();
    };

    class WriteOp {
    public:
        WriteOp(GSP_RW & data);
        ~WriteOp();
    };
};
```

This policy has two parameters so there are two nested classes

Generic Synchronization Policies in C++

19

Mapping BoundedProdCons into C++

■ This is the mapping for

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

```
class GSP_BoundedProdCons {
public:
    GSP_BoundedProdCons(int size);
    ~GSP_BoundedProdCons();

    class PutOp {...};
    class GetOp {...};
    class OtherOp {...};
};
```

The size parameter to the policy maps into a parameter to the constructor of the class

This policy has three parameters so there are three nested classes

Generic Synchronization Policies in C++

20

Instantiating GSP_RW[ReadOp, WriteOp]

```
#include "gsp_rw.h"

class Foo {
private:
    GSP_RW m_sync;
public:

    void op1(...) {
        GSP_RW::ReadOp scopedLock(m_sync);
        ...
    }

    void op2(...) {
        GSP_RW::WriteOp scopedLock(m_sync);
        ...
    }
};
```

#include header file (name of class written in lowercase)

Add instance variable whose type is name of policy's outer class

Synchronize an operation by adding a local variable whose type is a nested class of the policy

Generic Synchronization Policies in C++

21

4. Critique

23

Instantiating GSP_BoundedProdCons

```
#include "gsp_boundedprodcons.h"

class Buffer {
private:
    GSP_BoundedProdCons m_sync;
public:
    Buffer(int size) : m_sync(size) { ... }

    void insert(...) {
        GSP_BoundedProdCons::PutOp scopedLock(m_sync);
        ...
    }

    void remove(...) {
        GSP_BoundedProdCons::GetOp scopedLock(m_sync);
        ...
    }
};
```

The size parameter of the policy is initialized with value of a parameter to the constructor

Generic Synchronization Policies in C++

22

Strengths of GSPs

- Only one person needs to know how to implement GSPs
 - Trivial for everyone else to instantiate GSPs
- Separates synchronization code from “business logic” code
 - Improve readability and maintainability of both types of code
- Removes a common source of bugs:
 - Locks are released even if an operation throws an exception
- Improves portability:
 - API of GSPs does *not* expose OS-specific details of synchronization
- Efficiency:
 - GSPs can be implemented with inline code

Generic Synchronization Policies in C++

24

Potential criticisms fo GSPs

- “Can they handle *all* my synchronization needs?”
 - 80/20 principle: *most* synchronization needs can be handled by just a small library of GSPs
 - You are not restricted to a library of pre-written GSPs. Instead...
 - You can write new GSPs if the need arises
- “GSPs are just a `ScopedMutex` with a new name”
 - The “just” part is inaccurate
 - GSPs generalize the `ScopedMutex` concept so it can be used for a much wider set of synchronization policies

Generic Synchronization Policies in C++

25

Issues not addressed

- GSPs do not address:
 - POSIX thread cancellation
 - Timeouts
 - Lock hierarchies
- In the author’s work, these issues arise infrequently so he did not bother to support them
 - GSPs could probably be extended to support the above issues

Generic Synchronization Policies in C++

26

5. Ready-to-run GSPs

27

Ready-to-run GSPs

- A library of ready-to-use GSPs is available:
 - Download from www.CiaranMcHale.com/download
 - Documentation provided in multiple formats:
 - Manual: LaTeX (source), PDF & HTML
 - Slides: PowerPoint, PDF and N-up PDF
- Library contains all GSPs discussed in this paper:
 - `Mutex[Op]`
 - `RW[ReadOp, WriteOp]`
 - `ProdCons[PutOp, GetOp, OtherOp]`
 - `BoundedProdCons(int size)[PutOp, GetOp, OtherOp]`
- GSPs are implemented for multiple thread packages:
 - POSIX, Solaris, Windows, DCE
 - Dummy (for non-threaded applications)

Generic Synchronization Policies in C++

28

Using GSP classes

- Define one of the following preprocessor symbols before you `#include` a GSP header file
 - `P_USE_POSIX_THREADS`
 - `P_USE_SOLARIS_THREADS`
 - `P_USE_WIN32_THREADS`
 - `P_USE_DCE_THREADS`
 - `P_USE_NO_THREADS`
- Typically done with `-D<symbol>` command-line option to compiler

Summary

- GSPs are a generalization of the `ScopedMutex` class:
 - Out-of-the-box support for mutual-exclusion, readers-writer and (bounded) producer-consumer policies
 - You can write new GSPs if the need arises
- Benefits:
 - Makes it trivial to add synchronization to a C++ class
 - Makes code easier to read and maintain
 - Portability across multiple thread packages
 - Minimal performance overhead due to `inline` implementation
- All software and documentation is available:
 - MIT-style license (open-source, non-viral)
 - Download from www.CiaranMcHale.com/download