

Scoped Locking

Das C++-Idiom *Scoped Locking* garantiert, dass der Eintritt in einen Anweisungsblock automatisch eine neue Sperre akquiriert und dass das Verlassen des Blocks diese Sperre automatisch wieder freigibt – unabhängig davon, wie der Block verlassen wird.

Synchronized Block, Resource-Acquisition-is-Initialization [Str00],¹
Guard, Execute-Around Object [Hen00]

Auch bekannt als

Kommerzielle Webserver enthalten in der Regel einen »Trefferzähler«. Dieser zeigt an, wie viele Male Clients über einen bestimmten Zeitraum auf jede URL des Servers zugegriffen haben. Um Latenzzeiten zu minimieren, verwaltet ein Webserver-Prozess den Trefferzähler in einer speicherresidenten Komponente, anstatt in einer Festplattendatei.

Beispiel

Um ihren Durchsatz zu maximieren, sind Webserver-Prozesse meist nebenläufig [HS98]. Öffentliche (Public) Methoden in der Trefferzähler-Komponente müssen daher serialisiert werden. Dies hindert Ablaufäden daran, den internen Zustand dieser Komponente unbeabsichtigt zu verfälschen, wenn sie deren Trefferzähler nebenläufig aktualisieren.

Eine Möglichkeit, den Zugriff auf eine Trefferzähler-Komponente zu serialisieren, ist es, in jeder ihrer öffentlichen Methoden eine Sperre explizit zu akquirieren und freizugeben. Das folgende C++-Beispiel verwendet die im Wrapper-Facade-Muster (53) definierte `Thread_Mutex`-Klasse, um kritische Abschnitte zu serialisieren:

```
class Hit_Counter {
public:
    // Erhöhe den Trefferzaehler für die URL <path>.
    bool increment (const string &path) {
        // Akquiriere die Sperre und betrete dann den
        // kritischen Abschnitt.
        lock_.acquire ();
        Table_Entry *entry = lookup_or_create (path);
        if (entry == 0) {
            // Fehler!
            lock_.release ();
            return false; // Gib einen Fehlerwert aus.
        }
    }
};
```

1. Das Scoped-Locking-Idiom ist eine Spezialisierung des Idioms »Resource-Acquisition-is-Initialization« von Bjarne Stroustrup [Str00]. Wir führen dieses Idiom der Vollständigkeit halber in unserem Buch auf, um darzustellen, wie wir Stroustrup's Idiom für nebenläufige Programme verwenden können.

```

        else { // Erhoehe den Zaehler fuer die URL <path>.
            entry->increment_hit_count ();
            // Gib die Sperre vor dem Verlassen des
            // kritischen Abschnitts wieder frei.
            lock_.release ();
            return true;
        }
    }

    // Weitere oeffentliche Methoden...
private:
    // Schaeue in der Trefferzaehlertabelle nach, ob es
    // schon einen Zaehler fuer die URL <path> gibt,
    // und erzeuge diesen, falls nicht.
    Table_Entry *lookup_or_create (const string &path);

    // Serialisiere den Zugriff auf den kritischen
    // Abschnitt.
    Thread_Mutex lock_;
};

```

Obgleich dieser Hit_Counter-Code korrekt funktioniert, gestaltet sich seine Implementierung und Wartung jedoch als schwierig. Programmierer können u. a. vergessen, die lock_-Sperre in den Rücksprungpfaden aus der increment()-Methode freizugeben, z.B. dann, wenn wir ihre else-Funktion dahingehend modifizieren, eine neue Fehlermeldung zu überprüfen:

```

else if (entry->increment_hit_count () == SOME_FAILURE)
    return false; // Gib einen Fehlerwert zurueck.

```

Hinzu kommt noch, dass die Implementierung nicht ausnahmesicher ist: Die lock_-Sperre wird nicht freigegeben, falls eine spätere Version der increment()-Methode Ausnahmen wirft oder eine Hilfsmethode aufruft, die selbst Ausnahmen wirft [Mue96].

Beide Modifikationen haben den Effekt, dass die increment()-Methode terminiert, *ohne* die lock_-Sperre freizugeben. Wenn lock_ aber nicht freigegeben ist, »hängt« sich der Webserver-Prozess auf, sobald andere Ablaufäden versuchen, die lock_-Sperre zu akquirieren. Wenn solche Fehlersituationen auch noch sehr selten eintreten, werden mögliche Probleme mit dem Code beim Testen des Systems u. U. gar nicht bemerkt und erst bei Einsatz des Systems entdeckt.

- Kontext* Ein nebenläufiges System, das Betriebsmittel enthält, welche von mehreren Ablaufäden gemeinsam genutzt und nebenläufig manipuliert werden können.
- Problem* Code, der nicht nebenläufig ausgeführt werden soll, muss durch eine Sperre geschützt werden. Diese Sperre wird akquiriert, wenn die Programmkontrolle einen kritischen Abschnitt betritt, und wieder freige-

geben, wenn die Programmkontrolle diesen Abschnitt verlässt. Programmierer, die Sperren explizit akquirieren und freigeben müssen, können in der Regel jedoch nicht 100-prozentig sicherstellen, dass sie diese Sperren in jedem Pfad durch den kritischen Abschnitt auch wieder freigeben. Beispielsweise kann in C++ jeder Block aufgrund einer return-, break-, continue- oder goto-Anweisung verlassen werden – aber auch aufgrund von Ausnahmen, die außerhalb dieses Blocks behandelt werden.

Definieren Sie eine Wächterklasse (Guard Class), deren Konstruktor automatisch eine Sperre akquiriert, wenn die Programmkontrolle einen kritischen Bereich betritt, und deren Destruktor die Sperre automatisch freigibt, wenn die Programmkontrolle diesen Block wieder verlässt. Instanzieren Sie die Wächterklasse in denjenigen Methoden oder Blöcken, welche die kritischen Abschnitte enthalten, um so die zum Schutz dieser Abschnitte benötigten Sperren zu akquirieren bzw. freizugeben.

Lösung

Die Implementierung des Scoped-Locking-Idioms ist sehr einfach.

Implementierung

Definieren Sie eine Wächterklasse, die eine bestimmte Art von Sperre in ihrem Konstruktor akquiriert und in ihrem Destruktor freigibt. Der Konstruktor der Wächterklasse speichert einen Zeiger oder eine Referenz auf die Sperre und akquiriert diese anschließend. Der Destruktor der Wächterklasse verwendet den zwischengespeicherten Zeiger oder die Referenz zur Freigabe dieser Sperre.

Aktivität 1

✘ Die folgende Klasse skizziert einen »Wächter«, der auf Basis der Thread_Mutex-Wrapper-Facade (53) entworfen ist:

```
class Thread_Mutex_Guard {
public:
    // Speichere einen Zeiger auf die Sperre und
    // akquiriere sie.
    Thread_Mutex_Guard (Thread_Mutex &lock)
        : lock_ (&lock), owner_ (false) {
        lock_->acquire ();

        // Wird auf true gesetzt, wenn <acquire>
        // erfolgreich war.
        owner_ = true;
    }

    // Gib die Sperre wieder frei.
    ~Thread_Mutex_Guard () {
        // Gib die Sperre nur frei, wenn sie auch
        // tatsaechlich akquiriert wurde.
        if (owner_) lock_->release ();
    }
}
```

```
private:
    Thread_Mutex *lock_; // Zeiger auf die Sperre.
    bool owner_; // Ist <lock_> akquiriert?

    // Verbiere Kopieren und Zuweisungen.
    Thread_Mutex_Guard (const Thread_Mutex_Guard &);
    void operator= (const Thread_Mutex_Guard &);
};
```

✘

In der Implementierung einer Wächterklasse sollten wir stets einen Zeiger auf eine Sperre anstelle eines Sperrobjekts verwenden, um so das Kopieren oder Zuweisen einer Sperre zu verhindern. Dies würde nur zu Fehlern führen, wie wir im Abschnitt *Implementierung* des Musters Wrapper Facade (53) bereits erörtert haben.

Außerdem ist es sinnvoll, ein Flag hinzuzufügen, z.B. das `owner_`-Flag im obigen `Thread_Mutex_Guard`-Beispiel. Dieses zeigt an, ob ein Wächter die Sperre erfolgreich akquiriert hat. Das Flag kann aber auch Fehler anzeigen, die aufgrund von Problemen mit der Initialisierungsreihenfolge statischer/globaler Sperren auftreten können [LGS99]. Durch Überprüfung dieses Flags im Destruktor des Wächters können wir Laufzeitfehler vermeiden, die auftreten würden, falls der Wächter die Sperre nicht erfolgreich akquirieren konnte.

Aktivität 2

»Verknüpfen« *Sie jeden kritischen Abschnitt mit dem Gültigkeitsbereich und der Lebenszeit eines Wächterobjekts.* Um einen kritischen Abschnitt vor nebenläufigem Zugriff zu schützen, kapseln Sie diesen in einem eigenen Block – wenn dies nicht schon getan ist – und erzeugen Sie als erste Anweisung innerhalb dieses Blocks ein Wächterobjekt auf dem Laufzeitkellerspeicher. Der Konstruktor der Wächterklasse akquiriert die benötigte Sperre dann automatisch. Beim Verlassen des Blocks mit dem kritischen Abschnitt wird automatisch der Destruktor des Wächters aufgerufen – und gibt die Sperre frei. Aufgrund der Semantik von C++-Destruktoren werden »bewachte« Sperren sogar dann freigegeben, wenn innerhalb des kritischen Abschnitts C++-Ausnahmen geworfen werden.

✘ Das Scoped-Locking-Idiom löst die oben beschriebenen Probleme mit der `Hit_Counter`-Klasse in unserem nebenläufigen Webserver:

```
class Hit_Counter {
public:
    // Erhöhe den Trefferzaehler für die URL <path>.
    bool increment (const string &path) {
        // Benutze Scoped Locking, um den <lock_>
        // automatisch zu akquirieren und freizugeben.
        Thread_Mutex_Guard guard (lock_);
        Table_Entry *entry = lookup_or_create (path);
```

```

    if (entry == 0)
        return false;
        // Destruktor gibt <lock_> wieder frei
    else { // Erhoehe den Zahler fuer die URL <path>.
        entry->increment_hit_count ();
        return false;
        // Destruktor gibt <lock_> wieder frei
    }
}

// Weitere oeffentliche Methoden...

private:
    // Serialisiere den Zugriff auf den kritischen
    // Abschnitt.
    Thread_Mutex lock_;
};

```

In dieser Lösung stellt der Wächter sicher, dass die Sperre automatisch akquiriert bzw. freigegeben wird, wenn die Programmkontrolle die `increment()`-Methode betritt bzw. verlässt. ✘

Explicit Accessors. Die im Abschnitt *Implementierung* beschriebene `Thread_Mutex_Guard`-Schnittstelle hat einen Nachteil: Es ist nicht möglich, die Sperre wieder freizugeben, ohne explizit die sie umgebende Methode oder den sie umgebenden Block zu verlassen.

Varianten

✘ Der folgende Code veranschaulicht eine Situation, in der die Sperre zweimal freigegeben werden könnte – abhängig davon, ob die Bedingung in der `if`-Anweisung erfüllt ist:

```

{
    Thread_Mutex_Guard guard (lock);
    // Tue etwas...
    if (/* eine bestimmte Bedingung erfuellt ist */)
        lock->release ()
    // Tue noch was...

    // Verlasse den Block, was die Sperre erneut freigibt.
}

```

✘

Um diesen Fehler zu verhindern, sollten wir nicht direkt auf die Sperre zugreifen. Stattdessen sollten wir explizite Zugriffsmethoden (*Explicit Accessors*) auf die Sperre verwenden, die wir in deren Wächterklasse definieren.

✘ Wir revidieren die `Thread_Mutex_Guard`-Klasse wie folgt:

```
class Thread_Mutex_Guard {
public:
    // Speichere einen Zeiger auf die Sperre und akquiriere sie.
    Thread_Mutex_Guard (Thread_Mutex &lock)
        : lock_ (&lock), owner_ (false) {
        acquire ();
    }

    void acquire () {
        lock_->acquire ();
        // Wird auf true gesetzt, wenn <acquire> erfolgreich war.
        owner_ = true;
    }

    void release () {
        // Gib die Sperre nur frei, wenn sie auch tatsaechlich
        // akquiriert und noch nicht freigegeben wurde.
        if (owner_) {
            owner_ = false;
            lock_->release ();
        }

        // Gib die Sperre wieder frei.
        ~Thread_Mutex_Guard () { release (); }
private:
    Thread_Mutex *lock_; // Zeiger auf die Sperre.
    bool owner_; // Ist <lock_> akquiriert?

    // Verbiere Kopieren und Zuweisungen...
};
```

✘

Die Zugriffsmethoden `acquire()` und `release()` »merken« sich, ob die Sperre schon freigegeben ist. Wenn ja, wird die Sperre im Destruktor der `guard`-Funktion nicht noch einmal freigegeben.

✘ Mit der aktualisierten `Thread_Mutex_Guard`-Klasse funktioniert unser Code korrekt:

```
{
    Thread_Mutex_Guard guard (lock);
    // Tue etwas...
    if (/* eine bestimmte Bedingung erfuellt ist*/)
        guard.release ();
    // Tue noch was...
    // Verlasse den Block, was die Sperre erneut freigibt.
}
```

✘

Strategized Scoped Locking. Die Definition eines separaten Wächters für jeden Sperrentyp ist lästig, fehleranfällig und unnötig. Es erhöht zudem den Speicherplatzverbrauch von Anwendungen und Kompo-

nen. Eine gängige Variante des Scoped-Locking-Idioms beruht daher auf dem Muster Strategized Locking (369), bei dessen Implementierung wir entweder parametrisierte Typen oder Polymorphismus einsetzen können.

Booch Components. Die Booch Components [BV93] waren eine der ersten C++-Klassenbibliotheken, die zur Serialisierung nebenläufiger C++-Programme das Scoped-Locking-Idiom benutzt haben.

Anwendungsbeispiele

ACE [Sch97]. Das Scoped-Locking-Idiom wird an vielen Stellen innerhalb des Rahmenwerks ADAPTIVE Communication Environment verwendet. ACE definiert eine ACE_Guard-Implementierung, welche der Thread_Mutex_Guard-Klasse sehr ähnlich ist, die wir in den Abschnitten *Implementierung* und *Varianten* beschrieben haben.

Threads.h++. Die Bibliothek Threads.h++ von Rogue Wave definiert einen Satz verschiedener Wächterklassen, die entsprechend der Scoped-Locking-Entwürfe in ACE modelliert wurden.

Java definiert eine Programmierspracheneigenschaft namens Synchronized Block, die das Scoped-Locking-Idiom in dieser Sprache implementiert. Java-Compiler generieren dazu einen Block von Bytecode-Befehlen, der von einem `monitorenter`- und `monitorexit`-Befehl eingeklammert wird. Um sicherzustellen, dass die Sperre auch bei *jedem* »Ausstieg« aus diesem Block freigegeben wird, generieren die Compiler zusätzliche Funktionalität zur Behandlung aller Ausnahmen, die im Synchronized Block geworfen werden können [Eng99].

Das Scoped-Locking-Idiom bietet folgende **Vorteile**:

Auswirkungen

Erhöhte Robustheit. Durch Verwendung dieses Idioms werden Sperren automatisch akquiriert und wieder freigegeben, wenn die Programmkontrolle kritische Abschnitte betritt und verlässt, die durch C++-Methoden und Blöcke gekapselt werden. Dies erhöht die Robustheit nebenläufiger Anwendungen, da das Idiom gängige Programmierfehler im Zusammenhang mit Synchronisation und Nebenläufigkeit eliminiert.

Allerdings gibt es zwei **Nachteile** bei der Verwendung des Scoped-Locking-Idioms für nebenläufige Anwendungen und Komponenten:

Potenzial für Verklemmung bei rekursiven Methodenaufrufen. Wenn eine Methode, die das Scoped-Locking-Idiom verwendet, sich selbst rekursiv aufruft, so tritt eine »Selbstverklemmung« ein, wenn die Sperre kein »rekursiver« Mutex ist. Das Thread-Safe-Interface-Muster (383) beschreibt jedoch eine Technik, die dieses Problem vermeidet.

Das Muster gewährleistet, dass nur Schnittstellenmethoden einer Komponente das Scoped-Locking-Muster einsetzen, ihre internen Implementierungsmethoden aber nicht.

Einschränkungen im Zusammenhang sprachenspezifischer Semantik. Das Scoped-Locking-Idiom basiert auf einer C++-Spracheigenschaft und wird darum nicht bei betriebssystemspezifischen Aufrufen eingesetzt. Es ist somit nicht sichergestellt, dass Sperren automatisch wieder freigegeben werden, wenn Ablaufäden oder Prozesse abbrechen oder Systemaufrufe aus einem kritischen Abschnitt »herausspringen«. Die Sperren werden auch nicht freigegeben, wenn Systemaufrufe die Standard-C-Funktion `longjmp()` aufrufen, da diese Funktion die Destruktoren von C++-Objekten nicht aufruft, wenn der Laufzeitkeller »zurückspult«.

✘ Die folgende Änderung der `increment()`-Funktion bewirkt, dass das Scoped-Locking-Idiom nicht mehr funktioniert:

```

Thread_Mutex_Guard guard (&lock_);
Table_Entry *entry = lookup_or_create (path);
if (entry == 0)
    // Etwas ging schief, verlasse den Ablaufäden.
    thread_exit ();
    // Der Destruktor wird nicht aufgerufen,
    // <lock_> wird somit nicht freigegeben!

```

Aus diesem Grund ist es nicht sinnvoll, mitten in der Ausführung einer Komponente aus einem Ablaufäden oder Prozess »auszusteigen«. Stattdessen sollte im Fehlerfall eine Ausnahmebehandlung durchgeführt oder ein Fehler-Propagierungs-Muster eingesetzt werden [Mue96].

Übermäßig viele Warnungen der Compiler. Das Scoped-Locking-Idiom definiert ein Wächterobjekt, welches in dem Block, in dem es definiert ist, nicht explizit verwendet wird. Die durch den Wächter definierte Sperre wird nur implizit in dessen Konstruktor akquiriert und in dessen Destruktor freigegeben. Leider geben einige C++-Compiler unnötige »Statement has no effect«-Warnungen aus, wenn wir Wächter innerhalb eines Blocks definieren, diese aber nicht explizit in diesem Block verwenden. Bestenfalls sind diese Warnungen störend – schlimmstenfalls animieren sie Entwickler, bestimmte Compiler-Warnungen auszuschalten, die auf Probleme mit ihrem Code hinweisen. Dieses Problem können wir jedoch effektiv lösen, indem wir ein Makro definieren, das diese Warnungen eliminiert, ohne zusätzlichen Code zu generieren.

✘ Folgendes Makro ist in ACE [Sch97] definiert:

```
#define UNUSED_ARG(arg) { if (&arg) /* null */; }
```

Dieses Makro können wir direkt nach einem Wächter in den Code einfügen, was zahlreiche C++-Compiler davon abhält, falsche oder irreführende Warnungen zu generieren:

```
{ // Neuer Block.  
    Thread_Mutex_Guard guard (lock_);  
    UNUSED_ARG (guard);  
    // ... } ✘
```

Das Scoped-Locking-Idiom ist eine Spezialisierung des viel allgemeineren C++-Idioms Resource-Acquisition-is-Initialization [Str00] und des Idioms Execute-Around Object [Hen00]. In diesen Idiomen akquiriert ein Konstruktor ein Betriebsmittel, wann immer ein neuer Block betreten wird. Ein Destruktor gibt das Betriebsmittel wieder frei, wenn dieser Block verlassen wird. Wenn wir diese Idiome auf die Programmierung nebenläufiger Systeme anwenden, ist das akquirierte und wieder freigegebene Betriebsmittel eine Sperre.

Siehe auch

Wir bedanken uns vielmals bei Brad Appleton für seine umfangreichen Kommentare zu einer früheren Version dieses Idioms.

Danksagungen