Synchronizace

scoped / strategized locking
thread-safe interface
double-checked locking

Synchronizovaný blok (scoped locking)

```
struct SynchronizedCounter {
public:

    unsigned increment(
        lock_.acquire()
        unsigned ret;
        if (value_ == U
            ret = value
            lock_.relea
            return ret;
        }
        ret = ++value_;
        lock_.release();
        return ret;
    }

private:
    Lock lock_;
    unsigned value_;
};
```

ne svelte pismo na tmavem podklade !

zdrojový kód ne jako obrázek

→

```
struct SynchronizedCounter {
public:

    unsigned increment() {
        LockGuard _(lock_);
        if (value_ == UINT_MAX)
            return value_;
        return ++value_;
    }

private:
    Lock lock_;
    unsigned value_;
};
```

Synchronizovaný blok (scoped locking)

- spíš ustálený obrat než návrhový vzor
- cíl: mít zámek zamčený tehdy a pouze tehdy, když probíhá kritická sekce, na kterou je vázán, aniž by na to autor kritické sekce musel neustále myslet
- myšlenka: zámek je vázán na blok kódu obsahující krit. sekci - na jeho začátku se zamkne a uvolní se automaticky právě tehdy, když blok opustíme
- implementace: různá dle jazyka, ale zpravidla dost přímočará - hodně jazyků na to má dedikovaný mechanismus

Synchronizovaný blok (scoped locking)

C++

(RAII)

```
class Lock;

class LockGuard {
public:
    LockGuard(Lock& lock) :lock_(&lock) {
        lock_->acquire();
    }

    ~LockGuard() {
        lock_->release();
    }
private:
    Lock* lock_;

    //zakážeme kopirování
    LockGuard(const LockGuard&) = delete;
    LockGuard& operator=(const LockGuard&) = delete;
};
```

```
struct SynchronizedCounter {
public:

    unsigned increment() {
        LockGuard _(lock_);
        if (value_ == UINT_MAX)
            return value_;
        return ++value_;
    }

private:
    Lock lock_;
    unsigned value_;
};
```

Synchronizovaný blok (scoped locking)

exception handling

```
class Lock;

class LockGuard {
public:
    LockGuard(Lock& lock) :lock_(&lock) {
        lock_->acquire();
    }

    ~LockGuard() {
        lock_->release();
    }
private:
    Lock* lock_;

    //zakážeme kopírování
    LockGuard(const LockGuard&) = delete;
    LockGuard& operator=(const LockGuard&) = delete;
};
```

```
template<typename T>
struct SynchronizedCounter {
public:

    T increment() {
        lock_.acquire();
        T ret;
        if (value_ == T::MAX_VALUE) {
            ret = value_;
            lock_.release();
            return ret;
        }
        ret = ++value_;
        lock_.release();
        return ret;
    }

private:
    Lock lock_;
    T value_;
};
```

```
template<typename T>
struct SynchronizedCounter {
public:

    T increment() {
        LockGuard _(lock_);
        if (value_ == T::MAX_VALUE)
            return value_;
        return ++value_;
    }

private:
    Lock lock_;
    T value_;
};
```

Synchronizovaný blok (scoped locking)

C#

```
class SynchronizedCounter
{
    public uint Increment()
    {
        try
        {
            lock_.Acquire();
            if (_value == uint.MaxValue)
                return _value;
            return ++_value;
        }
        finally
        {
            lock_.Release();
        }
    }
    private Lock lock_;
    private uint _value;
}
```

```
abstract class Lock
{
    public abstract void Acquire();
    public abstract void Release();
}

struct LockGuard : IDisposable
{
    public LockGuard(Lock lockInstance)
    {
        _lock = lockInstance;
        _lock.Acquire();
    }

    public void Dispose() => _lock.Release();

    private readonly Lock _lock;
}
```

```
class SynchronizedCounter
{
    public uint Increment()
    {
        using(LockGuard _ = new LockGuard(_lock))
        {
            if (_value == uint.MaxValue)
                return _value;
            return ++_value;
        }
    }

    private Lock _lock;
    private uint _value;
}
```

```
class SynchronizedCounter
{
    public uint Increment()
    {
        lock(lock_)
        {
            if (_value == uint.MaxValue)
                return _value;
            return ++_value;
        }
    }
    private object lock_ = new object();
    private uint _value;
}
```

```
class SynchronizedCounter
{

    public uint Increment()
    {
        using LockGuard _ = new LockGuard(_lock);
        if (_value == uint.MaxValue)
            return _value;
        return ++_value;
    }

    private Lock _lock;
    private uint _value;
}
```

Synchronizovaný blok (scoped locking)

Java

```
class SynchronizedCounter{

    public int increment(){
        try{
            lock.acquire();
            if(value == Integer.MAX_VALUE)
                return value;
            return ++value;
        }
        finally{
            lock.release();
        }
    }

    private int value;
    private Lock lock;
}
```

```
abstract class Lock{
    public abstract void acquire();
    public abstract void release();
}

class LockGuard implements AutoCloseable{

    public LockGuard(Lock lock){
        this.lock = lock;
        this.lock.acquire();
    }

    @Override public void close() {
        lock.release();
    }
    private final Lock lock;
}
```

```
class SynchronizedCounter{

    public int increment(){
        try(LockGuard __ = new LockGuard(lock)){
            if(value == Integer.MAX_VALUE)
                return value;
            return ++value;
        }
    }

    private int value;
    private Lock lock;
}
```

```
class SynchronizedCounter{

    public int increment(){
        synchronized (lock){
            if(value == Integer.MAX_VALUE)
                return value;
            return ++value;
        }
    }

    private int value;
    private Object lock = new Object();
}
```

```
class SynchronizedCounter{

    public synchronized int increment(){
        if(value == Integer.MAX_VALUE)
            return value;
        return ++value;
    }

    private int value;
}
```

Synchronizovaný blok (scoped locking)

výhody:

- po opuštění bloku se zámek vždy automaticky uvolní
  - možnost soustředit se na ostatní věci
  - prevence deadlocku
- manipulace se zámkem napsána jednou v Guard třídě, pak se provádí implicitně
  - stručnost
  - jednodušší údržba
  - DRY

Synchronizovaný blok (scoped locking)

někdy je ale potřeba zámek uvolnit uvnitř bloku

```
class LockGuard;

class Lock {
protected:
    virtual void acquire() = 0;
    virtual void release() = 0;

    friend LockGuard;
};

class LockGuard {
public:
    LockGuard(Lock& lock) :lock_(&lock) {
        lock_->acquire();
    }

    ~LockGuard() {
        release();
    }

    void release() {
        if (lock_)
            lock_->release();
        lock_ = nullptr;
    }

private:
    Lock* lock_;

    //zakážeme kopírování
    LockGuard(const LockGuard&) = delete;
    LockGuard& operator=(const LockGuard&) = delete;
};
```

- kdybychom nepřistupovali skrze Guard, uvolnil by se dvojitě - chyba / undefined
- zobecnění: libovolné zdroje hlídané zámkem mít jako jeho privátní položky přístupné jedině přes Guard -> vynucení že nebudou manipulovány když zámek není držen

```
Lock *lock;

{LockGuard guard(*lock);

    //...
    if (something()) {
        guard.release();
        do_something();
        return;
    }
    //...
}
```

Synchronizovaný blok (scoped locking)

na co si dát pozor - C++:

- volání destruktorů zajištěné pouze sémantikou jazyka
- při volání funkce, ze které se nevrací (př. exit, exec,...), nebo Céčkovské longjmp k uvolnění zámku nedojde
- otravné stížnosti kompilátoru na nepoužívanou proměnnou

Synchronizovaný blok (scoped locking)

na co si dát pozor - Java/C#:

- antipattern: zámek na *this* - vidí i náhodné objekty zvenku - klidně taky mohou zamykat -> nepředvídatelný deadlock
- řešení: vytvořit si privátní objekt
- obdobně pro statické metody a *typeof(this)*

```
class SynchronizedCounter{

    public synchronized int increment(){
        if(value == Integer.MAX_VALUE)
            return value;
        return ++value;
    }

    private int value;
}
```

```
class SynchronizedCounter
{

    public uint Increment()
    {
        lock (this)
        {
            if (_value == uint.MaxValue)
                return _value;
            return ++_value;
        }
    }

    private uint _value;
}
```

```
class SynchronizedCounter{

    public int increment(){
        synchronized (lock){
            if(value == Integer.MAX_VALUE)
                return value;
            return ++value;
        }
    }

    private int value;
    private Object lock = new Object();
}
```

Synchronizovaný blok (scoped locking)

s čím lze kombinovat:

- Strategized Locking: jediná implementace platná pro všechny možné typy zámků - ještě víc DRY
- Thread-safe interface: ještě větší odolnost proti self-deadlocku
- prakticky jakýkoliv reálný scénář kde je potřeba držet nějaký zámek (reálně používáno skoro všude)

Zamykání dle strategie (strategized locking)

- problém: jeden kus funkcionality, několik různých prostředí (co do interakce vláken) ve kterých má běžet
- řešení: pro každé prostředí naimplementovat funkcionalitu znovu (velmi nepraktické) X použít návrhový vzor Strategie - do ní zamykání vyabstrahovat
- samotný objekt zámku i strategii pro jeho používání pro stručnost zkoncentrujeme do jediného objektu, kterým je naše funkcionalita parametrizována

Zamykání dle strategie (strategized locking)

- strategie předána např. polymorfně nebo jako typový parametr (silnější)
- všechny typy zámků se musí shodnout na rozhraní - mohou se hodit vzory Adaptér, Wrapperová fasáda, ...
- vhodné zkombinovat s blokovým zamykáním - naimplementovat obecný Guard rovněž parametrizovaný zámkem (= strategií)

```
template<typename TLock>
class Guard {
public:
    Guard(TLock& lock) :lock_(&lock) {
        lock_->acquire();
    }

    ~Guard() {
        lock_->release();
    }

private:
    std::remove_reference_t<TLock> *lock_;

    //zakážeme kopírování
    Guard(const Guard&) = delete;
    Guard& operator=(const Guard&) = delete;
};
```

```
template<typename TLock>
struct SynchronizedCounter {
public:
    SynchronizedCounter(TLock lock):lock_(lock){}

    unsigned increment() {
        Guard<TLock> _(lock_);
        if (value_ == UINT_MAX)
            return value_;
        return ++value_;
    }

private:
    TLock lock_;
    unsigned value_;
};
```

Zamykání dle strategie (strategized locking)

na co si dát pozor:

- mít na zámkový typ co nejméně požadavků - vyhýbat se dvojitému zamčení apod. (jinak pro některé parametry self-deadlock) -> viz např. *Thread-Safe Interface* pattern
- může se stát, že jsme v čistě single thread prostředí - šikovný trik: lock s prázdnou implementací - dodat jako parametr a zbavíme se lockovacího overheadu

```
class NullLock {
public:
    void acquire(){}

    void release(){}
};
```

Zamykání dle strategie (strategized locking)

výhody:

- jako u obecného použití patternu Strategie
- flexibilita a možnost přizpůsobení (možné spojit libovolnou komponentu a strategii jak zrovna člověk potřebuje)
- jednodušší údržba - DRY - každá věc napsána jen na jednom místě
- výsledek obecnější, znovupoužitelnější
- zamykací strategie a funkcionalita komponenty nuceně ortogonální -> větší přehlednost

Zamykání dle strategie (strategized locking)

na co si dát pozor:

- zamykací strategie a funkcionalita komponenty nuceně ortogonální - někdy se nehodí
- při compile-time parametru rozhlašujeme do světa, jaká konkrétní zamykací strategie bude použita
- někdy není třeba (v praxi bude použita vážně jen jediná zamykací strategie) - zbytečná komplexita a prostor pro vznik chyb (např. nezkušený programátor parametrizuje čím nemá -> neočekávané chování)

Thread-safe interface

- situace: komponenta v paralelním prostředí, jejíž metody se vzájemně volají
- cíl: snížit zamykací overhead, zamezit self-deadlockům
- myšlenka: zamknout pouze jednou na hranici komponenty, jakmile jsme uvnitř, už se o zámek nestarat
- možné řešení:
  - privátní metody... v nich veškerá logika komponenty; věří, že při jejich volání je již zámek držen
  - veřejné metody... pouze zajistí zámek a zavolá privátní metodu, volána pouze zvenku komponenty, nikdy ne zevnitř (-> self-deadlock)

Thread-safe interface

```cpp
template<typename TLock>
struct SynchronizedCounter {
public:
    SynchronizedCounter(TLock lock):lock_(lock){}

    unsigned increment_and_get() {
        Guard<TLock>_(lock_);
        increment();
        return value_;
    }

    void increment() {
        Guard<TLock>_(lock_);
        if (value_ != UINT_MAX)
            ++value_;
    }

private:
    TLock lock_;
    unsigned value_;
};
```

```cpp
template<typename TLock>
struct SynchronizedCounter {
public:
    SynchronizedCounter(TLock lock):lock_(lock){}

    unsigned increment_and_get() {
        Guard<TLock>_(lock_);
        return increment_and_get_impl();
    }

    void increment() {
        Guard<TLock>_(lock_);
        return increment_impl();
    }

private:
    TLock lock_;
    unsigned value_;

    unsigned increment_and_get_impl() {
        increment_impl();
        return value_;
    }
    void increment_impl() {
        if (value_ != UINT_MAX)
            ++value_;
    }
};
```

Thread-safe interface

- nevýhody:
  - metody psány pokaždé dvakrát
  - volání veřejné funkce z privátní - chyba X kompilátor ji nepodchytí
- lepší řešení: při psaní komponenty ignorovat synchronizaci, zamykání soustředíme do *Dekorátoru*, kterým komponentu příp. obalíme
  - např. java.util.Collections.synchronizedMap()
  - větší modularita, nemožné volat zamykací metodu z ‘privátní’

```
interface Counter{
    public void increment();

    public int incrementAndGet();
}

class SynchronizedCounter implements Counter{
    public SynchronizedCounter(Counter impl){
        this.impl = impl;
    }

    @Override public void increment() {
        synchronized (lock) {
            impl.increment();
        }
    }
    @Override public int incrementAndGet() {
        synchronized (lock){
            return impl.incrementAndGet();
        }
    }

    private final Counter impl;
    private final Object lock = new Object();
}
```

Thread-safe interface

- implementace skrze polymorfismus - objekt navíc - zbytečný overhead
- pořád musíme psát synchronizační *Dekorátor* pro každé rozhranní zvlášť
- v C++ se dá přetížením operator->() napsat proxy synchronizující obecně libovolný objekt
- někdy chceme jedním zámkem synchronizovat přístup k celé síti objektů - synchronizovaná *Fasáda*

```
interface Counter{
    public void increment();

    public int incrementAndGet();
}

class SynchronizedCounter implements Counter{
    public SynchronizedCounter(Counter impl){
        this.impl = impl;
    }

    @Override public void increment() {
        synchronized (lock) {
            impl.increment();
        }
    }
    @Override public int incrementAndGet() {
        synchronized (lock){
            return impl.incrementAndGet();
        }
    }

    private final Counter impl;
    private final Object lock = new Object();
}
```

Thread-safe interface

na co si dát pozor:

- self-deadlock pořád možný (z privátní metody voláme jiný objekt - ten obratem volá naši veřejnou metodu)
- znemožňuje zamykání jemněji než vždy pro celou metodu - ne dokonale optimální (zbytečné zamykání apod.)

Double-Checked locking optimization

- situace: kritická sekce, která má proběhnout nanejvýš jednou za běh programu, uvnitř procedury, jež je volána velmi často (př. u *Singletonu*)
- chceme, aby zamykání proběhlo vážně jen jednou a nepřineslo overhead ke každému volání procedury

Není thread-safe

```
class Singleton {
public:
    static Singleton *get_instance() {
        if (!instance_)
            instance_ = new Singleton();
        return instance_;
    }

private:
    static Singleton* instance_;
    static Lock lock_;
};
```

Zbytečný overhead

```
class Singleton {
public:
    static Singleton *get_instance() {
        Guard<Lock>_(lock_);
        if (!instance_)
            instance_ = new Singleton();
        return instance_;
    }

private:
    static Singleton* instance_;
    static Lock lock_;
};
```

Double-Checked locking optimization

1. nápad

```
class Singleton {
public:
    static Singleton *get_instance() {
        if (!instance_) {
            Guard<Lock>_(lock_);
            instance_ = new Singleton();
        }
        return instance_;
    }

private:
    static Singleton* instance_;
    static Lock lock_;
};
```

race condition

Double-Checked locking optimization

víceméně funkční řešení

```
class Singleton {
public:
    static Singleton *get_instance() {
        if (!instance_) {
            Guard<Lock>_(lock_);
            if(!instance_)
                instance_ = new Singleton();
        }
        return instance_;
    }

private:
    static Singleton* instance_;
    static Lock lock_;
};
```

testujeme podruhé

Double-Checked locking optimization

problémy - C++:

- kompilátor může prohodit pořadí instrukcí, odoptimalizovat 2. test pryč, ... -> řešení: volatile proměnná nebo bariéra
- někdy CPU cache nejsou transparentní (Itanium, ...) - potřeba specielní bariéra
- přiřazení pointeru nemusí být atomické -> race condition SEGFAULT
- -> špatná portabilita

```
class Singleton {
public:
    static Singleton *get_instance() {
        if (!instance_) {
            Guard<Lock>_(lock_);
            if(!instance_)
                instance_ = new Singleton();
        }
        return instance_;
    }

private:
    static Singleton* instance_;
    static Lock lock_;
};
```

testujeme podruhé

Double-Checked locking optimization

Java/C#:

- mají standardně volatile proměnné (Java od v5.0)
- většinou není vůbec potřeba řešit
  - pořadí inicializace statických proměnných dobře definované, probíhá líně až při 1. interakci s třídou
  - narozdíl od C++
    - nedefinované pořadí inicializace napříč kompilačními jednotkami
    - všechno globální se inicializuje vždy a hned při spuštění
    - -> inicializovat explicitně líně je jediné řešení

Class loader vyhodnotí líně

```
class Singleton
{
    public static Singleton Instance { get; } = new();
}
```

Načtena až voláním Singleton.Instance

```
class Singleton
{
    //some other stuff...

    private static class InstanceContainer
    {
        public static Singleton Instance { get; } = new();
    }
    public static Singleton Instance => InstanceContainer.Instance;

    //other stuff...
}
```

Double-Checked locking optimization

další varianta:

```
class Singleton
{
    private static readonly object _lock = new object();

    private static Singleton _globalInstance = null;

    [ThreadStatic]
    private static Singleton _threadLocalInstance = null;



    public static Singleton Instance { get {
        if(_threadLocalInstance == null)
            {
                lock (_lock)
                {
                    if (_globalInstance == null)
                        _globalInstance = new Singleton();
                    _threadLocalInstance = _globalInstance;
                }
            }
            return _threadLocalInstance;
    }}
}
```