Unfallfreies Software-Designs mit Threads und Mutex

…oder keine Angst vor’m bösen Dead Lock


BITTE BENUTZEN SIE DIE ANDERE KASSE
BITTE BENUTZEN SIE DIE ANDERE KASSE

Übersicht

- Warum Threads, warum Locks?
- Deadlocks
- Primitiven in C++
- Pattern
- Anti-Pattern
- Pitfalls
- Advance C++ Threading Primitives

Warum Threads?

- Nutzung mehrerer CPUs
- Nutzung mehrerer blockender APIs

Warum Locks?

- Invarianten dürfen weiterhin nicht sichtbar werden
- Exklusiver Zugriff im Fall, dass Invarianten kurzfristig verletzt sind.

Deadlocks

„Deadlock bezeichnet in der Informatik einen Zustand, bei dem eine zyklische Wartesituation zwischen mehreren Prozessen auftritt, wobei jeder beteiligte Prozess auf die Freigabe von Betriebsmitteln wartet, die ein anderer beteiligter Prozess bereits exklusiv belegt hat.“

Wikipedia

typische „Betriebsmittel"

- mutex / condition_variable
- files / sockets
- Ende eines threads (`join()`)
- queues
- oder allgemein: alles was blocken kann.

Basic C++ Thread Primitives

- std::thread
- std::mutex
- std::condition_variable

```cpp
#include <thread>
#include <iostream>

void calc_fac( int f, int& result )
{
    for ( result = 1; f; --f )
        result *= f;
}

int main()
{
    int fac3, fac5, fac7;

    std::thread calc_fac3( calc_fac, 3, std::ref( fac3 ) );
    std::thread calc_fac5( calc_fac, 5, std::ref( fac5 ) );
    calc_fac( 7, fac7 );

    calc_fac3.join();
    calc_fac5.join();

    std::cout << "3!: " << fac3 << std::endl;
    std::cout << "5!: " << fac5 << std::endl;
    std::cout << "7!: " << fac7 << std::endl;
}
```

```
#include <thread>
#include <vector>

using data = std::tuple< int, int, int >;

class indexed_list
{
public:
    void add( const data& );

    data find_by_first( int ) const;
    data find_by_second( int ) const;

private:
    std::vector< data >          data_;
    std::vector< std::size_t >   first_index_;
    std::vector< std::size_t >   second_index_;

    mutable std::mutex  mutex_;
};
```

```
template < std::size_t pos >
void update_index(
    const std::vector< data >& data,
    std::vector< std::size_t >& index );

void indexed_list::add( const data& d )
{
    std::lock_guard< std::mutex > lock( mutex_ );

    data_.push_back( d );
    update_index< 0 >( data_, first_index_ );
    update_index< 1 >( data_, second_index_ );
}

template < std::size_t pos >
data find_by_index(
    const std::vector< data >& data,
    const std::vector< std::size_t >& index,
    int key );

data indexed_list::find_by_first( int first ) const
{
    std::lock_guard< std::mutex > lock( mutex_ );

    return find_by_index< 0 >( data_, first_index_, first );
}
```

```
#include <thread>
#include <deque>

template < typename T >
class queue
{
public:
    void push( const T& );
    T pop();

private:
    std::deque< T >              queue_;

    std::mutex                   mutex_;
    std::condition_variable      not_empty_;
};
```

```
template < typename T >
void queue< T >::push( const T& element )
{
    {
        std::lock_guard< std::mutex > lock( mutex_ );

        queue_.push_back( element );
    }

    not_empty_.notify_one();
}

template < typename T >
T queue< T >::pop()
{
    std::lock_guard< std::mutex > lock( mutex_ );

    while ( queue_.empty() )
        not_empty_.wait( lock );

    const T result = queue_.front();
    queue_.pop_front();

    return result;
}
```

API-Übersicht

- `std::thread`
  - `c'tor` startet eine Funktion in einem neuen thread
  - `join()` wartet auf die Beendigung des threads
- `std::mutex`
  - `lock()` / `unlock()` werden üblicherweise nicht direkt genutzt
- `std::condition_variable`
  - `wait()` wartet auf Zustand
  - `notify_one()` / `notify_all()` zeigt Zustand an

Pattern

- Scoped Locking
- Thread-safe Interface
- Lock Hierarchies
- Queues
- Thread Pools
- Inmutable Data
- Design for Testing / Stopping

Scoped Locking

```
template < typename T >
void queue< T >::push( const T& element )
{
    {
        std::lock_guard< std::mutex > lock( mutex_ );

        queue_.push_back( element );
    }

    not_empty_.notify_one();
}
```

Thread-safe Interface

```
#include <thread>

class key;
class element;

class cache
{
public:
    const element* look_up( const key& k ) const;

    bool insert( const element&, const key& );
private:
    // ...
    mutable std::mutex mutex_;
};
```

```
const element* cache::look_up( const key& k ) const
{
    std::lock_guard< std::mutex > lock( mutex_ );

    const element* result = nullptr;
    //...
    return result;
}

bool cache::insert( const element& e, const key& k )
{
    std::lock_guard< std::mutex > lock( mutex_ );

    const element* const found = look_up( k );

    if ( found )
        return false;

    //...
    return true;
}
```

Double Lock

```
const element* cache::look_up( const key& k ) const
{
    std::lock_guard< std::mutex > lock( mutex_ );

    const element* result = nullptr;
    //...
    return result;
}

bool cache::insert( const element& e, const key& k )
{
    std::lock_guard< std::mutex > lock( mutex_ );

    const element* const found = look_up( k );

    if ( found )
        return false;

    //...
    return true;
}
```

```
const element* cache::look_up( const key& k ) const
{
    std::lock_guard< std::mutex > lock( mutex_ );

    return look_up_impl( k );
}

bool cache::insert_impl( const element& e, const key& k )
{
    const element* const found = look_up_impl( k );

    if ( found )
        return false;

    //...
    return true;
}

bool cache::insert( const element& e, const key& k )
{
    std::lock_guard< std::mutex > lock( mutex_ );

    return insert_impl( e, k );
}
```

Lock Hierarchies

- Zwei Locks, die gleichzeitig gelocked werden, werden immer in der gleichen Reihenfolge gelocked.
- Dadurch werden dead locks unmöglich.
- Dazu muss Code aber wissen, welche Locks bereits gelocked wurden und welcher Code welche Locks lockt.
- Letzteres ist meist nicht trivial (observer pattern, callbacks etc.).

Queues

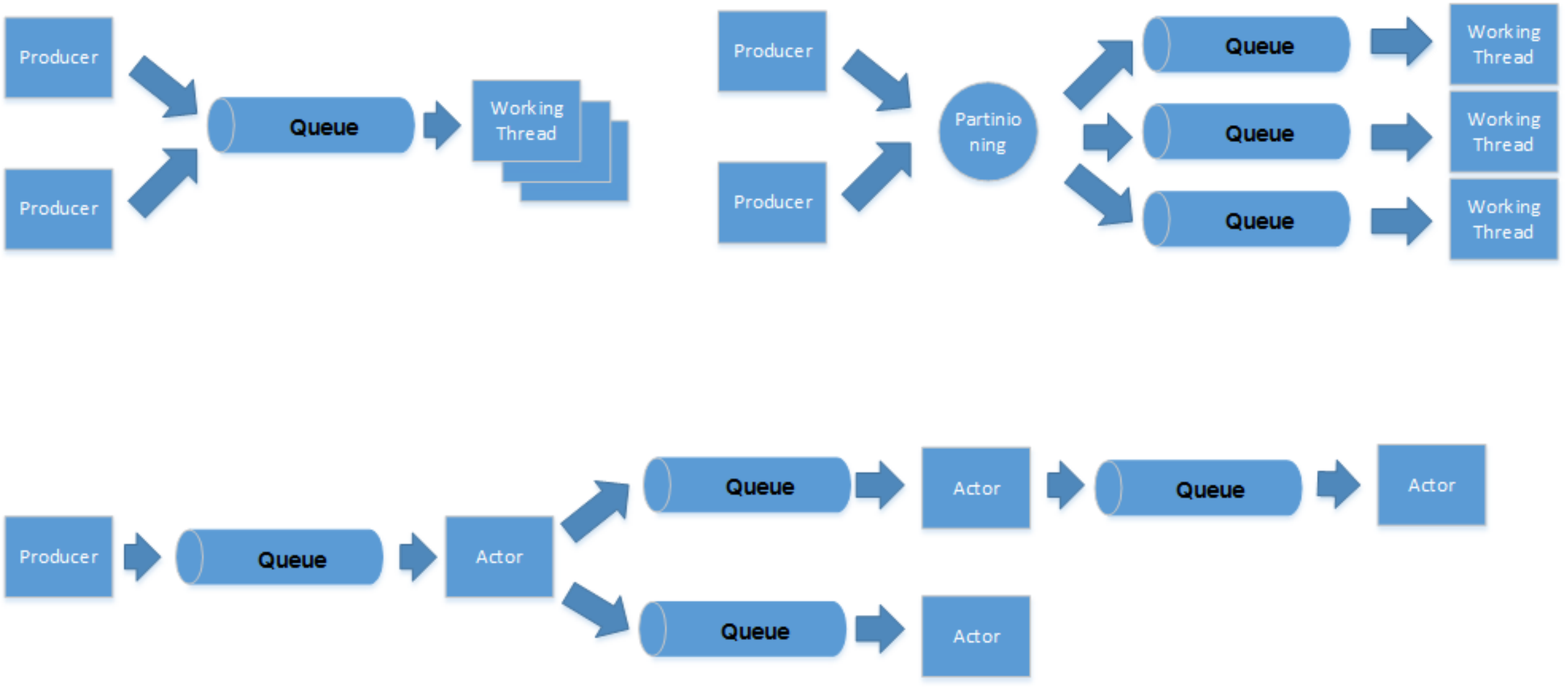
- entkoppeln producer und consumer
- kann Latenz erhöhen
- kann Latenz verringern
- Interface: `push(const T&) / T pop()`

Thread Pools

- Mehrere threads; Gleiche Funktion.
- Es spielt keine Rolle, welcher Thread eine Arbeit erledigt.
- threads suchen sich die Arbeit; nicht die Arbeit sucht sich einen thread
- Pool-Größe wird üblicherweise angepasst (Anzahl CPUs, Festplatten, etc.)

möglicher Einsatz

- Worker-Queue (Thread Pool mit Queue)
  - Working Horse for CPU bound tasks
  - Möglichkeit der Priorisierung
- Zugriff auf limitierte Ressourcen
  - Ein thread pro Ressource (Log-File, Datenbank-Verbindung)


Producer
Producer
Queue
Working Thread
Producer
Producer
Partinio ning
Queue
Queue
Queue
Working Thread
Working Thread
Working Thread
Producer
Queue
Actor
Queue
Queue
Actor
Actor

Inmutable Data

- zero costs / still free lunch
- copy on write can help
- „Immuable“ bezieht sich auf Sichtbarkeit

Design for Testing

- thread starten ist einfach
- thread stoppen ist nicht einfach
  - worauf kann ein Thread alles warten?
  - wie kann ich ihn da raus kegeln?
- Kann ich eine Aggregation von threads, queues etc. einfach stoppen?

Anti-Pattern

- Recursive Locks
- Delay
- Synchronization with / Overuse of -Priorities
- Blocking in Critical Sections
- Use of volatile
- Kill
- None Atomic Functions
- Mutex zum Warten nutzen

Pitfalls

- False Sharing
- Priority Inversion
- Don’t expect the Scheduler to be fair!
- Lock Contention / Lock Granularity

Advance C++ Threading Primitives

- std::shared_lock
- std::once_flag
- std::promise / std::future
- std::atomic
- thread_local

Danke!