Modernes C++

Sharing data between threads

Adrian Ziessler

5. Juni 2012

table of contents

race condition

Ein kritischer Wettlauf, auch Wettlaufsituation (engl. *Race Condition*) ist in der Programmierung eine Konstellation, in der das Ergebnis einer Operation vom zeitlichen Verhalten bestimmter Einzeloperationen abhängt.

example

Zwei gleichzeitig laufende Systeme wollen denselben Wert erhöhen.

Zeitpunkt	System A	Gespeicherter Wert	System B
1	Einlesen des Wertes: Wert = 1	1	
2	Erhöhen: Wert = 2	1	
3	Speichern	2	
4		2	Einlesen des Wertes: Wert = 2
5		2	Erhöhen: Wert = 3
6		3	Speichern

Erhöhen beide Systeme den Wert, ist der erwartete neue Wert 3.

example

Zwei gleichzeitig laufende Systeme wollen denselben Wert erhöhen.

Zeitpunkt	System A	Gespeicherter Wert	System B
1	Einlesen des Wertes: Wert = 1	1	Einlesen des Wertes: Wert = 1
2	Erhöhen: Wert = 2	1	Erhöhen: Wert = 2
3	Speichern	2	Speichern

Arbeiten beide Systeme gleichzeitig und ist die Reihenfolge der Befehle unbestimmt, kann die Wettlaufsituation jedoch dazu führen, dass der tatsächlich erhaltene, neue Wert stattdessen 2 beträgt.

example

Zwei gleichzeitig laufende Systeme wollen denselben Wert erhöhen.

Zeitpunkt	System A	Gespeicherter Wert	System B
1	Sperrung (bzw. warten)	1	
2	Einlesen des Wertes: Wert = 1	1	Sperrung (bzw. warten)
3	Erhöhen: Wert = 2	1	Auf Freigabe warten
4	Speichern	2	Auf Freigabe warten
5	Aufheben der Sperrung	2	Sperrung
6		2	Einlesen des Wertes: Wert = 2
7		2	Erhöhen: Wert = 3
8		3	Speichern
9		3	Aufheben der Sperrung

Zur Vermeidung des Problems sperrt A den Zugriff auf den Wert bis zum Abschluss der Veränderung, bevor B Zugriff erhält.
Dies wird in C++ mit einem sogenannten *mutex* realisiert.

using mutex in c++

- *mutex* ist die Abkürzung von **mut**ual **ex**clusion

using mutex in c++

- *mutex* ist die Abkürzung von **mut**ual **ex**clusion

```
1  class mutex
   {
3  public:
     mutex(mutex const&)=delete;
5    mutex& operator=(mutex const&)=delete;

7    mutex();
     ~mutex();
9
     void lock();
11   void unlock();
     bool try_lock();
13 };
```

using mutex in c++

- *mutex* ist die Abkürzung von **mut**ual **ex**clusion

```
1  class mutex
   {
3  public:
     mutex(mutex const&)=delete;
5    mutex& operator=(mutex const&)=delete;

7    mutex();
     ~mutex();
9
     void lock();
11   void unlock();
     bool try_lock();
13 };
```

- std::mutex mu

using mutex in c++

- *mutex* ist die Abkürzung von **mut**ual **ex**clusion

```
1  class mutex
   {
3  public:
     mutex(mutex const&)=delete;
5    mutex& operator=(mutex const&)=delete;

7    mutex();
     ~mutex();
9
     void lock();
11   void unlock();
     bool try_lock();
13 };
```

- std::mutex mu
- *lock()* und *unlock()* von Hand setzen

using mutex in c++

- *mutex* ist die Abkürzung von **mut**ual **ex**clusion

```
1  class mutex
   {
3  public:
     mutex(mutex const&)=delete;
5    mutex& operator=(mutex const&)=delete;

7    mutex();
     ~mutex();
9
     void lock();
11   void unlock();
     bool try_lock();
13 };
```

- std::mutex mu
- *lock()* und *unlock()* von Hand setzen
- Besser: *std::lock_guard* class template nutzen
  lock() on construction, unlock on destruction

protecting a list with a mutex

```
1  #include <list>
   #include <mutex>
3  #include <algorithm>

5  std::list<int> some_list; //single global variable
   std::mutex some_mutex; //protected with a corresponding global instance of std::mutex
7
   void add_to_list(int new_value)
9  {
     std::lock_guard<std::mutex> guard(some_mutex)
11   some_lust.push_back(new_value);
   }
13 bool list_contains(int value_to_find)
   {
15   std::lock_guard<std::mutex> guard(some_mutex);
     return std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
17 }
```

protecting a list with a mutex

```
1  #include <list>
   #include <mutex>
3  #include <algorithm>

5  std::list<int> some_list; //single global variable
   std::mutex some_mutex; //protected with a corresponding global instance of std::mutex
7
   void add_to_list(int new_value)
9  {
     std::lock_guard<std::mutex> guard(some_mutex)
11   some_lust.push_back(new_value);
   }
13 bool list_contains(int value_to_find)
   {
15   std::lock_guard<std::mutex> guard(some_mutex);
     return std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
17 }
```

Das benutzen von *std::lock_guard* in *add_to_list* und *list_contains* bedeutet, dass die Zugriffe auf die Funktionen sich gegenseitig ausschließen

Accidently passing out a reference to protected data 1/2

Achtung: aufpassen bei Übergabe von Zeigern oder Referenzen an die gesicherten Daten.

```
1  class some_data
   {
3    int a;
     std::string b;
5  public:
     void do_something();
7  };

9  class data_wrapper
   {
11 private:
     some_data data;
13   std::mutex m;
   public:
15   template<typename Function>
     void process_data(Function func)
17   {
       std::lock_guard<std::mutex> l(m);
19     func(data); //pass "protected" data to user-supplied function
     }
21 };
```

Accidently passing out a reference to protected data 2/2

```
1  some_data* unprotected;

3  void malicious_function(some_data& protected_data)
   {
5    unprotected=&protected_data;
   }
7
   data_wrapper x;
9  void foo()
   {
11   x.process_data(malicious_function); //pass in a malicious function
     unprotected->do_something(); //unprotected access to protected data
13 }
```

Das Aufrufen der Funktion *func* bedeutet, dass *foo* über *malicious_function* den Schutz umgehen kann und *do_something()* ohne gesperrten mutex aufruft.

common advise

Don’t pass pointers and references to protected data outside the scope of the lock, whether by returning them from a function, storing them in externally visible memory, or passing them as arguments to user-supplied functions.

the interface to the std::stack container adapter

```
1  template<typename T, typename Container=std::deque<T> >
   class stack
3  {
   public:
5    explicit stack(const Container&);
     explicit stack(Container&& = Container());
7    template <class Alloc> explicit stack(const Alloc&);
     template <class Alloc> stack(const Container&, const Alloc&);
9    template <class Alloc> stack(Container&&, const Alloc&);
     template <class Alloc> stack(stack&&, const Alloc&);
11
     bool empty() const; //results of empty() and
13   size_t size() const; //size() can't be relied on
     T& top();
15   T const& top() const;
     void push(T const&);
17   void push(T&&);
     void pop();
19   void swap(stack&&);
   };
```

Obwohl die Rückgaben von *size()* bzw. *empty()* zur Zeit des Aufrufs korrekt sein können, können andere Threads durch *pop()* oder *push()* den Stack verändern, bevor der Thread, der den Aufruf von *size()* bzw. *empty()* eingeleitet hat, diese Information nutzen kann.

problem with a shared stack object

```
stack<int> s;
if(!s.empty())
{
  int const value=s.top(); //undefined behavior?
  s.pop();
  do_something(value);
}
```

problem with a shared stack object

```
1  stack<int> s;
   if(!s.empty())
3  {
     int const value=s.top(); //undefined behavior?
5    s.pop();
     do_something(value);
7  }
```

Thread A	Thread B
if (!s.empty())	
int const value=s.top();	
	if (!s.empty())
s.pop();	
	int const value=s.top();
do_something(value);	s.pop();
	do_something(value);

a possible ordering of operations on a stack from two threads

Thread A	Thread B
`if (!s.empty())`	
	`if (!s.empty())`
`int const value=s.top();`	
	`int const value=s.top();`
`s.pop();`	
`do_something(value);`	`s.pop();`
	`do_something(value);`

possible solutions

- Interfaceänderung
  Schützt den Aufruf von *pop()* und *top()* mit einem *mutex*.

possible solutions

- Interfaceänderung
  Schützt den Aufruf von *pop()* und *top()* mit einem *mutex*.
- Übergabe einer Referenz an eine Variable (1)

```
std::vector<int> result;
some_stack.pop(result);
```

Nachteil: benötigt Instanz vom Wertetyp des Stacks vor dem Aufruf
-> "Teuer" bzw. unmöglich
-> wichtige Restriktion: Gelagerter Typ muss zuweisbar sein

possible solutions

- Interfaceänderung
  Schützt den Aufruf von *pop()* und *top()* mit einem *mutex*.
- Übergabe einer Referenz an eine Variable (1)

```
std::vector<int> result;
some_stack.pop(result);
```

  Nachteil: benötigt Instanz vom Wertetyp des Stacks vor dem Aufruf
  -> "Teuer" bzw. unmöglich
  -> wichtige Restriktion: Gelagerter Typ muss zuweisbar sein
- Fordern eines no-throw copy- oder move Konstruktors (2)

possible solutions

- Interfaceänderung
  Schützt den Aufruf von *pop()* und *top()* mit einem *mutex*.
- Übergabe einer Referenz an eine Variable (1)

```
std::vector<int> result;
some_stack.pop(result);
```

  Nachteil: benötigt Instanz vom Wertetyp des Stacks vor dem Aufruf
  -> "Teuer" bzw. unmöglich
  -> wichtige Restriktion: Gelagerter Typ muss zuweisbar sein
- Fordern eines no-throw copy- oder move Konstruktors (2)
- Zeigerübergabe (3)
  Nachteil: erfordert Mittel für Verwaltung der Speichergröße

possible solutions

- Interfaceänderung
  Schützt den Aufruf von *pop()* und *top()* mit einem *mutex*.
- Übergabe einer Referenz an eine Variable (1)

```
std::vector<int> result;
some_stack.pop(result);
```

  Nachteil: benötigt Instanz vom Wertetyp des Stacks vor dem Aufruf
  -> "Teuer" bzw. unmöglich
  -> wichtige Restriktion: Gelagerter Typ muss zuweisbar sein
- Fordern eines no-throw copy- oder move Konstruktors (2)
- Zeigerübergabe (3)
  Nachteil: erfordert Mittel für Verwaltung der Speichergröße
- Bereitstellen von Option 1 und entweder Option 2 oder Option 3

an outline class definition for a thread-safe stack 1/2

```
#include <exception>
#include <memory>  //for std::shared_ptr<>

struct empty_stack: std::exception
{
  const char* what() const throw();
};

template<typename T>
class threadsafe_stack
{
private:
  std::stack<T> data;
  mutable std::mutex m;
public:
  threadsafe_stack();
  threadsafe_stack(const threadsafe_stack& other) //copyable
  {
    std::lock_guard<std::mutex> lock(other.m);
    data=other.data; //copy performed in constructor body
  }
  threadsafe_stack& operator=(const threadsafe_stack&) = delete;  //assignment operator is deleted
  .
  .
  .
```

an outline class definition for a thread-safe stack 2/2

```
1    .
     .
3    .
     void push(T new_value)
5    {
       std::lock_guard<std::mutex> lock(m);
7      data.push(new_value);
     }
9    std::shared_ptr<T> pop();
     {
11     std::lock_guard<std::mutex> lock(m);
       if(data.empty()) throw empty_stack();
13     //allocate return value before modifying stack
       std::shared_ptr<T> const res(std::make_shared<T>(data.top()));
15     data.pop();
       return res;
17   }
     void pop(T& value)
19   {
       std::lock_guard<std::mutex> lock(m);
21     if(data.empty()) throw empty_stack();
       value = data.top();
23     data.pop();
     }
25   bool empty() const
     {
27     std::lock_guard<std::mutex> lock(m);
       return data.empty();
29   }
};
```

deadlocks

- deadlocks können dann auftreten, wenn zwei oder mehrere mutexe gesperrt werden müssen

deadlocks

- deadlocks können dann auftreten, wenn zwei oder mehrere mutexe gesperrt werden müssen
- gängiger Rat: sperre die mutex immer in der gleichen Reihenfolge

Jedoch lässt sich dies nicht immer bewerkstelligen. Zum Beispiel würde schon eine einfache *swap()* Funktion einen *deadlock* erzeugen, wenn zwei Threads mit jeweils vertauschten Parametern auf die Funktion zugreifen würden.

using std::lock() and std::lock_guard in a swap operation

```
class some_big_object;
void swap(some_big_object& lhs, some_big_object& rhs);

class X
{
private:
  some_big_object some_detail;
  std::mutex m;
public:
  X(some_big_object const& sd):some_detail(sd){}

  friend void swap(X& lhs, X& rhs)
  {
    if(&lhs == &rhs)
      return;
    std::lock(lhs.m,rhs.m); //lock both mutexes
    std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock); //lock_guard instances for each mutex
    std::lock_guard<std::mutec> lock_b(rhs.m, std::adopt_lock);
    swap(lhs.some_detail, rhs.some_detail);
  }
};
```

using std::lock() and std::lock_guard in a swap operation

```
1  class some_big_object;
   void swap(some_big_object& lhs, some_big_object& rhs);
3
   class X
5  {
   private:
7    some_big_object some_detail;
     std::mutex m;
9  public:
     X(some_big_object const& sd):some_detail(sd){}
11
     friend void swap(X& lhs, X& rhs)
13   {
       if(&lhs == &rhs)
15       return;
       std::lock(lhs.m,rhs.m); //lock both mutexes
17     std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock); //lock_guard instances for each mutex
       std::lock_guard<std::mutec> lock_b(rhs.m, std::adopt_lock);
19     swap(lhs.some_detail, rhs.some_detail);
     }
21 };
```

lock_guard(mutex_type& m,std::adopt_lock) konstruiert eine *std::lock_guard* Instanz, die den *lock* des mitübergebenen *mutex* besitzt.

guidelines for avoiding deadlock

- Keine verschachtelten Locks

guidelines for avoiding deadlock

- Keine verschachtelten Locks
- Vermeide das Aufrufen von Benutzerseitig bereitgestelltem Code, während ein Lock aktiv ist
  Aber: Dies ist nicht immer vermeidbar. In diesem Fall benötigt man eine neue Richtlinie:

guidelines for avoiding deadlock

- Keine verschachtelten Locks
- Vermeide das Aufrufen von Benutzerseitig bereitgestelltem Code, während ein Lock aktiv ist
  Aber: Dies ist nicht immer vermeidbar. In diesem Fall benötigt man eine neue Richtlinie:
- Locks in fester Reihenfolge setzen

guidelines for avoiding deadlock

- Keine verschachtelten Locks
- Vermeide das Aufrufen von Benutzerseitig bereitgestelltem Code, während ein Lock aktiv ist
  Aber: Dies ist nicht immer vermeidbar. In diesem Fall benötigt man eine neue Richtlinie:
- Locks in fester Reihenfolge setzen
- Spezieller: Lock Hierarchie benutzen

using a lock hierarchy to prevent deadlock

```
1  hierarchical_mutex high_level_mutex(10000);
   hierarchical_mutex low_level_mutex(5000);
3  int do_low_level_stuff();
   int low_level_func()
5  {
     std::lock_guard<hierarchical_mutex> lk(low_level_mutex);
7    return do_low_level_stuff();
   }
9  void high_level_stuff(int some_param);
   void high_level_func()
11 {
     std::lock_guard<hierarchical_mutex> lk(high_level_mutex);
13   high_level_stuff(low_level_func());
   }
15 void thread_a() //works fine
   {
17   high_level_func();
   }
19 hierarchical_mutex other_mutex(100);
   void do_other_stuff();
21 void other_stuff()
   {
23   high_level_func();
     do_other_stuff();
25 }
   void thread_b() //works not fine
27 {
     std::lock_guard<hierarchical_mutex> lk(other_mutex);
29   other_stuff(); //
   }
```

end

Vielen Dank für Ihre Aufmerksamkeit