Verteilte Systeme

...für C++ Programmierer

Zustandsabhängige Steuerung

by

Dr. Günter Kolousek

unique_lock

- Destruktor gibt Lock frei, wenn gelockt
  - kann auch manuell freigegeben werden!
- kann verschoben werden
  - besitzt daher nicht notwendigerweise den `mutex`
- ist rekursiv (mit `recursive_mutex`)
- hat Timeout (mit `timed_mutex`)
- kann auch angelegt werden
  - und den Lock übernehmen (`adopt_lock`)
  - ohne den Lock zu halten (`defer_lock`)
  - mit dem Versuch den Lock zu bekommen, d.h. blockiert nicht (`try_to_lock`)
    - $\rightarrow$ `try_lock()`
- Verwendung mit Bedingungsvariable…
- *nur* zu verwenden, wenn `lock_guard` nicht ausreicht!!

unique_lock-2

```cpp
#include <iostream>  // uniquelock.cpp
#include <thread>
#include <mutex>
using namespace std;
int main() {
    mutex m{};
    thread t1{[&]() { unique_lock<mutex> ul{m};
          cout << 't' << '1' << endl;
          }};  // lock_guard would be sufficient!
    thread t2{[&]() { unique_lock<mutex> ul{m};
          cout << 't' << '2' << endl;
          }};
    t1.join();  t2.join();
}
```

```
t1
t2
```

unique_lock-3

```
#include <iostream>  // uniquelock2.cpp
#include <thread>
#include <mutex>
using namespace std;
using namespace std::literals;
void f(unique_lock<mutex> ul) {
    cout<< 'f'<< endl;
    this_thread::yield();  /* ”no effect” */ }
int main() {
    mutex m{};
    thread t1{[&]() {
      unique_lock<mutex> ul{m};
      cout << 't' << '1'; f(move(ul));
      this_thread::sleep_for(10ms);
      cout << 't'; this_thread::sleep_for(10ms);
      cout << '1' << endl;  }};
```

unique_lock-4

```
    thread t2{[&]() {
      unique_lock<mutex> ul{m};
      cout << 't' << '2'; f(move(ul));
      this_thread::sleep_for(10ms);
      cout << 't'; this_thread::sleep_for(10ms);
      cout << '2' << endl;  }};
    t1.join();
    t2.join();
}
```

Folgende Ausgabe möglich:

```
t2f
t1f
tt2
1
```

unique_lock-5

```cpp
#include <iostream>  // uniquelock3.cpp
#include <thread>
#include <mutex>
using namespace std;
mutex m1{}; mutex m2{};
void f(int i) {
    unique_lock<mutex> ul1{m1, defer_lock};
    unique_lock<mutex> ul2{m2, defer_lock};
    lock(ul1, ul2);
    cout << 'f' << i << endl;
}
int main() {
    mutex m1{}; mutex m2{};
    thread t1{f, 1};  thread t2{f, 2};
    t1.join();  t2.join();
}
```

Problematik – 4

Producer/Consumer Problem

- Ein Boss befüllt eine Queue (Warteschlange) mit Arbeitspaketen.
- Arbeiter entnehmen die Arbeitspakete wieder aus der Queue arbeiten die Arbeitspakete ab.
- Queue ist begrenzt ($\rightarrow$ bounded buffer)
- Zugriff auf Queue muss synchronisiert werden.
- Wenn *Queue voll*, dann kann Boss kein weiteres Arbeitspaket einstellen und muss warten.
- Wenn *Queue leer*, dann kann Arbeiter kein Arbeitspaket entnehmen und muss ebenfalls warten.

Problematik – 5 & Lösungen

$\rightarrow$ Condition synchronization (dt. zustandsabhängige Steuerung) notwendig!

- Polling
  - t1: Locken, Flag setzen (Bedingung überprüfen), Lock freigeben,…
  - t2: Locken, Flag prüfen, Lock freigeben,…
  - $\rightarrow$ Ressourcenverbrauch!
- Polling mit Warten bis zu einer gewissen Zeitpunkt oder für eine gewisse Dauer
  - Zeit
    - zu groß $\rightarrow$ zeitliche Verzögerung
    - zu klein $\rightarrow$ Ressourcenverbrauch
- Thread schlafen legen bis Bedingung eintritt
  - $\rightarrow$ condition variable (Bedingungsvariable)

Polling mit Warten

```cpp
#include <iostream>  // polling.cpp
#include <thread>
#include <mutex>
using namespace std;
using namespace std::literals;
bool flag;
mutex mtx;
void wait_for_flag() {
    unique_lock<mutex> lck{mtx};
    while (!flag) {
        lck.unlock();
        this_thread::sleep_for(100ms);
        cout << "*" << flush;
        lck.lock();
    }
}
```

Polling mit Warten – 2

```
void set_flag() {
    this_thread::sleep_for(3s);
    lock_guard<mutex> lck{mtx};
    flag = true;
}

int main() {
    thread t1{wait_for_flag};
    set_flag();
    t1.join();
    cout << endl << "done!" << endl;
}
```

```
******************************
done!
```

Bedingungsvariable

```
#include <iostream>  // condvar.cpp
#include <thread>
#include <mutex>
#include <condition_variable>
#include <vector>
using namespace std;
using namespace std::literals;

struct IntQueue {
    vector<int> v;  // just for demo purposes
    mutex mtx;
    condition_variable not_empty;
  public:
    void put(int);
    int take();
};
```

Bedingungsvariable – 2

```
void IntQueue::put(int elem) {
    lock_guard<mutex> lck{mtx};
    v.push_back(elem);
    not_empty.notify_one();
}

int IntQueue::take() {
    unique_lock<mutex> lck{mtx};
    not_empty.wait(lck, [this]{ return v.size();});
    int elem{v.front()};
    v.erase(v.begin());
    return elem;
}
```

Bedingungsvariable – 3

```
int main() {
    IntQueue s;
    s.put(1);
    s.put(2);
    cout << s.take() << endl;
    cout << s.take() << endl;
    thread t{[&]() { cout << s.take() << endl; }};
    this_thread::sleep_for(3s);
    s.put(3);
    t.join();
}

1
2
3   // after 3 seconds
```

Bedingungsvariable – 4

- `condition_variable` (`unique_lock` erforderlich!)
- `notify_one` bzw. `notify_all`
  - Lock muss nicht gehalten werden!
  - Irgendeiner bzw. alle wartenden Threads
    - kein wartender → geht verloren: ”lost wake-up”
- `void wait();` mit Prädikatsfunktion: `bool wait_for(), bool wait_until()`
  - muss gelockt sein! Thread geht schlafen & unlocken
  - Wenn Prädikatsfunktion angegeben, dann äquivalent zu

    ```
    while (!pred()) wait(lck);
    ```

  - Wenn benachrichtigt | Timeout | ”spurious wakeup”, dann
    - Thread wird aufgeweckt und Lock wird erworben
    - Wenn Bedingung falsch → schlafen legen & unlocken
    - ”spurious wakeup” → no side effects, please!
    - Rückgabe des Wahrheitswertes der Bedingung (wenn abgelaufen, dann `false`)

Lost Wakeup

```
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <thread>

using namespace std;

mutex mtx;
condition_variable data_ready;

void waiting_for_work(){
    cout << ”waiting...” << endl;
    unique_lock<mutex> lck(mtx);
    data_ready.wait(lck);  // 1
    cout << ”running ” << endl;
}
```

Lost Wakeup – 2

```
void set_data_ready(){
    cout << ”data prepared!” << endl;
    data_ready.notify_one();  // 2
}

int main() {
    thread t1{set_data_ready};
    thread t2{waiting_for_work};
    t1.join();
    t2.join();
}
```

```
data prepared!
waiting...
```