

30. Parallel Programming IV

Futures, Read-Modify-Write Instruktionen, Atomare Variablen, Idee der lockfreien Programmierung

[C++ Futures: Williams, Kap. 4.2.1-4.2.3] [C++ Atomic: Williams, Kap. 5.2.1-5.2.4, 5.2.7] [C++ Lockfree: Williams, Kap. 7.1.-7.2.1]

Futures: Motivation

Threads waren bisher Funktionen ohne Resultat:

```
void action(some parameters){
    ...
}

std::thread t(action, parameters);
...
t.join();
// potentially read result written via ref-parameters
```

924

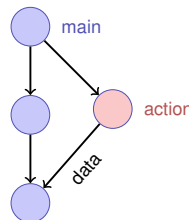
925

Futures: Motivation

Wir wollen nun etwas in dieser Art:

```
T action(some parameters){
    ...
    return value;
}

std::thread t(action, parameters);
...
value = get_value_from_thread();
```



Wir können das schon!

- Wir verwenden das Producer/Consumer Pattern (implementiert mit Bedingungsvariablen)
- Starten einen Thread mit Referenz auf den Buffer
- Wenn wir das Resultat brauchen, holen wir es vom Buffer
- Synchronisation ist ja bereits implementiert

926

927

Zur Erinnerung

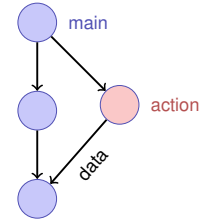
```
template <typename T>
class Buffer {
    std::queue<T> buf;
    std::mutex m;
    std::condition_variable cond;
public:
    void put(T x){ std::unique_lock<std::mutex> g(m);
        buf.push(x);
        cond.notify_one();
    }
    T get(){ std::unique_lock<std::mutex> g(m);
        cond.wait(g, [&]{return (!buf.empty());});
        T x = buf.front(); buf.pop(); return x;
    }
};
```

928

Anwendung

```
void action(Buffer<int>& c){
    // some long lasting operation ...
    c.put(42);
}

int main(){
    Buffer<int> c;
    std::thread t(action, std::ref(c));
    t.detach(); // no join required for free running thread
    // can do some more work here in parallel
    int val = c.get();
    // use result
    return 0;
}
```

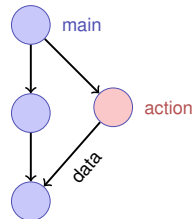


929

Mit C++11 Bordmitteln

```
int action(){
    // some long lasting operation
    return 42;
}

int main(){
    std::future<int> f = std::async(action);
    // can do some work here in parallel
    int val = f.get();
    // use result
    return 0;
}
```



930

30.2 Read-Modify-Write

931

Beispiel: Atomare Operationen in Hardware

CMPXCHG **Compare and Exchange**

Compares the value in the AL, AX, EAX, or RAX register with the value in a register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the AL, AX, EAX, or RAX register and clears the ZF flag to 0.

The OF, SF, AF, DF, and CF flags are set to reflect the results of the compare.

When the first memory operand is the same as the second, the instruction performs a read-modify-write on the memory location. If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the flag registers to 1. Otherwise it copies the value in the first operand to the first operand and clears ZF flag to 0.

The forms of the LOCK prefix. For details see the LOCK prefix section.

Mnemonic: CMPXCHG r/m16, r/m16, r/m16

Instruction Formats

24594—Rev. 3.14—September 2007 AMD64 Technology

«The lock prefix causes certain kinds of memory read-modify-write instructions to occur atomically»

AMD64 Architecture Programmer's Manual

932

Read-Modify-Write

Konzept von **Read-Modify-Write**: Lesen, Verändern und Zurückschreiben, zu einem Zeitpunkt (atomar).

933

Beispiel: Test-And-Set

Pseudo-Code für TAS (C++ Stil):

```
bool TAS(bool& variable){  
    atomic bool old = variable;  
    variable = true;  
    return old;  
}
```

934

Verwendungsbeispiel TAS in C++11

Wir bauen unser eigenes Lock:

```
class SpinLock{  
    std::atomic_flag taken {false};  
public:  
  
    void lock(){  
        while (taken.test_and_set()); // TAS returns old value  
    }  
  
    void unlock(){  
        taken.clear();  
    }  
};
```

935

30.3 Lock-Freie Programmierung

Ideen

Compare-And-Swap

```
bool CAS(int& variable, int& expected, int desired){  
    if (variable == expected){  
        variable = desired;  
        return true;  
    }  
    else{  
        expected = variable;  
        return false;  
    }  
}
```

atomic

936

937

Lock-freie Programmierung

Datenstruktur heisst

- **lock-frei**: zu jeder Zeit macht mindestens ein Thread in beschränkter Zeit Fortschritt, selbst dann, wenn viele Algorithmen nebenläufig ausgeführt werden. Impliziert systemweiten Fortschritt aber nicht Starvationfreiheit.
- **wait-free**: jeder Thread macht zu jeder Zeit in beschränkter Zeit Fortschritt, selbst dann wenn andere Algorithmen nebenläufig ausgeführt werden.

Fortschrittsbedingungen

	Lock-frei	Blockierend
Jeder macht Fortschritt	Wait-frei	Starvation-frei
Mindestens einer macht Fortschritt	Lock-frei	Deadlock-frei

938

939

Implikation

- Programmieren mit Locks: jeder Thread kann andere Threads beliebig blockieren.
- Lockfreie Programmierung: der Ausfall oder das Aufhängen eines Threads kann nicht bewirken, dass andere Threads blockiert werden

Wie funktioniert lock-freie Programmierung?

Beobachtung:

- RMW-Operationen sind in Hardware *Wait-Free* implementiert.
- Jeder Thread sieht das Resultat eines CAS oder TAS in begrenzter Zeit.

Idee der lock-freien Programmierung: lese Zustand der Datenstruktur und verändere die Datenstruktur *atomar* dann und nur dann, wenn der gelesene Zustand unverändert bleibt.

940

941

Beispiel: lock-freier Stack

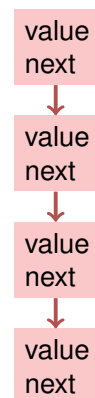
Nachfolgend vereinfachte Variante eines Stacks

- pop prüft nicht, ob der Stack leer ist
- pop gibt nichts zurück

(Node)

Nodes:

```
struct Node {  
    T value;  
  
    Node<T>* next;  
    Node(T v, Node<T>* nxt): value(v), next(nxt) {}  
};
```

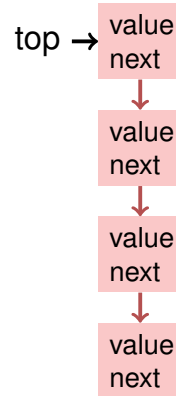


942

943

(Blockierende Version)

```
template <typename T>
class Stack {
    Node<T> *top=nullptr;
    std::mutex m;
public:
    void push(T val){ guard g(m);
        top = new Node<T>(val, top);
    }
    void pop(){ guard g(m);
        Node<T>* old_top = top;
        top = top->next;
        delete old_top;
    }
};
```



Lock-Frei

```
template <typename T>
class Stack {
    std::atomic<Node<T>*> top {nullptr};
public:
    void push(T val){
        Node<T>* new_node = new Node<T> (val, top);
        while (!top.compare_exchange_weak(new_node->next, new_node));
    }
    void pop(){
        Node<T>* old_top = top;
        while (!top.compare_exchange_weak(old_top, old_top->next));
        delete old_top;
    }
};
```

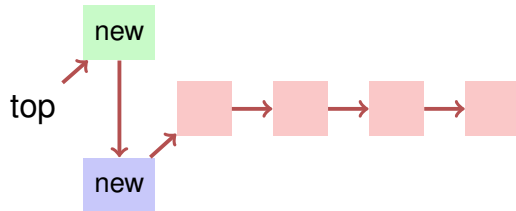
944

945

Push

```
void push(T val){
    Node<T>* new_node = new Node<T> (val, top);
    while (!top.compare_exchange_weak(new_node->next, new_node));
}
```

2 Threads:

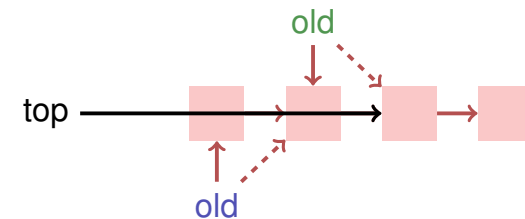


946

Pop

```
void pop(){
    Node<T>* old_top = top;
    while (!top.compare_exchange_weak(old_top, old_top->next));
    delete old_top;
}
```

2 Threads:



947

Lockfreie Programmierung – Grenzen

- Lockfreie Programmierung ist kompliziert.
- Wenn mehr als ein Wert nebenläufig angepasst werden muss (Beispiel: Queue), wird es schwieriger. Damit Algorithmen lock-frei bleiben, müssen Threads sich “gegenseitig helfen”.
- Bei Speicherwiederverwendung kann das *ABA Problem* auftreten.