

30. Parallel Programming IV

Futures, Read-Modify-Write Instructions, Atomic Variables, Idea of lock-free programming

[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

Up to this point, threads have been functions without a result:

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

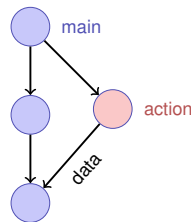
1001

1002

Futures: Motivation

Now we would like to have the following

```
T action(some parameters){  
    ...  
    return value;  
}  
  
std::thread t(action, parameters);  
...  
value = get_value_from_thread();
```



We can do this already!

- We make use of the producer/consumer pattern, implemented with condition variables
- Start the thread with reference to a buffer
- We get the result from the buffer.
- Synchronisation is already implemented

1003

1004

Reminder

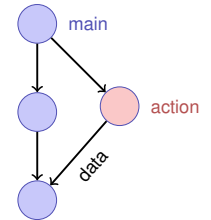
```
template <typename T>
class Buffer {
    std::queue<T> buf;
    std::mutex m;
    std::condition_variable cond;
public:
    void put(T x){ std::unique_lock<std::mutex> g(m);
        buf.push(x);
        cond.notify_one();
    }
    T get(){ std::unique_lock<std::mutex> g(m);
        cond.wait(g,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

1005

Application

```
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;
}
```

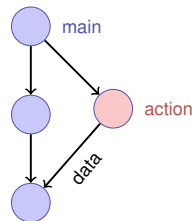


1006

With features of C++11

```
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;
}
```



1007

30.2 Read-Modify-Write

1008

Example: Atomic Operations 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 operand, the instruction performs a read-modify-write on the same value to the same location. For details about the LOCK prefix, see the LOCK instruction.

The forms of the LOCK prefix are:

Mnemonic: `LOCK CMPXCHG reg, mem`

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

1009

Read-Modify-Write

Concept of **Read-Modify-Write**: The effect of reading, modifying and writing back becomes visible at one point in time (happens atomically).

Pseudocode for CAS – Compare-And-Swap

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

atomic

Application example CAS in C++11

We build our own (spin-)lock:

```
class Spinlock{
    std::atomic<bool> taken {false};
public:
    void lock(){
        bool old = false;
        while (!taken.compare_exchange_strong(old=false, true)){}
    }
    void unlock(){
        bool old = true;
        assert(taken.compare_exchange_strong(old, false));
    }
};
```

30.3 Lock-Free Programming

Ideas

Lock-free programming

Data structure is called

- *lock-free*: at least one thread always makes progress in bounded time even if other algorithms run concurrently. Implies system-wide progress but not freedom from starvation.
- *wait-free*: all threads eventually make progress in bounded time. Implies freedom from starvation.

1013

1014

Progress Conditions

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free

Implication

- Programming with locks: each thread can block other threads indefinitely.
- Lock-free: failure or suspension of one thread cannot cause failure or suspension of another thread !

1015

1016

Lock-free programming: how?

Beobachtung:

- RMW-operations are implemented *wait-free* by hardware.
- Every thread sees his result of a CAS or TAS in bounded time.

Idea of lock-free programming: read the state of a data structure and change the data structure *atomically* if and only if the previously read state remained unchanged meanwhile.

1017

Example: lock-free stack

Simplified variant of a stack in the following

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

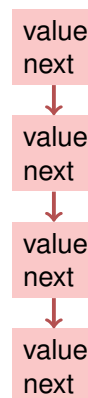
1018

(Node)

Nodes:

```
struct Node {
    T value;

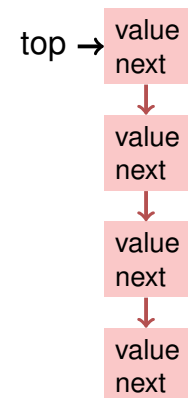
    Node<T>* next;
    Node(T v, Node<T>* nxt): value(v), next(nxt) {}
};
```



1019

(Blocking 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;
    }
};
```



1020

Lock-Free

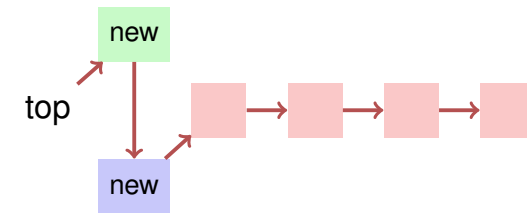
```
template <typename T>
class Stack {
    std::atomic<Node<T*>> top {nullptr};
public:
    void push(T val){
        Node<T*> new_node = new Node<T> (val, top);
        while (!top.compare_exchange_weak(new_node->next, new_node));
    }
    void pop(){
        Node<T*> old_top = top;
        while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

1021

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:

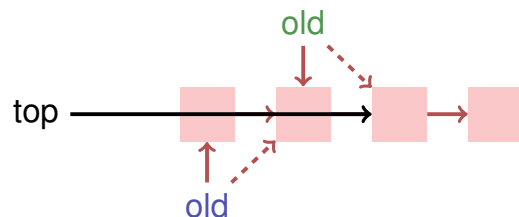


1022

Pop

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

2 Threads:



1023

Lock-Free Programming – Limits

- Lock-Free Programming is complicated.
- If more than one value has to be changed in an algorithm (example: queue), it is becoming even more complicated: threads have to “help each other” in order to make an algorithm lock-free.
- The *ABA problem* can occur if memory is reused in an algorithm. A solution of this problem can be quite expensive.

1024