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

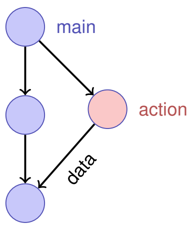
Up to this point, threads have been functions without a result:

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

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


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();
```



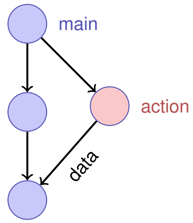
We can do this already!

- 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

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;
  }
};
```

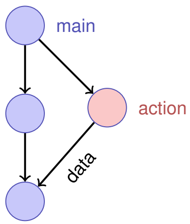

Application




```
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;
}
```

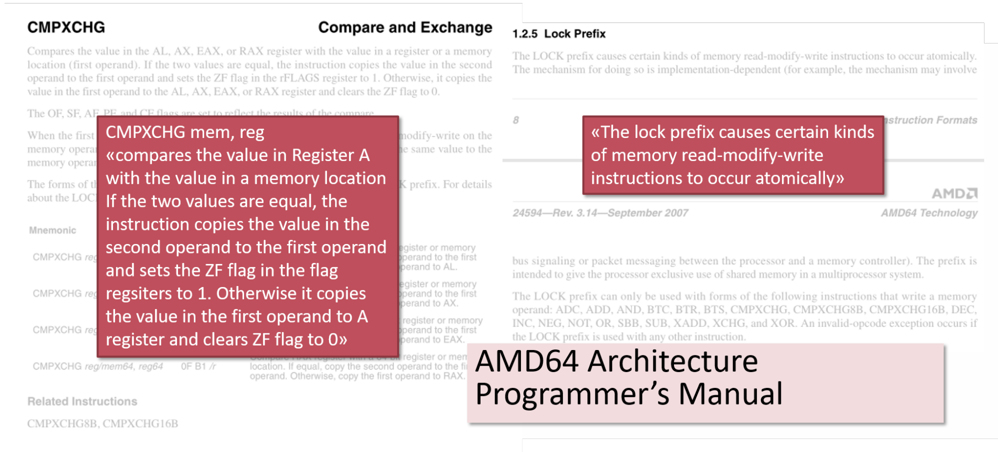

With features of C++11

```
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;
}
```



30.2 Read-Modify-Write

Example: Atomic Operations in Hardware



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).

Psudocode for CAS – Compare-And-Swap

atomic

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


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.

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 !

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 sructure and change the data structure *atomically* if and only if the previously read state remained unchanged meanwhile.

Example: lock-free stack

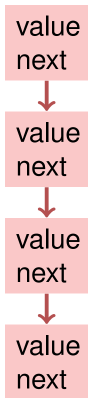
Simplified variant of a stack in the following

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

(Node)

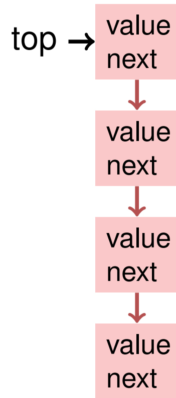
Nodes:

```
struct Node {
  T value;

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



(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;
    }
};
```



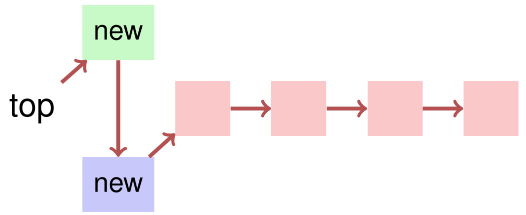
Lock-Free

```
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;
  }
};
```


Push

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

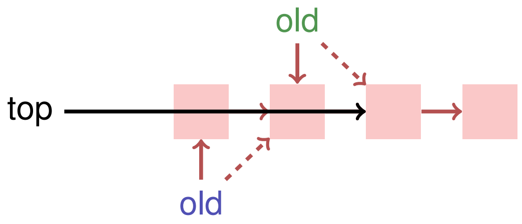
2 Threads:



Pop

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

2 Threads:



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.