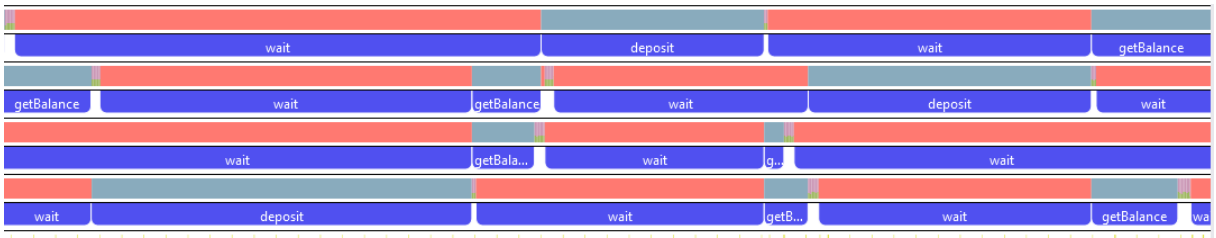
Prof. Dr. C. Stamm christoph.stamm@fhnw.ch Tel.: 056 202 78 32

Exercise 0

In computer systems, it is common to have some processes (called readers) that read data and others (called writers) that write it. For example, in a bank there may be many more readers than writers – many inquiries will be made against a database of bank accounts before the customer withdraws or deposits some money.

Because readers do not change the balance of the account, many readers may access the account at once (see getBalance in the figure). But a writer can modify the account balance, so it must have exclusive access (deposit in the figure). When a writer is active, no other readers or writers may be active. This exclusion needs only to be enforced at the record level. It is not necessary to grant a writer exclusive access to the entire database.



1 Read-Write Lock in modern C++

Implement the methods of the following class for the “readers and writers” problem. Only one writer may be active at once, when a writer is active, the Boolean variable `writeLocked` is true. The variable `readLocked` indicates the number of active readers. When the number of readers is reduced to zero, then a waiting writer may be become active. The condition variable `readingAllowed` is waited upon by a new reader that cannot proceed, because a writer is active. The condition variable `writingAllowed` is waited upon by a new writer that cannot proceed, because another writer or some readers are active. Accessing these four variables must be synchronized. Therefore, a `mutex` managed by a `unique_lock` is used to grant exclusive access to critical sections (the `unique_lock` is necessary in combination with condition variables):

```
{
   unique_lock<mutex> monitor(m_mutex);
   // critical section
}
```

When a reader wishes to read, it calls `lockR()`; a reader that has finished calls `unlockR()`. When a process wishes to write, it calls `lockW()`; a writer that has finished calls `unlockW()`.

```
class ConditionVariable : public condition_variable {
    size_t m_waitingThreads;            // number of waiting threads

public:
    ConditionVariable() : m_waitingThreads(0) {}
    void wait(unique_lock<mutex>& m) {
        m_waitingThreads++;
        condition_variable::wait(m);
        m_waitingThreads--;
    }
    bool hasWaitingThreads() const { return m_waitingThreads > 0; }
};

class RWLock {
    mutex m_mutex;                      // re-entrance not allowed
    ConditionVariable m_readingAllowed; // true: no writer at work
    ConditionVariable m_writingAllowed; // true: no reader and no writer at work
    bool m_writeLocked = false;         // locked for writing
    size_t m_readLocked = 0;            // number of concurrent readers
public:
```

```
    size_t getReaders() const;      // returns number of concurrent readers (debugging)
    void lockR();                   // locks for reading
    void unlockR();                 // unlocks a read-lock
    void lockW();                   // locks for writing
    void unlockW();                 // unlocks a write-lock
};
```

The modern C++ standards are very explained on the website: cppreference.com

2 Bank Account

Implement the methods of the following class for a bank account. Slow down the execution speed of `getBalance()` and `deposit()` with `this_thread::sleep_for(chrono::milliseconds(…))` to make the concurrent effects clearer.

```
class BankAccount {
   mutable RWLock m_lock;           // mutable: can be modified even in const methods
   double m_balance = 0;            // bank account balance
public:
   size_t getReaders() const;       // returns number of concurrent readers (debugging)
   double getBalance() const;       // returns the account balance
   void deposit(double amount);     // modifies the account balance
};
```