References

Ben-Ari, *Principles of Concurrent and Distributed Programming*, second edition, Addison-Wesley, 2006.
Hinnant, Howard E., "Mutex, Lock, Condition Variable Rationale", Document number: N2406=07-0266, September 2007.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2406.html

Introduction

Ben-Ari's Algorithm 7.4 is the solution to the readers writers problem for a Hoare semantics monitor. It defines four monitor operations, StartRead, EndRead, StartWrite, and EndWrite, implemented with the help of two condition variables, OKtoRead and OKtoWrite.

Program `ReadersWritersA` in our Chapter 7 slides shows how to use the `shared_mutex` and `shared_lock` types with the RAII design pattern. In that pattern, the lock and unlock operations are implicit. When a lock is allocated on the run-time stack, its constructor locks the mutex. The declaration of the lock is visible in the code, but the lock operation is not. When a lock is deallocated on function termination, its destructor unlocks the mutex. Neither the deallocation nor the unlock operation is visible in the code. That is why the code in `ReadersWritersA` appears so simple.

Here is the correspondence between the terminology of Ben-Ari and C++17.

- StartRead corresponds to `lock_shared()`.
- EndRead corresponds to `unlock_shared()`.
- StartWrite corresponds to `lock()`.
- EndWrite corresponds to `unlock()`.

The Terekhov algorithm

Without the `shared_mutex` type, introduced in C++17, you would need to program the readers writers problem with just the `mutex` type. The Terekhov algorithm is a solution to the readers writers problem without the `shared_mutex` type. It is the algorithm C++17 uses to implement the `shared_mutex` type with the above correspondence.

Program `ReadersWritersB`, which is available in our `cosc450CppDistr` software distribution, is the Terekhov algorithm for the readers writers problem using the Ben-Ari terminology. The monitor code is below. See the software distribution for the complete program in the CLion IDE.

The algorithm maintains two condition variables, `gate1` and `gate2`, with the following rules.

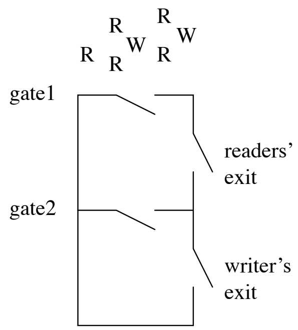
- When a reader enters `gate1`, it has read access. However, a writer must enter first `gate1` and then `gate2` to have write access.
- There can be any number of readers and at most one writer inside `gate1`. There cannot be any readers inside `gate2`.
- No one can enter `gate1` if a writer is inside `gate1` or `gate2`. If a reader or writer tries to enter it is blocked on `gate1`.
- A writer can only enter `gate2` when the number of readers inside `gate1` drops to 0. If it tries to enter `gate2` when there are readers inside `gate1` it is blocked on `gate2`.

Implementation of the Terekhov algorithm

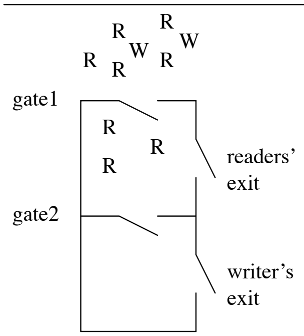
```
class RWMonitor {
private:
    mutex rwMutex;
    condition_variable gate1;
    condition_variable gate2;
    int readers = 0;
    bool writer = false;

public:
    void startRead() {
        unique_lock<mutex> guard(rwMutex);
        gate1.wait(guard, [this] { return !writer; });
        readers++;
    }

    void endRead() {
        unique_lock<mutex> guard(rwMutex);
        readers--;
        if (writer && (readers == 0)) {
            gate2.notify_one();
        }
    }

    void startWrite() {
        unique_lock<mutex> guard(rwMutex);
        gate1.wait(guard, [this] { return !writer; });
        writer = true;
        gate2.wait(guard, [this] { return readers == 0; });
    }

    void endWrite() {
        unique_lock<mutex> guard(rwMutex);
        readers = 0;
        writer = false;
        gate1.notify_all();
    }
};
```

Terekhov algorithm scenario

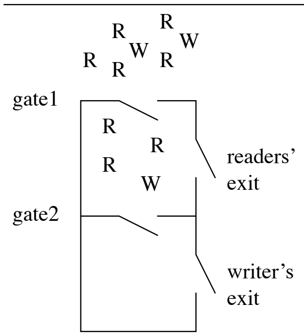
The following figure illustrates the progression of states with the Terekhov algorithm.



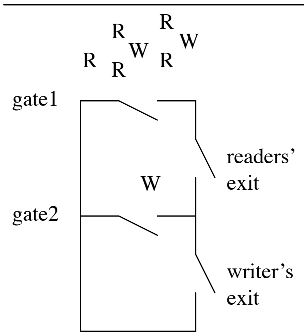
Initial state



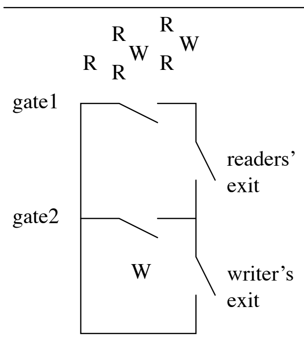
Some readers enter gate1 and exit.



A writer enters gate1.
Readers and writers are blocked on gate1.



The number of readers inside gate1 drops to 0.



The writer enters gate2.
Readers and writers are still blocked on gate1.
When the writer exits, the system returns to its initial state.

Optimization techniques

Program `ReadersWritersC` in our software distribution is an optimized implementation of the Terekhov algorithm for the readers writers problem using the Ben-Ari terminology. For example, `startRead()` is how C++17 implements `lock_shared()`. The monitor code is below.

The number of readers inside gate 1, an integer, and whether a writer is inside gate1 or gate2, a boolean, define the state of the computation. The optimized version encodes the state in a single unsigned integer named `state`. The first bit of `state` is 1 if `writer` is true and 0 otherwise. The remaining bits are the count of `readers`.

The optimization uses two constant masks, `writerMask`, whose first bit is 1 and remaining bits are 0, and `readerMask`, whose first bit is 0 and remaining bits are 1.

Typically, an unsigned integer would be 32 or 64 bits long. Here are some examples with an 8-bit unsigned integer.

`writerMask`: 1000 0000
`readerMask`: 0111 1111

`state`: 0000 0110 ⇒ six readers inside gate1 and no writer inside gate1 or gate2
`state`: 1000 0110 ⇒ six readers inside gate1 and one writer inside gate1 or gate2

The optimization uses bitwise `&` and bitwise `|` operations, which are extremely fast, with the masks to extract the `readers` and `writer` values on the fly. It is coded to be safe from integer overflow. Here are some examples of expressions in the optimized code and their meanings. Note the C semantics that integer zero is false and nonzero is true.

Expression	Meaning
`state & writerMask`	True iff a writer is inside gate1 or gate2
`state & readerMask`	Number of readers inside gate1
`(state & readerMask) == readerMask`	True iff the number of readers inside gate1 is the maximum we can count
`readers == readerMask - 1`	True iff the number of readers inside gate1 is one less than the maximum we can count
`unsigned readers = (state & readerMask) + 1; state &= writerMask; state |= readers;`	Adds 1 to number of readers
`state |= writerMask;`	Sets `state` to specify that a writer is inside

The optimized code also programs the spurious wakeup loop explicitly without the predicate parameter in the `wait()` function. For example, in `startWrite()` the unoptimized statement

```
gate1.wait(guard, [this] { return !writer; });
```

is coded as

```
while (state & writerMask)
    gate1.wait(guard);
```

Optimized implementation of the Terekhov algorithm

```
class RWMonitor {
private:
    mutex rwMutex;
    condition_variable gate1;
    condition_variable gate2;
    unsigned state = 0;
    static const unsigned writerMask = 1U << (sizeof(unsigned) * CHAR_BIT - 1);
    static const unsigned readerMask = ~writerMask;

public:
    void startRead() {
        unique_lock<mutex> guard(rwMutex);
        while ((state & writerMask) || (state & readerMask) == readerMask)
            gate1.wait(guard);
        unsigned readers = (state & readerMask) + 1;
        state &= writerMask;
        state |= readers;
    }

    void endRead() {
        unique_lock<mutex> guard(rwMutex);
        unsigned readers = (state & readerMask) - 1;
        state &= writerMask;
        state |= readers;
        if (state & writerMask) {
            if (readers == 0)
                gate2.notify_one();
        } else {
            if (readers == readerMask - 1)
                gate1.notify_one();
        }
    }
```

```
    void startWrite() {
        unique_lock<mutex> guard(rwMutex);
        while (state & writerMask)
            gate1.wait(guard);
        state |= writerMask;
        while (state & readerMask)
            gate2.wait(guard);
    }

    void endWrite() {
        unique_lock<mutex> guard(rwMutex);
        state = 0;
        gate1.notify_all();
    }
};
```