Readers Writers problem

The problem: in a concurrent situation

- You don't have to protect access to a shared resource if it is only read by multiple threads.
-
- **BUT, as soon as there is a single write, then all accesses, both Reads and Writes must be protected!**
-
- But what if you have mostly readers? Can you make it more efficient for them?
- Or if you can somehow block all writers, then can you allow unrestricted concurrent reads from multiple threads?

The problem: in a concurrent situation

-
-
- Yes, you can

The light switch problem

- A bunch of people will meet in a room
- The first person turns on the lights
- Everyone else walks in
- Meeting occurs
- Everyone walks out
- Last person turns out the lights

The light switch problem

- A bunch of people will meet in a room
- The first person turns on the lights
- Everyone else walks in
- Meeting occurs
- Everyone walks out
- Last person turns out the lights
-

Applied to our Readers/Writers Problem

- First reader thread locks out all writer threads
- All reader threads can read concurrently
- Last reader thread lets writers back in...

Applied to our Readers/Writers Problem

- First reader thread locks out all writer threads
- All reader threads can read concurrently
- Last reader thread lets writers back in...

- Rules:
  - Any number of readers can read at a time
  - Only 1 writer can write at a time
  - If a writer is currently writing shared data, no readers may read it

Applied to our Readers/Writers Problem

- Need a way to track number of readers so we know when the first reader enters and the last reader exits.
- This is reference counting

```
class Reader_Writer_lock {
public:
    Reader_Writer_lock();
    virtual ~Reader_Writer_lock();

    //readers should call read when they start
    //and a coresponding read_done() when finished
    //MANY readers at a time, NO writer
    void read();
    void read_done();

    //writers should call write when they start
    //and a coresponding write_done() when finished
    //NO readers at a time, ONE writer
    void write();
    void write_done();
private:
    int curReaders;          //how many readers?
    std::mutex mNoWriters;   //locks out writers
    std::mutex mCount;       //lock access to curReaders
};
```

```
void Reader_Writer_lock::read() {
    //one reader at a time in this block
    //on exit below guard unlocks
    //and lets other readers in
    //but the writer is locked out!
    lock_guard<mutex> lck(mCount);

    curReaders++;          //indicate there is a reader

    if (curReaders == 1)     //first reader
        mNoWriters.lock();   //then lock out writers
}
void Reader_Writer_lock::read_done() {
    //one reader at a time in this block
    //on exit below guard unlocks
    //but the writer is locked out!
    lock_guard<mutex> lck(mCount);

    curReaders--;            //indicate a leaving reader
    if (curReaders == 0)     //if no readers
        mNoWriters.unlock();//then let in the writers
}
```

Applied to our Readers/Writers Problem

- Almost works.
- The problem is that mutexes require that the thread that locked them must be the thread that unlocks them.

- Semaphores do not have this requirement.
- So change the semaphore to a mutex and it all works flawlessly!

Applied to our Readers/Writers Problem

- Demo **410_readers_writers_mutexes** project for a demo with mutexes (FLAWED!) and a correct solution with semaphores.