Lecture 9
Reader-Writer Locks

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me
Outline

• Coordinating Resources: Reasoning about two mutex/semaphore-based schemes
  • Reader-Writer Locks
The Reader-Writer Problem

• Consider a resource
  • Shared by several threads
  • Some threads may only want to read
  • Others may want to modify

• Could we coordinate these writers and readers?

• Strawman: single mutex \textit{will} work, but ...
Motivating Example: Responsive Hashtable

- Many read operations, rare write operations
- `lookup()`
  - Common operation; should allow many read-only `lookup()`s in parallel
- `insert()`
  - Uncommon operation, should block all reads while writing
  - Could lock a single bucket – but insertions can cause resizes!

*Note:* `lookup()` must be truly read-only
The Reader-Writer Problem

• Idea: a reader-writer lock [pair]
  • Each reader acquires a special lock that allows them to share the resource with other readers
  • A writer acquires another kind of lock that gives it exclusive access to the resource
  • The locks work in tandem to guarantee the resource’s consistency

• Let’s worry about how to build this after we see it in action.

• Hashtable: read lock for lookup(), write lock for insert()
POSIX File Reader-Writer Lock

• File locking between processes or threads

• `flock(file_handle, mode)`
  • LOCK_SH: Shared (reader) lock
  • LOCK_EX: Exclusive (writer) lock
  • Bitwise OR with LOCK_NB: Nonblocking

Process 1:
```c
FILE* fh = fopen(F);
flock(fh, LOCK_SH);
```

Locked: Shared, 1
POSIX File Reader-Writer Lock

- File locking between processes or threads
- `flock(file_handle, mode)`
  - LOCK_SH: Shared (reader) lock
  - LOCK_EX: Exclusive (writer) lock
  - Bitwise OR with LOCK_NB: Nonblocking

Process 2:
```c
FILE* fh = fopen(F);
flock(fh, LOCK_SH);
```
POSIX File Reader-Writer Lock

- File locking between processes or threads
- `flock(file_handle, mode)`
  - LOCK_SH: Shared (reader) lock
  - LOCK_EX: Exclusive (writer) lock
  - Bitwise OR with LOCK_NB: Nonblocking

```
Process 3:
FILE* fh = fopen(F);
flock(fh, LOCK_EX);
```
POSIX File Reader-Writer Lock

- File locking between processes or threads
- `flock(file_handle, mode)`
  - LOCK_SH: Shared (reader) lock
  - LOCK_EX: Exclusive (writer) lock
  - Bitwise OR with LOCK_NB: Nonblocking

Process 1:
```
flock(fh, LOCK_UN);
```

Process 2:
```
flock(fh, LOCK_UN);
```
Simple Reader-Writer Lock

• Forgot files: let’s implement a simple reader–writer lock

• Semantics:
  • Allow any number of shared readers
  • Allow a single exclusive writer
  • Fairness? Worry about it later

• Toolset
  • Mutices
Strawman #2: Writer Mutex

• Single mutex: writer mutex

```python
mutex write_lock
```

```python
reader_lock():
    pass
reader_unlock():
    pass
```

```python
writer_lock():
    lock(write_lock)
writer_unlock():
    unlock(write_lock)
```

• Allow any number of shared readers
• Allow a single exclusive writer
• What about a reader and a writer?
Strawman #3: Better Writer Mutex

- Single mutex: writer mutex

```python
mutex write_lock
```

```python
caller = 0
while (caller == 0):
    reader_lock():
        lock(write_lock)
    reader_unlock():
        unlock(write_lock)
```  

```python
writer_lock():
    lock(write_lock)
writer_unlock():
    unlock(write_lock)
```

- Allow any number of shared readers
- Allow a single exclusive writer
- Prevents simultaneous readers + writers
Towards the R-W Lock: Add a Read Count

```c
int read_count = 0
mutex write_lock

reader_lock():
    read_count += 1
    if read_count == 1:
        lock(write_lock)

reader_unlock():
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)

writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```

- Allow any number of shared readers
- Allow a single exclusive writer
- Prevents simultaneous readers + writers
Towards the R-W Lock: Add a Read Count

```c
int read_count = 0
mutex write_lock

reader_lock():
    read_count += 1
    if read_count == 1:
        lock(write_lock)

reader_unlock():
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)

writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```

- Allow any number of shared readers
- Allow a single exclusive writer
- Prevents simultaneous readers + writers
- Introduced race condition
Towards the R-W Lock: Add a Read Count

```c
int read_count = 0
mutex write_lock

reader_lock():
    read_count += 1
    if read_count == 1:
        lock(write_lock)

reader_unlock():
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)

writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```

- Allow any number of shared readers
- Allow a single exclusive writer
- Prevents simultaneous readers + writers
- Introduced race condition
Simple Reader-Writer Lock

- Add a critical section in reader lock/unlock

```python
int read_count = 0
mutex mut_read, write_lock

reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)

writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```

Who gets the priority? Readers or writers?
Simple Reader-Writer Lock

Reader arrives before writer

```c
int read_count = 1
mutex mut_read, write_lock

reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)

writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```
Simple Reader-Writer Lock

Reader arrives before writer

```c
int read_count = 1
mutex mut_read, write_lock

reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)

writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```

```c
int read_count = 1
mutex mut_read, write_lock

reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)
```
Simple Reader-Writer Lock: Starvation

Second reader arrives before first reader finishes

```c
int read_count = 2
mutex mut_read, write_lock

reader_lock():
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)

writer_lock():
    lock(write_lock)

writer_unlock():
    unlock(write_lock)
```
Reader-Writer Lock v2

Give writers priority over readers.

```c
int read_count, write_count
mutex mut_read, mut_write, read_lock, write_lock

reader_lock():
    lock(read_lock)
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)
    unlock(read_lock)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)

writer_lock():
    lock(mut_write)
    write_count += 1
    if write_count == 1:
        lock(read_lock)
    unlock(mut_write)
    lock(write_lock)

writer_unlock():
    lock(mut_write)
    write_count -= 1
    if write_count == 0:
        unlock(read_lock)
    unlock(mut_write)
    unlock(write_lock)
```
Reader-Writer Lock v2

One reader, then one writer, arrives.

```python
int read_count = 1, write_count
mutex mut_read, mut_write, read_lock, write_lock

reader_lock():
    lock(read_lock)
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)
    unlock(read_lock)

reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)

writer_lock():
    lock(mut_write)
    write_count += 1
    if write_count == 1:
        lock(read_lock)
    unlock(mut_write)
    lock(write_lock)

writer_unlock():
    lock(mut_write)
    write_count -= 1
    if write_count == 0:
        unlock(read_lock)
    unlock(mut_write)
    unlock(write_lock)
```
Second reader arrives.

```c
int read_count = 0, write_count = 2
mutex mut_read, mut_write, read_lock, write_lock

_reader_lock():
    lock(read_lock)
    lock(mut_read)
    read_count += 1
    if read_count == 1:
        lock(write_lock)
    unlock(mut_read)
    unlock(read_lock)

_reader_unlock():
    lock(mut_read)
    read_count -= 1
    if read_count == 0:
        unlock(write_lock)
    unlock(mut_read)

_writer_lock():
    lock(mut_write)
    write_count += 1
    if write_count == 1:
        lock(read_lock)
    unlock(mut_write)
    lock(write_lock)

_writer_unlock():
    lock(mut_write)
    write_count -= 1
    if write_count == 0:
        unlock(read_lock)
    unlock(mut_write)
    unlock(write_lock)
```
Reader-Writer Lock v2

Now writers get priority

```plaintext
int read_count = 0, write_count = 2
mutex mut_read, mut_write, read_lock, write_lock

reader_lock():
  lock(read_lock)
  lock(mut_read)
  read_count += 1
  if read_count == 1:
    lock(write_lock)
  unlock(mut_read)
  unlock(read_lock)

reader_unlock():
  lock(mut_read)
  read_count -= 1
  if read_count == 0:
    unlock(write_lock)
  unlock(mut_read)

writer_lock():
  lock(mut_write)
  write_count += 1
  if write_count == 1:
    lock(read_lock)
  unlock(mut_write)
  lock(write_lock)

writer_unlock():
  lock(mut_write)
  write_count -= 1
  if write_count == 0:
    unlock(read_lock)
  unlock(mut_write)
  unlock(write_lock)
```
Reader-Writer Lock v3

Better performance, with cond vars. Tracing a reader, then a writer

```c
int a_readers, a_writers, p_readers, p_writers // Active & pending
mutex mut, cond_var read_cond, write_cond

reader_lock():
    lock(mut)
    while a_writers + p_writers:
        p_readers += 1
        read_cond.wait(mut)
        p_readers -= 1
    a_readers += 1
    unlock(mut)

reader_unlock():
    lock(mut)
    a_readers -= 1
    if !a_readers && p_writers:
        write_cond.signal()
    unlock(mut)

writer_lock():
    lock(mut)
    while a_writers + a_readers:
        p_writers += 1
        write_cond.wait(mut)
        p_writers -= 1
    a_writers += 1
    unlock(mut)

writer_unlock():
    lock(mut)
    a_writers -= 1
    if p_writers:
        write_cond.signal()
    else if p_readers:
        read_cond.broadcast()
    unlock(mut)
```
Reader-Writer Lock v3

Tracing a reader, then a writer

```c
int a_readers, a_writers, p_readers, p_writers  // Active & pending
mutex mut, cond_var read_cond, write_cond

reader_lock():
    lock(mut)
    while a_writers + p_writers:
        p_readers += 1
        read_cond.wait(mut)
        p_readers -= 1
        a_readers += 1
    unlock(mut)

reader_unlock():
    lock(mut)
    a_readers -= 1
    if !a_readers & p_writers:
        write_cond.signal()
    unlock(mut)

writer_lock():
    lock(mut)
    while a_writers + a_readers:
        p_writers += 1
        write_cond.wait(mut)
        p_writers -= 1
        a_writers += 1
    unlock(mut)

writer_unlock():
    lock(mut)
    a_writers -= 1
    if p_writers:
        write_cond.signal()
    else if p_readers:
        read_cond.broadcast()
    unlock(mut)
```
Reader-Writer Lock v3

Tracing a reader, a writer, a second writer, then a second reader.

```c
int a_readers, a_writers, p_readers, p_writers  // Active & pending
mutex mut, cond_var read_cond, write_cond

reader_lock():
    lock(mut)
    while a_writers + p_writers:
        p_readers += 1
        read_cond.wait(mut)
        p_readers -= 1
        a_readers += 1
    unlock(mut)

read_unlock():
    lock(mut)
    a_readers -= 1
    if !a_readers && p_writers:
        write_cond.signal()
    unlock(mut)

writer_lock():
    lock(mut)
    while a_writers + a_readers:
        p_writers += 1
        write_cond.wait(mut)
        p_writers -= 1
        a_writers += 1
    unlock(mut)

writer_unlock():
    lock(mut)
    a_writers -= 1
    if p_writers:
        write_cond.signal()
    else if p_readers:
        read_cond.broadcast()
    unlock(mut)
```
Reader-Writer Lock v3

Tracing a reader, a writer, a second writer, then a second reader.

```c
int a_readers, a_writers, p_readers, p_writers // Active & pending
mutex mut, cond_var read_cond, write_cond

reader_lock():
    lock(mut)
    while a_writers + p_writers:
        p_readers += 1
        read_cond.wait(mut)
        p_readers -= 1
        a_readers += 1
    unlock(mut)

reader_unlock():
    lock(mut)
    a_readers -= 1
    if !a_readers && p_writers:
        write_cond.signal()
    unlock(mut)

writer_lock():
    lock(mut)
    while a_writers + a_readers:
        p_writers += 1
        write_cond.wait(mut)
        p_writers -= 1
        a_writers += 1
    unlock(mut)

writer_unlock():
    lock(mut)
    a_writers -= 1
    if p_writers:
        write_cond.signal()
    else if p_readers:
        read_cond.broadcast()
    unlock(mut)
```

Choose priority here
Pthread Reader-Writer Lock

- **Type:** `pthread_rwlock_t`
  - (Rough C++17 equivalent: `std::shared_mutex`)

- **Initialization:**
  ```
  int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
  ```

- **Lock for read:**
  - **Blocking:**
    ```
    int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
    ```
  - **Nonblocking:**
    ```
    int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
    ```

- **Lock for write**
  - **Blocking:**
    ```
    int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
    ```
  - **Nonblocking:**
    ```
    int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
    ```
C++ Reader-Writer Lock

- Up to C++11 inclusive: no dice
- C++14: `std::shared_timed_mutex`
  - Exclusive Locking
    - `lock()`, `try_lock()`, `try_lock_for()`, `unlock()`
  - Shared Locking
    - `lock_shared()`, `try_lock_shared()`,
      `try_lock_shared_for()`, `unlock_shared()`
- C++17: `std::shared_mutex`
  - Same, but without timeouts
Performant Hashable Revisited

• Let’s wrap `std::unordered_map`, and not worry about the underlying implementation.

• In practice, would want to use a data structure we are sure does not mutate state in “read”-only operations: e.g. by making sure methods we think are read-only are tagged `const`.

• `std::unordered_map<K, V>` important methods:
  • `insert(std::pair<K, V>)`
  • `find(const K&)`
  • `operator[](const K&)`
  • `erase(const K&)`
Performant Hashtable Revisited

```cpp
#include <shared_mutex>
#include <unordered_map>

template<typename K, typename V>
template<typename K, typename V>
class ParallelUnorderedMap {
    std::shared_mutex rw_lock_;  
    std::unordered_map<K, V> map_;   
public:
    ParallelUnorderedMap() {}  
    bool insert(std::pair<K, V>);  
    const std::pair<bool, V> find(const K&);  // why copy?
    bool erase(const K&);
    // V& operator[](const K&);  // why not?
};
```

Why these methods? Where are the iterators? Where’s the PUM[K] operator?
Performant Hashtable Revisited

template<
typename K, typename V>
bool ParallelUnorderedMap::insert(std::pair<K, V> kv_pair) {
    rw_lock_.lock();  // exclusive: writer
    const auto& rval = map_.insert(kv_pair);
    rw_lock_.unlock();
    return rval.second;
}

A few comments on this...
Performant Hashtable Revisited

template<typename K, typename V>
const std::pair<bool, V> ParallelUnorderedMap::find(const K& k) {
    rw_lock_.lock_shared(); // shared: reader
    const std::unordered_map<K, V>::iterator it = map_.find(k);
    std::pair<bool, V> rval = {
        it != map_.end(), (it != map.end()) ? it->second : V()};
    rw_lock_.unlock_shared();
    return rval;
}

A few comments on this, especially lock ordering...
Performant Hashtable Revisited

```cpp
template<typename K, typename V>
bool ParallelUnorderedMap::erase(const K & k) {
    rw_lock_.lock();  // exclusive: writer
    const size_t rval = map_.erase(k);
    rw_lock_.unlock();
    return rval;
}
```

Few comments on this...
Conclusion

• Every time we see a structure taking many readers, R/W seem the thing to do.

• However, not a magic bullet. Peering into implementations:
  • Even in the reader-only case, there could be contention on the reader counter mutex.
  • Maintaining fairness can cause contention
  • Still more expensive than uncontested locks

• Recent works:
  • “Scalable Reader-Writer Locks”, Lev et al., 2009
  • “NUMA-Aware Reader-Writer Locks”, Calciu et al., 2013
  • “Scalable Read-Mostly Synchronization Using Passive Reader-Writer Locks”, Liu et al., 2014
Conclusion

- Homework 2 Questions
- Lab 1 Questions