Professor: How do you like mutexes?

Student: Oh hi Ben! Are they going to be on the exam?

Professor: Absolutely! They will also follow you throughout your life :)

Student: Well, it's not too bad. After all, it's just lock and unlock.

Professor: Yeah, that's what I thought too until the students kept missing all the locking questions.

Student: Uh oh... I am struggling a bit with the next programming assignment.

Professor: Yeah, it's funny that such a simple construct as a mutex can be so difficult to use..

Student: I'm not sure funny is the correct word here.

Professor: I think I've come up with a set of rules you can follow to more easily use locks.

Student: Hey wait, this is starting to sound like a chapter intro.

Professor: Yep, it is.

Rules for locking

Whenever we have multiple threads accessing shared data, we need some form of mutual exclusion or atomic operations to deal with the race conditions that arise. In practice, there are a couple of other widespread methods of dealing with race conditions: 1) ignore them[1] or 2) add sleeps or logic to make them less likely.

We have spent the last couple of chapters covering locks. Locks are simple constructs: you can take a lock, and you can release a lock, but we have already seen some of the problems we can run into if we do not use them properly. Of the problems that arise from race conditions, deadlocks can be the worst since they are noticeable. Miscalculated data or data that is only a little inconsistent often goes by unnoticed and may later be replaced by valid data. Even crashes can be "remedied" by automatically restarting programs. Deadlocks stop everything. They result in service outages and very noticeable problems.

Because of these problems, programmers often resort to running fast-and-loose and skipping locks altogether, hoping that the races work out in their favor; after all, there is some probability that everything will be okay. At some point, a mature program will need to get the race conditions addressed. (Tragically, this is often after the "rockstar" programmer who wrote the code has moved on to bigger and better things.) Retroactively fixing infrequent race conditions is difficult to test precisely because they are infrequent and thus require extensive repeated testing to ensure that the condition is indeed fixed.

This chapter presents some simple and easily applied rules that can ensure that you properly address race conditions. We give them names so that the conditions that violate the rules can easily be identified in code reviews. These rules help engineers understand where to use locking in their code. How to lock and rules for avoiding deadlocks are well explained by many books and articles, but when an engineer is presented with a pile of code and asked to make it threadsafe, there is a void of information usually filled by experience from the school of hard knocks.

In this chapter, we use the terms shared state and shared data interchangeably. The terms may refer to a single global variable, an object in memory, or a complex data structure. To protect them using locks (mutexes) we can treat them all the same way. Some simple primitives, such as integers, can be shared in a threadsafe way using atomic objects as long as they are involved in only relatively simple operations. Some data structures can use wait-free algorithms, but they are outside the scope of this chapter. In this chapter we focus on using mutexes.

- **fundamental rule of locking**: All mutable shared state must be protected with locks.
- **consistency rule**: If a lock is protecting state in one code block, it must be protected with the same lock in all code blocks.

[1] this guest author is ashamed to confess that early in his career, some of his code shipped with debug enabled to "make sure it worked properly". this chapter will make sure you never have to resort to that!

- **stable lock rule**: Locks used to protect state must completely overlap the time state needs to be threadsafe.
- **atomicity rule**: State changes that need to appear atomic must be under the locks to protect the state from the first access of any shared state to the last access.
- **locked block** *practice*: Lock-protected code is more readable and maintainable if acquisition and release happen in the same code block.
- **quick lock** *practice*: Minimize the amount of work done under a lock. Avoid blocking calls!
- **recursive locks** *religious*: Religious zeal surrounds recursive locks. Understand the religious context you are working in and conform to it.

You may find code that breaks these rules with awesome results or awesomely bad results. Usually, the awesome code is written by someone who knows the rules well and can break the rules awesomely.

For a running example, let's take a super simple example of a class with some state:

```
typedef struct account_s {
    HashMap_t *subtotals; // do not access directly (private)
    long total; // do not access directly (private)
    long min_total; // min total required for the account
    long max_total; // max total allowed in the account
} account_t;

// initialize an account with the given min and max totals
// can only be called once and must be called when an account is created.
void init_account(account_t *acct, long min, long max);

// add delta to the subtotal for the given key.
// create a subtotal with the delta for the key if not present already.
// if check is true, the addition will not happen but the min/max checks will
bool add_to_subtotal(account_t *acct, const char *key, int delta, bool check);

// return true if key is present. subtotal and total will contain the updated values
bool get_subtotal(const account_t *acct, const char *key, int *subtotal, int *total);

// print the subtotals and total for the account to the filehandle
void print_account(const account_t *acct, FILE *fh);

// transfer delta for subtotal with the given key from one account to the other.
void transfer_account(account_t *from, account_t *to, const char *key, int delta);
```

Not only do we want the `subtotals` to be thread-safe, but we also want to make sure that the total stays consistent with the sum of the subtotals. In other words, we want updates to the `total` and `subtotals` to be atomic.

Fundamental rule of locking

> "All mutable shared state must be protected with locks."

When using locks to make code threadsafe, we only need to worry about mutable state that can be accessed by multiple threads. Thus, shared constants or data that is initialized before it is accessed by multiple threads and never changes does not need to be protected. Also, state made threadsafe with other methods, such as atomic variables or wait-free data structures, may not need to be protected with a lock; although, if those variables and data structures are part of an atomic operation, you may still need to use locks.

What's the problem? It is not always obvious what state needs to be protected by locks. Adding locks to state that is not accessed by multiple threads will result in unnecessary lock overhead. Not adding locks

for data that is accessed by multiple threads will result in race conditions.

How we do it. To identify data to be locked, start by identifying variables or member variables that may be accessed by multiple threads. For `account_t` all the fields could be accessed directly, but comments indicate that only `min_total` and `max_total` should be.

Next, identify all of the functions that may be used by multiple threads concurrently for a given account. That list will include all of the account functions except for `init_account` since an account must be initialized before it can be used for anything else. Only `init_account` can change `min_total` and `max_total` and will execute before any other threads concurrently access an account. Thus `min_total` and `max_total` are effectively read-only, which means they do not need to be protected by a lock. The other functions do access and may change `total` and `subtotals` concurrently so that shared state must be protected.

How we review it. When reviewing code that is supposed to be threadsafe, you should first apply the reviewing technique for any existing protected state. For new state or existing state that may be exposed in new ways, use the analysis in the previous section to identify data that may potentially be accessed by multiple threads directly or through a function. Any identified data that is not protected should be marked as needing a lock and the author should provide an explanation why multiple threads cannot access the data (hopefully in the form of a comment in the code) or add locking.

Consistency rule

> "If a lock is protecting state in one code block, it must be protected with the same lock in all code blocks."

Mutual exclusion (mutex) means that only a single thread will have access to some state at a given time; all other threads are excluded. However, the kernel and locking libraries only enforce that a single thread will hold a lock. The programmer defines the relationship between state being locked and the locks themselves.

What's the problem? Applying the procedure mentioned above to find data to be protected by a lock will help identify locations to put locks to protect access to data. However, it does not say how many locks to use or which locks will protect which data. Also, you might miss some cases, or new code may get added that will also require a locking.

This rule provides a nice way to review code and ensure that locks are correctly placed by identifying data currently protected by a lock and making sure that any other locations where that data is accessed are also protected by the same lock.

How we do it. First, we determine the granularity of locking we want to do. A course grain lock will cover more states and is generally easier to use and analyze since there are fewer locks, but it can also result in more lock contention because a single lock protects more state. A fine grain locks will usually result in better performance, but the code may be more difficult to write and analyze since there are more locks. No matter the granularity, the approach is the same:

1. Choose the data that you want to protect with a single lock.
2. Pick where the lock will be stored. If the data you are protecting are stored as members of a structure, it makes sense to make the lock protecting that data a member of that structure. If using a language with native locking support, such as Java, it may make sense to use the object containing the data as members as the lock.
3. Go through your code and ensure that whenever you access any of the data you are protecting with the lock holds the lock before the access (whether read or write) is done.
4. Repeat this process until all of the data you identified with the consistency rule has been protected with locks.

There are a couple of exceptions to this locking rule. We have already mentioned the first one: Code that initializes the state before it is accessible by other threads need not be locked. Also, code that finalizes the state after sharing has ended need not be locked. In practice, it can be tricky to ensure you are in that state or that later code changes preserve this property.

Stable lock rule

> "Locks used to protect state must completely overlap the time state needs to be threadsafe."

This rule is a specific case of the consistency rule, but we call it out because it can easily be broken without realizing it. Here are some common examples of violations of the consistency rule.

Local locks

```
bool add_to_subtotal(const account_t *acct, const char *key, int delta, bool check) {
    bool rc = true;
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_lock(&mutex);
    long new_total = acct->total + delta;
    if (new_total < acct->min_total || new_total > acct->max_total) rc = false;
    else if (!check) {
        acct->total = new_total;
        HashMap_add(&acct->subtotals, key, delta);
    }
    pthread_mutex_unlock(&mutex);
    return true;
}
```

This code is clearly broken. A superficial examination of the code appears to satisfy the consistency rule, but in reality, every time the state is accessed, it is protected by a different, newly instantiated lock. All threads will be able to get the lock whenever this function is called since each invocation will have a fresh acquired lock.

Unstable locks

Perhaps in the `HashMap_t` structure, there is a mutex that could be used to protect both `subtotals` and `total`.

```
bool add_to_subtotal(const account_t *acct, const char *key, int delta, bool check) {
    bool rc = true;
    pthread_mutex_lock(&acct->subtotals->mutex);
    long new_total = acct->total + delta;
    if (new_total < acct->min_total || new_total > acct->max_total) rc = false;
    else if (!check) {
        acct->total = new_total;
        HashMap_add(&acct->subtotals, key, delta);
    }
    pthread_mutex_unlock(&acct->subtotals->mutex);
    return true;
}
```

> **Note:** Be careful when using mutexes from other objects. If we are using a mutex from HashMap, we should be sure that it is part of the public API and understand any caveats to using it. For example, we need to make sure that it is a recursive mutex!

This code works assuming that the mutex in subtotals does not get changed, but let's see what happens if we need to change `add_to_subtotal` to resize the hashtable. Let's assume we have the functions `HashMap_full`, which returns `true` if the hashtable is full, and `HashMap_double`, which will copy a hashtable into a new hashtable with twice the capacity. We can then incorporate them into `add_to_subtotal` as:

```
bool add_to_subtotal(const account_t *acct, const char *key, int delta) {
    bool rc = true;
    pthread_mutex_lock(&acct->subtotals->mutex);
    long new_total = acct->total + delta;
    if (new_total < acct->min_total || new_total > acct->max_total) rc = false;
    else if (!check) {
        if (HashMap_full(acct->subtotals)) {
            acct->subtotals = HashMap_double(&acct->subtotals);
        }
        acct->total = new_total;
        HashMap_add(&acct->subtotals, key, delta);
    }
    pthread_mutex_unlock(&acct->subtotals->mutex);
    return true;
}
```

This small change adds only a single line, but the new hashtable returned from `HashMap_double` probably has a new mutex as well. This means that we are locking on an unstable lock: we are using the same mutex pointer variable, but the variable has changed what it is pointing to.

What's the problem? To achieve mutual exclusion for a particular piece of data or set of data, all threads accessing the data must use the same stable lock. If two or more different locks are used to protect the data and different invocations of code use different locks to access the data, two threads will be able to access the data and violate mutual exclusion.

For mutual exclusion and, thus, thread safety to work, the same stable lock must be used throughout the lifetime of the data. This means that the lock must be created before multiple threads start accessing the data and cannot be destroyed or changed until the data is no longer accessible by multiple threads. Keep in mind that a lock is a particular mutex, not the pointer or variable that references the mutex! As we saw in the Unstable locks section above, the same variable may refer to two different mutexes, which will cause problems.

How we do it. We have two tasks to do. First, we need to make sure that a given piece of data is consistently protected by the same stable lock. Second, we need to make sure that a given lock doesn't change during its lifetime.

To make sure that data is consistently protected by the same stable lock:

1. Choose the data that we want to ensure is consistently locked. For example, we may pick the `subtotals` hashtable. Note: this data will often also have a context associated with it. For example, `subtotals` is associated with a particular `account_t` object.
2. We find all places where the data is accessed for reading or writing and make sure that it is being locked with the same stable lock in each of those places. For example, `subtotals` for a particular `account_t` object is locked with the `subtotals->mutex` lock.
3. Repeat this process for all of the protected data.

To make sure that a lock doesn't change during its lifetime, only set the mutex once in the code, usually during creation.

How we review it. You only need to apply the second step of the "How we do it" process to each reference of shared data in the code. Simply search for other examples of access to the shared data and see what locks

are being used. If a different lock is being used in other places in the code, have the author review the locking. Sometimes, as described in the next section, data may be accessed in a piece of code with multiple locks held. You will need to analyze that code to figure out which of the held locks is protecting which pieces of data.

Atomicity rule

> "State changes that need to appear atomic must be under the locks to protect the state from the first access of any shared state to the last access."

Often, atomic updates involve a single lock. For example, up until now, we have only used a single lock to protect both `total` and `subtotals`. Single locks can provide atomicity as long as the lock is held for the full duration of the atomic operation.

If we unlock and relock the mutex after setting the total, the code is threadsafe, but the update to total and subtotals are not atomic.

```
acct->total = new_total;
pthread_mutex_unlock(&acct->subtotals->mutex);
pthread_mutex_lock(&acct->subtotals->mutex);
HashMap_add(&acct->subtotals, key, delta);
```

In this code, the total may be updated, and another thread may acquire the lock before subtotals are updated. This can result in the thread seeing subtotals that do not match the total. By holding the lock for both updates, we avoid this problem.

> **Note:** Be careful when using condition variables with atomic operations. Remember that the mutex may be released and reacquired when waiting on a condition variable. Never put a condition variable wait in the middle of an atomic operation.

Things get more complicated when multiple locks are involved. Consider the code below :

```
bool transfer_account(account_t *from, account_t *to, const char *key, int delta) {
    if (add_to_subtotal(to, key, delta, true) &&
        add_to_subtotal(from, key, -delta), true) {
        add_to_subtotal(to, key, delta, false);
        add_to_subtotal(from, key, delta, false);
        return true;
    }
    return false;
}
```

This code is threadsafe, but it is not atomic. You can imagine two transfers of 100 from account1 to account2, but only one can go through without taking the total of account1 below the minimum. The code above checks for that condition, and `add_to_subtotal` is threadsafe, but the checks and the changes are not atomic: another thread can be executing the same code on the same accounts and cause the destination account to receive a count that did not come out of the source account. We need to make this atomic.

What's the problem? When executing an atomic operation over some data, we need to make sure that we have exclusive access to all the data we are operating for the whole transaction.

How we do it.

1. Identify all the data protected by locks that are involved in the atomic operation.
2. Take all the locks that protect the data before starting the operation. **If there are multiple locks, make sure to use a well-defined lock ordering.**
3. Release the locks after the atomic operation completes.

How we review it. For the code review, go through the same steps in "How we do it." and make sure the locks are held the entire time.

Locked block practice

> Lock-protected code is more readable and maintainable if acquisition and release happen in the same code block.

In Java this happens naturally if the synchronized keyword is used. For Java locks, try-with-resource blocks can be used. Lock guards can be used in C++.

What's the problem? By this point, you've probably begun to realize that locks are tricky to use correctly. It benefits everyone (the original code author, the review, the maintainer, and the world in general) if the code clearly shows what the lock is protecting. Taking the lock in one piece of code and releasing it in another makes it very difficult to analyze what is covered by the lock and if it is being used correctly.

We also have to worry about code blocks that return prematurely while holding a lock. The following code pattern complicates lock analysis:

```
// this is horrible code. don't do this!
if (protected) pthread_mutex_lock(&mutex);
if (total + delta > max_total) {
   return false;
} else {
   total += delta;
   rc = true;
}
if (rc && protected && !exit_with_lock) pthread_mutex_unlock(&mutex);
return rc;
```

Is the code above correct? Who knows? It's terrible to read. It does have an early return problem that will sometimes exit holding the lock.

How we do it. The lock and unlock should be in the same block of code, and the lock and unlock should occur in pairs. In more advanced languages like Java and C++, this can happen automatically. Java using `synchronized` blocks:

```
    synchronized (mutex) {
        if (total + delta > max_total) return false;
        total += delta;
        return true;
    }
```

or C++

```
    {
        const std::lock_guard<std::mutex> _(mutex);
        if (total + delta > max_total) return false;
        total += delta;
        return true;
    }
```

With these code blocks it is impossible to do the conditional locking above, but that kind of conditional locking makes it very difficult to analyze.

How we review it. This practice tries to make code easier to analyze, so if reviewing is hard, the code probably needs to be fixed.

Here is a quick checklist:

- Are there any returns in the middle? Early returns are okay if Java or C++ lock blocks are used, but manual lock/unlock pairs should not have returns in the middle.
- Is manual lock/unlock being done with languages that have exceptions, such as Java or C++? Make sure to put the unlock in a try/finally block. Better yet, use lock guards.

Quick lock practice

> Minimize the amount of work done under a lock. Avoid blocking calls!

Because locks restrict the number of threads that can access certain data, we need to make the amount of time a thread holds the lock as little as possible. Amdahl's Law shows that we must minimize the amount of serializing threads using locks if we want to get good speed using parallel processing. Lock overhead also goes up as lock contention increases. All of this motivates us to minimize the amount of work we do while holding a lock.

What's the problem? It's tempting to simply lock whole functions to ensure that the function is threadsafe. Java even encourages this behavior by providing a `synchronized` modifier on method declarations. It's better to analyze the code inside of a lock and make sure none of that work can be done before or after the lock is taken or released.

There is an important source of performance problems to avoid while holding a lock: I/O. Waiting for input is obviously bad while holding a lock since the user may never input anything, and the lock will be held forever. A similar thing can happen if you are doing something as simple as a `printf()`. Outputting characters to the screen has a non-trivial amount of overhead, but even worse, the screen may be paused with `Ctrl-S` or on a stuck network session. Do not do network I/O while holding a lock. The only exception to this rule is if you are using the lock specifically to guarantee exclusive use of an I/O resource.

There are other smaller things that you can watch out for, such as doing allocations or setting up a data structure for a subsequent call. In the end, each of these may be quite small, but ideally, what you are doing inside the lock is also quite small, so relatively, the savings may be quite large.

How we do it.

1. Are there big operations inside the lock, such as I/O or a bulk computation that can be broken up so that part of it can be done before or after the lock?
2. Go through each line inside the lock section. If it doesn't involve data protected by the lock, you should be able to move it before you take the lock. If the line doesn't involve protected data and is not later used by protected data, move it out of the lock.
3. Are you still doing too much inside the lock? Sometimes, you might make copies of data and move the processing outside.

Let's apply this to the implementation of `print_account`:

```
void print_account(const account_t *acct, FILE *fh) {
    pthread_mutex_lock(&acct->subtotals->mutex);
    fprintf(fh, "total: %ld\n", acct->total);
    int count;
    HashMap_entry_t *entries = HashMap_get_entries(acct->subtotals, &count);
    for (int i = 0; i < count; i++) {
        fprintf(fh, "%s\t%ld\n", entries[i].name, entries[i].amount);
    }
    free(entries);
    pthread_mutex_unlock(&acct->subtotals->mutex);
}
```

This code is threadsafe, but it violates the quick lock practice. There is a lot of I/O going on, and `entries` is not protected by the lock. We can drastically reduce the amount of code covered by the lock by applying the rules:

```
void print_account(const account_t *acct, FILE *fh) {
    int count;
    long atotal;
    pthread_mutex_lock(&acct->subtotals->mutex);
    atotal = acct->total;
    HashMap_entry_t *entries = HashMap_get_entries(acct->subtotals, &count);
    pthread_mutex_unlock(&acct->subtotals->mutex);
    fprintf(fh, "total: %ld\n", atotal);
    for (int i = 0; i < count; i++) {
        fprintf(fh, "%s\t%ld\n", entries[i].name, entries[i].amount);
    }
    free(entries);
}
```

This updated code not only minimizes the amount of work done by the CPU, but it also makes the reviewer's and maintainer's job easier since there are only two lines to check inside the lock.

How we review it. If the author of the change has done their work, you should be able to quickly go through the "How we do it" steps and see that there is nothing more to do.

Religious recursive locks

> Religious zeal surrounds recursive locks. Understand the religious context you are working in and conform to it.

When locking, you need to choose between recursive and non-recursive locks. The difference between the two is that recursive locks can be acquired twice (or more) before being released if acquired both times by the same thread; the lock has to be released the same number of times before it is actually released. Non-recursive locks will block (causing a self-deadlock) or return an error if the same thread tries to acquire a lock the second time.

Some recursive religious zealots declare that rightness dictates to always use recursive locks since it's silly to deadlock on yourself. Other non-recursive zealots declare that rightness dictates that you write your code rightly to never try to take a lock you already have!

If you are in a code base of non-recursive practitioners, you need to watch for functions that assume a lock is held. They sometimes follow a particular naming convention. If you write a function that may need to sometimes be called by code holding a lock and other times by code not holding a lock, you will write a function assuming the lock is held and then write the locking version that acquires the lock and calls the locked version. Pay particular attention to the consistency locking rule to make sure the right locks are held when calling the unlocked version.

You will note that in our running example, we have assumed recursive locks. We locked the same mutex that is used by the hashmap. If the mutex were not recursive, we would deadlock when we invoked the hashmap functions.

Summary

With discipline and organization, you can safely and efficiently use locks. As you master these rules and develop some of your own, you will also find opportunities to break them at the right time with excellent results. My experience has been that more often, you will convince yourself and your coworkers that you are in an exceptional case where these rules don't apply only to be hunting down the 1 in a million requests race condition that teaches you why the rule does actually apply.

You will also want to master atomic objects and wait-free data structures that can provide more performance but also come with their own fun practices to learn.