Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses[1,2]

Chapter 8
Shared-Memory Concurrency Control with a Data-Structures Focus

Dan Grossman

University of Washington

djg@cs.washington.edu

[1]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st Edition, Morgan Kaufmann, ISBN : 9780128038994, Pages: 360.

[2]Free preprint version of the CDER book: `http://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book`.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 8

SHARED-MEMORY CONCURRENCY CONTROL WITH A DATA-STRUCTURES FOCUS

Dan Grossman

University of Washington

djg@cs.washington.edu

ABSTRACT

This chapter is a companion to the previous chapter. It introduces the need for concurrency control (synchronization) when threads are accessing shared resources, particularly shared memory. It presents the need for mutual-exclusion locks and how to use them correctly. The presentation focuses on the concept of locking first and then the details of locks (via synchronized statements and methods) in Java. It then draws a distinction between *bad interleavings* — when observable intermediate states cause a software component in a concurrent setting not to meet its specification — from *data races*, in which modern platforms' relaxed memory-consistency models mean even seemingly correct programs are in fact wrong. Given this basic definition of what concurrency control is when it is needed, the chapter then presents a series of guidelines for how to use locks correctly and effectively. The chapter finishes with several topics not previously introduced, including deadlock, reader/writer locks, and condition variables.

Relevant core courses: Data Structures and Algorithms, Second Programming Course in the Introductory Sequence

Relevant PDC topics: Shared memory, Language extensions, Libraries, Task/thread

spawning, Synchronization, Critical regions, Concurrency defects, Memory models, Non-determinism

Learning outcomes: Students mastering the material in this chapter should be able to:

- Use locks to implement critical sections correctly.
- Identify the need for critical sections and the incorrect results that can arise if a lock is omitted or the wrong lock is used in a concurrent program.
- Distinguish data races as a distinct notion from a bad interleaving even though both arise from too little synchronization.
- Follow basic guidelines for easier concurrent programming, such as avoiding mutable thread-shared state where possible and following consistent protocols for what locks protect what data.
- Define deadlock and explain why a consistent order on lock acquisition avoids it.
- Explain the purpose of reader/writer locks and condition variables.

Context for use: This chapter and the previous one complement each other. Both are designed to be used in an intermediate-advanced data-structures courses — the course that covers asymptotic complexity, balanced trees, hash tables, graph algorithms, sorting, etc., though some of the material has also been used in second programming courses. While the previous chapter focuses on parallelism for efficiency, this chapter focuses on concurrency control, predominantly via *mutual-exclusion locks*.

8.1 Introduction

This chapter complements the previous one in terms of the conceptual distinction between parallelism and concurrency control presented in Section **??**. The previous chapter focused on fork-join parallelism using the shared-memory programming model. This chapter uses the same programming model, but leaves fork-join parallelism behind to focus on concurrency control, which is about correctly and efficiently controlling access by multiple threads to shared resources.

We will still have threads and shared memory. We will use Java's built-in threads (`java.lang.Thread`). The "shared resources" will be memory locations (fields of objects) used by more than one thread. We will learn how to write code that provides properly synchronized access to shared resources even though it may not be known what order the threads may access the data. In fact, multiple threads may *try* to access and/or modify the data at the same time. We will see when this cannot be allowed and how programmers must use programming-language features to avoid it. The features we will focus on involve *mutual-exclusion locks*, introduced in Section 8.3.2.

Here are some simple high-level examples of shared resources where we need to control concurrent access:

1. We are writing banking software where we have an object for each bank account. Different threads (e.g., one per bank teller or ATM) deposit or withdraw funds from various accounts. What if two threads try to manipulate the same account at the same time?

2. We have a hashtable storing a cache of previously-retrieved results. For example, maybe the table maps patients to medications they need to be given. What if two threads try to insert patients — maybe even the same patient — at the same time? Could we end up with the same patient in the table twice? Could that cause other parts of the program to indicate double the appropriate amount of medication for the patient?

3. Suppose we have a *pipeline*, which is like an assembly line, where each *stage* is a separate thread that does some processing and then gives the result to the subsequent stage. To pass these results, we could use a queue where the result-provider enqueues and the result-receiver dequeues. So an n-stage pipeline would have $n-1$ queues; one queue between each pair of adjacent stages. What if an enqueue and a dequeue happen at the same time? What if a dequeue is attempted when there are no elements in the queue (yet)? In sequential programming, dequeueing from an empty queue is typically an error. In concurrent programing, we may prefer instead to *wait* until another thread enqueues something.

8.2 The Programming Model

As the examples above hint at, in concurrent programming we have multiple threads that are "largely doing their own thing" but occasionally need to coordinate since they are accessing shared resources. It is like different cooks working in the same kitchen — easy if they are using different utensils and stove burners, but more difficult if they need to share things. The cooks may be working toward a shared goal like producing a meal, but while they are each working on a different recipe, the shared goal is not the focus.

In terms of programming, the basic model comprises multiple threads that are running in a mostly uncoordinated way. We might create a new thread when we have something new to do. The operations of each thread are *interleaved* (running alongside, before, after, or at the same time) with operations by other threads. This is very different from fork-join parallelism. In fork-join algorithms, one thread creates a helper thread to solve a specific subproblem and the creating thread waits (by calling join) for the other thread to finish. Here, we may have 4 threads processing bank-account changes as they arrive. While it is unlikely that two threads would access the same account at the same time, it is possible and we must be correct in this case.

One thing that makes these sort of programs very difficult to debug is that the bugs can be very unlikely to occur. If a program exhibits a bug and you re-run the program

another million times with the same inputs, the bug may not appear again. This is because what happens can depend on the order that threads access shared resources, which is not entirely under programmer control. It can depend on how the threads are *scheduled* onto the processors, i.e., when each thread is chosen to run and for how long, something that is decided *automatically* (when debugging, it often feels *capriciously*) by the implementation of the programming language, with help from the operating system. Therefore, how a program executes is *nondeterministic* and the outputs may not depend only on the inputs but also on scheduling decisions not visible to or controllable by the programmer. Because testing concurrent programs is so difficult, it is exceedingly important to design them well using well-understood design principles (see Section 8.5) from the beginning.

As an example, suppose a bank-account class has methods `deposit` and `withdraw`. Suppose the latter throws an exception if the amount to be withdrawn is larger than the current balance. If one thread deposits into the account and another thread withdraws from the account, then whether the withdrawing thread throws an exception could depend on the order the operations occur. And that is still assuming all of one operation completes before the other starts. In upcoming sections, we will learn how to use *locks* to ensure the operations themselves are not interleaved, but even after ensuring this, the program can still be nondeterministic.

As a contrast, we can see how fork-join parallelism as covered in the previous chapter made it relatively easy to avoid having two threads access the same memory at the same time. Consider a simple parallel reduction like summing an array. Each thread accesses a disjoint portion of the array, so there is no sharing like there potentially is with bank accounts. The sharing is with fields of the thread objects: One thread initializes fields (like the array range) before creating the helper thread. Then the helper thread may set some result fields that the other thread reads after the helper thread terminates. The *synchronization* here is accomplished entirely via (1) thread creation (not calling `start` or `fork` until the correct fields are written) and (2) `join` (not reading results until the other thread has terminated). But in this chapter, this form of synchronization is not applicable because we will not wait

for another thread to finish running before accessing a shared resource like a bank account.

It is worth asking why anyone would use this difficult programming model. It would certainly be simpler to have one thread that does everything we need to do. There are several reasons:

- *Parallelism:* Unlike in the carefully chosen algorithms in the previous chapter, it may well be that a parallel algorithm needs to have different threads accessing some of the same data structures in an unpredictable way. For example, we could have multiple threads search through a graph, only occasionally crossing paths.

- *Responsiveness:* Many programs, including operating systems and programs with user interfaces, want/need to respond to external events quickly. One way to do this is to have some threads doing the program's expensive computations while other threads are responsible for "listening for" events like buttons being clicked or typing occurring. The listening threads can then (quickly) write to some fields that the computation threads later read.

- *Processor utilization:* If one thread needs to read data from disk (e.g., a file), this will take a very long time relatively speaking. In a conventional single-threaded program, the program will not do anything for the milliseconds it takes to get the information. But this is enough time for another thread to perform millions of instructions. So by having other threads, the program can do useful work while waiting for I/O. This use of multithreading is called *masking (or hiding) I/O latency*.

- *Failure/performance isolation:* Sometimes having multiple threads is simply a more convenient way to structure a program. In particular, when one thread throws an exception or takes too long to compute a result, it affects only what code is executed by that thread. If we have multiple independent pieces of work, some of which might (due to a bug, a problem with the data, or some other reason) cause an exception or run for too long, the other threads can still continue executing. There are other approaches to these problems, but threads often work well.

It is common to hear that threads are useful *only* for performance. This is true only if "performance" includes responsiveness and isolation, which stretches the definition of performance.

8.3 Synchronization With Locks

8.3.1 The Need for Synchronization

This section presents in detail an initial example to demonstrate why we should prevent multiple threads from simultaneously performing operations on the same memory. We will focus on showing *possible interleavings* that produce the wrong answer. However, the example also has *data races*, which Section 8.4 explains must be prevented. We will show that using "ordinary" Java features to try to prevent bad interleavings simply does not work. Instead, we will learn to use *locks*, which are primitives provided by Java and other programming languages that provide what we need. We will not learn how to *implement* locks, since the techniques use low-level features more central to courses in operating systems and computer architecture.

Consider this code for a `BankAccount` class:

```
class BankAccount {
  private int balance = 0;
  int getBalance() {
    return balance;
  }
  void setBalance(int x) {
    balance = x;
  }
  void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
```

```
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
  }
  // ... other operations like deposit, etc.
}
```

This code is correct in a single-threaded program. But suppose we have two threads, one calls `x.withdraw(100)`, and the other calls `y.withdraw(100)`. The two threads could truly run at the same time on different processors. Or they may run one at a time, but the *thread scheduler* might stop one thread and start the other at any point, switching between them any number of times. Still, in many scenarios it is not a problem for these two method calls to execute concurrently:

- If `x` and `y` are not aliases, meaning they refer to distinct bank accounts, then there is no problem because the calls are using different memory. This is like two cooks using different pots at the same time.
- If one call happens to finish before the other starts, then the behavior is like in a sequential program. This is like one cook using a pot and then another cook using the same pot. When this is not the case, we say the calls *interleave*. Note that interleaving can happen even with one processor because a thread can be *pre-empted* at any point, meaning the thread scheduler stops the thread and runs another one.

So let us consider two interleaved calls to `withdraw` on the same bank account. We will "draw" interleavings using a vertical timeline (earlier operations closer to the top) with each thread in a separate column. There are many possible interleavings since even the operations in a helper method like `getBalance` can be interleaved with operations in other threads. But here is one incorrect interleaving. Assume initially the `balance` field holds 150 and both withdrawals are passed 100. Remember that each thread executes a different method call with its own "local" variables `b` and `amount` but the calls to `getBalance` and `setBalance` are, we assume, reading/writing the one `balance` field of the *same* object.

```
Thread 1                                Thread 2
--------                                --------
int b = getBalance();
// b holds 150
                                        int b = getBalance();
                                        if(amount > b)
                                          throw new ...;
                                        setBalance(b - amount); // sets balance to 50
if(amount > b) // no exception
  throw new ...;
setBalance(b - amount);
```

If this interleaving occurs, the resulting `balance` will be 50 and no exception will be thrown. But two `withdraw` operations were supposed to occur — there *should* be an exception. Somehow we "lost" a withdraw, which would not make the bank happy. The problem is that `balance` changed after Thread 1 retrieved it and stored it in `b`.

When first learning concurrent programming there is a natural but almost-always WRONG attempted fix to this sort of problem. It is tempting to rearrange or repeat operations to try to avoid using "stale" information like the value in `b`. Here is a WRONG idea:

```
void withdraw(int amount) {
  if(amount > getBalance())
    throw new WithdrawTooLargeException();
  // maybe balance changed, so get the new balance
  setBalance(getBalance() - amount);
}
```

The idea above is to call `getBalance()` a second time to get any updated values. But there is *still* the potential for an incorrect interleaving. Just because `setBalance(getBalance()`

`- amount)` is on one line in our source code does not mean it happens all at once. This code still (a) calls `getBalance`, then (b) subtracts `amount` from the result, then (c) calls `setBalance`. Moreover (a) and (c) may consist of multiple steps. In any case, the balance can change between (a) and (c), so we have not really accomplished anything except we might now produce a negative balance without raising an exception.

The sane way to fix this sort of problem, which arises whenever we have concurrent access to shared memory that might change (i.e., the contents are *mutable*), is to enforce *mutual exclusion*: allow only one thread to access any particular account at a time. The idea is that a thread will "hang a do-not-disturb sign" on the account before it starts an operation such as `withdraw` and not remove the sign until it is finished. Moreover, all threads will check for a "do-not-disturb sign" before trying to hang a sign and do an operation. If such a sign is there, a thread will *wait* until there is no sign so that it can be sure it is the only one performing an operation on the bank account. Crucially, the act of "checking there is no sign hanging and then hanging a sign" has to be done "all-at-once" or, to use the common terminology, *atomically*. We call the work done "while the sign is hanging" a *critical section*; it is critical that such operations not be interleaved with other conflicting ones. (Critical sections often have other technical requirements, but this simple informal definition will suffice for our purposes.)

Here is one WRONG way to try to implement this idea. You should never write code that tries to do this manually, but it is worth understanding why it is WRONG:

```
class BankAccount {
  private int balance = 0;
  private boolean busy = false;
  void withdraw(int amount) {
    while(busy) { /* spin-wait */ }
    busy = true;
    int b = getBalance();
    if(amount > b)
```

```
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
    busy = false;
  }
  // deposit would spin on same boolean
}
```

The idea is to use the `busy` variable as our do-not-disturb sign. All the other account operations would use the `busy` variable in the same way so that only one operation occurs at a time. The while-loop is perhaps a little inefficient since it would be better not to do useless work and instead be notified when the account is "available" for another operation, but that is not the real problem. The `busy` variable does not work, as this interleaving shows:

```
Thread 1                    Thread 2
--------                    --------
while(busy) { }
                            while(busy) { }
busy = true;
                            busy = true;
int b = getBalance();
                            int b = getBalance();
                            if(amount > b)
                              throw new ...;
                            setBalance(b - amount);
if(amount > b)
  throw new ...;
setBalance(b - amount);
```

Essentially we have the same problem with `busy` that we had with `balance`: we can check that `busy` is false, but then it might get set to true before we have a chance to set it

to true ourselves. We need to check there is no do-not-disturb sign *and* put our own sign up while *we still know there is no sign.* Hopefully you can see that using a variable to see if `busy` is busy is only going to push the problem somewhere else again.

To get out of this conundrum, we rely on *synchronization primitives* provided by the programming language. These typically rely on special hardware instructions that can do a little bit more "all at once" than just read or write a variable. That is, they give us the *atomic* check-for-no-sign-and-hang-a-sign that we want. This turns out to be enough to implement mutual exclusion properly. The particular kind of synchronization we will use is *locking.*

8.3.2 Locks

We can define a *mutual-exclusion lock*, also known as just a *lock* (or sometimes called a *mutex*), as an abstract datatype designed for concurrency control and supporting three operations:

- `new` creates a new lock that is initially "not held"
- `acquire` takes a lock and blocks until it is currently "not held" (this could be right away, i.e., not blocking at all, since the lock might already be not held). It sets the lock to "held" and returns.
- `release` takes a lock and sets it to "not held."

This is exactly the "do-not-disturb" sign we were looking for, where the `acquire` operation blocks (does not return) until the caller is the thread that most recently hung the sign. It is up to the lock implementation to ensure that it always does the right thing no matter how many acquires and/or releases different threads perform simultaneously. For example, if there are three acquire operations at the same time for a "not held" lock, one will "win" and return immediately while the other two will block. When the "winner" calls `release`, one other thread will get to hold the lock next, and so on.

The description above is a general notion of what locks are. Section 8.3.3 will describe how to use the locks that are part of the Java language. But using the general idea, we can write this PSEUDOCODE (Java does not actually have a `Lock` class), which is almost correct (i.e., still WRONG but close):

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();
  ...
  void withdraw(int amount) {
    lk.acquire(); /* may block */
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
    lk.release();
  }
  // deposit would also acquire/release lk
}
```

Though we show only the `withdraw` operation, remember that all operations accessing `balance` and whatever other fields the account has should acquire and release the lock held in `lk`. That way only one thread will perform any operation on a particular account at a time. Interleaving a `withdraw` and a `deposit`, for example, is incorrect. If any operation does not use the lock correctly, it is like threads calling that operation are ignoring the do-not-disturb sign. Because each account uses a different lock, different threads can still operate on different accounts at the same time. But because locks work correctly to enforce that only one thread holds any given lock at a time, we have the mutual exclusion we need for each account. If instead we used one lock for all the accounts, then we would still have mutual exclusion, but we could have worse performance since threads would be

waiting unnecessarily.

What, then, is wrong with the pseudocode? The biggest problem occurs if `amount > b`. In this case, an exception is thrown and therefore the lock is not released. So no thread will ever be able to perform another operation on this bank account and any thread that attempts to do so will be blocked *forever* waiting for the lock to be released. This situation is like someone leaving the room through a window without removing the do-not-disturb sign from the door, so nobody else ever enters the room.

A fix in this case would be to add `lk.release` in the branch of the if-statement before the throw-statement. But more generally we have to worry about any exception that might be thrown, even while some other method called from within the critical section is executing or due to a null-pointer or array-bounds violation. So more generally it would be safer to use a catch-statement or finally-statement to make sure the lock is always released. Fortunately, as Section 8.3.3 describes, when we use Java's locks instead of our pseudocode locks, this will not be necessary.

A second problem, also not an issue in Java, is more subtle. Notice that `withdraw` calls `getBalance` and `setBalance` while holding the lock for the account. These methods might also be called directly from outside the bank-account implementation, assuming they are public. Therefore, they should also acquire and release the lock before accessing the account's fields like `balance`. But if `withdraw` calls these helper methods *while holding the lock*, then as we have described it, the thread would block forever — waiting for "some thread" to release the lock when in fact the thread itself already holds the lock.

There are two solutions to this problem. The first solution is more difficult for programmers: we can define two versions of `getBalance` and `setBalance`. One would acquire the lock and one would assume the caller already holds it. The latter could be private, perhaps, though there are any number of ways to manually keep track in the programmer's head (and hopefully documentation!) what locks the program holds where.

The second solution is to change our definition of the `acquire` and `release` operations so that it is okay for a thread to (re)acquire a lock it already holds. Locks that support this

are called *reentrant locks*. (The name fits well with our do-not-disturb analogy.) To define reentrant locks, we need to adjust the definitions for the three operations by adding a *count* to track how many times the current holder has reacquired the lock (as well as the identity of the current holder):

- `new` creates a new lock with no current holder and a count of 0
- `acquire` blocks if there is a current holder *different from the thread calling it*. Else if the current holder is the thread calling it, do not block and increment the counter. Else there is no current holder, so set the current holder to the calling thread.
- `release` only releases the lock (sets the current holder to "none") if the count is 0. Otherwise, it decrements the count.

In other words, a lock is released when the number of `release` operations by the holding thread equals the number of `acquire` operations.

8.3.3 Locks in Java

The Java language has built-in support for locks that is different than the pseudocode in the previous section, but easy to understand in terms of what we have already discussed. The main addition to the language is a synchronized statement, which looks like this:

```
synchronized (expression) {
   statements
}
```

Here, `synchronized` is a keyword. Basically, the *syntax* is just like a while-loop using a different keyword, but this is *not* a loop. Instead it works as follows:

1. The `expression` is evaluated. It must produce (a reference to) an object — not `null` or a number. This object is treated as a lock. In Java, *every object is a lock* that any thread can acquire or release. This decision is a bit strange, but is convenient in an object-oriented language as we will see.

2. The synchronized statement acquires the lock, i.e., the object that is the result of step (1). Naturally, this may block until the lock is available. Locks are reentrant in Java, so the statement will not block if the executing thread already holds it.

3. After the lock is successfully acquired, the `statements` are executed.

4. When control leaves the `statements`, the lock is released. This happens either when the final } is reached *or* when the program "jumps out of the statement" via an exception, a `return`, a `break`, or a `continue`.

Step (4) is the other unusual thing about Java locks, but it is quite convenient. Instead of having an explicit release statement, it happens implicitly at the ending }. More importantly, the lock is still released if an exception causes the thread to not finish the `statements`. Naturally, `statements` can contain arbitrary Java code: method calls, loops, other synchronized statements, etc. The only shortcoming is that there is no way to release the lock *before* reaching the ending }, but it is rare that you want to do so.

Here is an implementation of our bank-account class using synchronized statements as you might expect. However, this example is UNUSUAL STYLE and is NOT ENCOURAGED. The key thing to notice is that any object can serve as a lock, so for the lock we simply create an instance of `Object`, the built-in class that is a superclass of every other class.

```
class BankAccount {
  private int balance = 0;
  private Object lk = new Object();
  int getBalance() {
    synchronized (lk) {
      return balance;
    }
  }
  void setBalance(int x) {
```

```
    synchronized (lk) {
      balance = x;
    }
  }
  void withdraw(int amount) {
    synchronized (lk) {
      int b = getBalance();
      if(amount > b)
        throw new WithdrawTooLargeException();
      setBalance(b - amount);
    }
  }
  // deposit and other operations would also use synchronized(lk)
}
```

Because locks are reentrant, it is no problem for `withdraw` to call `setBalance` and `getBalance`. Because locks are automatically released, the exception in `withdraw` is not a problem.

While it may seem naturally good style to make `lk` a `private` field since it is an implementation detail (clients need not care *how* instances of `BankAccount` provide mutual exclusion), this choice prevents clients from writing their own critical sections involving bank accounts. For example, suppose a client wants to double an account's balance. In a sequential program, another class could include this correct method:

```
  void doubleBalance(BankAccount acct) {
    acct.setBalance(acct.getBalance()*2);
  }
```

But this code is subject to a bad interleaving: the balance might be changed by another thread after the call to `getBalance`. If `lk` were instead `public`, then clients could use it — and the convenience of reentrant locks — to write a proper critical section:

```
  void doubleBalance(BankAccount acct) {
    synchronized(acct.lk) {
      acct.setBalance(acct.getBalance()*2);
    }
  }
```

There is a simpler and more idiomatic way, however: Why create a new `Object` and store it in a `public` field? Remember any object can serve as a lock, so it is more convenient, and the custom in Java, to use the instance of `BankAccount` itself (or in general, the object that has the fields relevant to the synchronization). So we would remove the `lk` field entirely like this (there is one more slightly shorter version coming):

```
class BankAccount {
  private int balance = 0;
  int getBalance() {
    synchronized (this) {
      return balance;
    }
  }
  void setBalance(int x) {
    synchronized (this) {
      balance = x;
    }
  }
  void withdraw(int amount) {
    synchronized (this) {
      int b = getBalance();
      if(amount > b)
        throw new WithdrawTooLargeException();
      setBalance(b - amount);
```

```
    }
  }
  // deposit and other operations would also use synchronized(this)
}
```

All we are doing differently is using the object itself (referred to by the keyword `this`) rather than a different object. Of course, all methods need to agree on what lock they are using. Then a client could write the `doubleBalance` method like this:

```
  void doubleBalance(BankAccount acct) {
    synchronized(acct) {
      acct.setBalance(acct.getBalance()*2);
    }
  }
```

In this way, `doubleBalance` acquires the same lock that the methods of `acct` do, ensuring mutual exclusion. That is, `doubleBalance` will not be interleaved with other operations on the same account.

Because this idiom is so common, Java has one more shortcut to save you a few keystrokes. When an entire method body should be synchronized on `this`, we can omit the synchronized statement and instead put the `synchronized` keyword in the method declaration before the return type. This is just a shorter way of saying the same thing. So our FINAL VERSION looks like this:

```
class BankAccount {
  private int balance = 0;
  synchronized int getBalance() {
    return balance;
  }
  synchronized void setBalance(int x) {
    balance = x;
```

```
  }
  synchronized void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
  }
  // deposit and other operations would also be declared synchronized
}
```

There is no change to the `doubleBalance` method. It must still use a synchronized statement to acquire `acct`.

While we have built up to our final version in small steps, in practice it is common to have classes defining shared resources consist of all synchronized methods. This approach tends to work well provided that critical sections access only the state of a single object. For some other cases, see the guidelines in Section 8.5.

Finally, note that the synchronized statement is far from the only support for concurrency control in Java. In Section 8.7.2, we will show Java's support for *condition variables*. The package `java.util.concurrent` also has many library classes useful for different tasks. For example, you almost surely should not implement your own concurrent hashtable or queue since carefully optimized correct implementations are already provided. In addition to concurrent data structures, the standard library also has many features for controlling threads. A standard reference (an entire book!) is, "Java Concurrency in Practice" by Brian Goetz et al. [1]

8.4 Race Conditions: Bad Interleavings and Data Races

A *race condition* is a mistake in your program (i.e., a bug) such that whether the program behaves correctly or not depends on the order that the threads execute. The name is a bit difficult to motivate: the idea is that the program will "happen to work" if certain threads

during program execution "win the race" to do their next operation before some other threads do their next operations. Race conditions are very common bugs in concurrent programming that, by definition, do not exist in sequential programming. This section defines two kinds of race conditions and explains why you must avoid them. Section 8.5 describes programming guidelines that help you do so.

One kind of race condition is a *bad interleaving*, sometimes called a *higher-level race* (to distinguish it from the second kind). Section 8.1 showed some bad interleavings and this section will show several more. The key point is that "what is a bad interleaving" *depends entirely on what you are trying to do.* Whether or not it is okay to interleave two bank-account withdraw operations depends on some specification of how a bank account is supposed to behave.

A *data race* is a specific kind of race condition that is better described as a "simultaneous access error" although nobody uses that term. There are two kinds of data races:

- When one thread might *read* an object field at the same moment that another thread *writes* the same field.
- When one thread might *write* an object field at the same moment that another thread also *writes* the same field.

Notice it is *not an error* for two threads to both *read* the same object field at the same time.

As Section 8.4.2 explains, *your programs must never have data races even if it looks like a data race would not cause an error — if your program has data races, the execution of your program is allowed to do very strange things.*

To better understand the difference between bad interleavings and data races, consider the final version of the `BankAccount` class from Section 8.1. If we accidentally omit the `synchronized` keyword from the `withdraw` definition, the program will have many bad interleavings, such as the first one we considered in Section 8.1. But since `getBalance` and `setBalance` are still `synchronized`, there are no data races: Every piece of code that reads or writes `balance` does so while holding the appropriate lock. So there can never be a

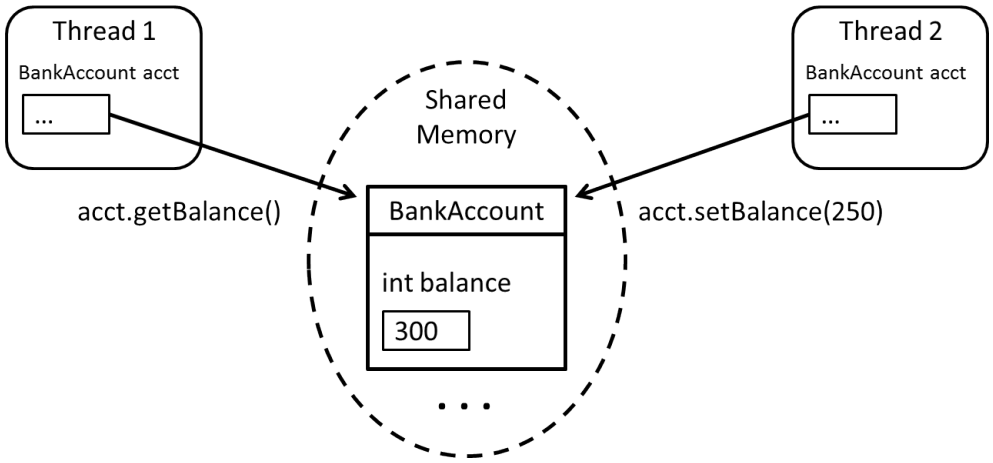

Figure (8.1) Visual depiction of a data race: Two threads accessing the same field of the same object, at least one of them writing to the field, without synchronization to ensure the accesses cannot happen, "at the same time."

simultaneous read/write or write/write of this field.

Now suppose `withdraw` is `synchronized` but we accidentally omitted `synchronized` from `getBalance`. Data races now exist and the program is wrong regardless of what a bank account is supposed to do. Figure 8.1 depicts an example that meets our definition: two threads reading/writing the same field at potentially the same time. (If both threads were writing, it would also be a data race.) Interestingly, simultaneous calls to `getBalance` do *not* cause a data race because simultaneous reads are not an error. But if one thread calls `getBalance` at the same time another thread calls `setBalance`, then a data race occurs. Even though `setBalance` is synchronized, `getBalance` is not, so they might read and write `balance` at the same time. If your program has data races, it is infeasible to reason about what might happen.

8.4.1 Bad Interleavings: An Example with Stacks

A useful way to reason about bad interleavings is to think about *intermediate states* that other threads must not "see." Data structures and libraries typically have *invariants* necessary for their correct operation. For example, a `size` field may need to store the correct

number of elements in a data structure. Operations typically violate invariants temporarily, restoring the invariant after making an appropriate change. For example, an operation might add an element to the structure *and* increment the size field. No matter what order is used for these two steps, there is an intermediate state where the invariant does not hold. In concurrent programming, our synchronization strategy often amounts to using mutual exclusion to ensure that no thread can observe an invariant-violating intermediate state produced by another thread. What invariants matter for program correctness depends on the program.

Let's consider an extended example using a basic implementation of a bounded-size stack. (This is not an interesting data structure; the point is to pick a small example so we can focus on the interleavings. Section 8.7.2 discusses an alternative to throwing exceptions for this kind of structure.)

```
class Stack<E> {
  private E[] array;
  private int index = 0;
  Stack(int size) {
    array = (E[]) new Object[size];
  }
  synchronized boolean isEmpty() {
    return index==0;
  }
  synchronized void push(E val) {
     if(index==array.length)
        throw new StackFullException();
     array[index++] = val;
  }
  synchronized E pop() {
     if(index==0)
```

```
        throw new StackEmptyException();
    return array[--index];
  }
}
```

The key invariant is that if `index>0` then `index-1` is the position of the most recently pushed item that has not yet been popped. Both `push` and `pop` temporarily violate this invariant because they need to modify `index` and the contents of the array. Even though this is done in one Java statement, it does not happen all at once. By making each method synchronized, no thread using these operations can see an incorrect intermediate state. It effectively ensures there is some global order of calls to `isEmpty`, `push`, and `pop`. That order might differ across different executions because scheduling is nondeterministic, but the calls will not be interleaved.

Also notice there are no data races because `array` and `size` are accessed only while holding the `this` lock. Not holding the lock in the constructor is fine because a new object is not yet reachable from any other thread. The new object that the constructor produces (i.e., the new instance of `Stack`) will have to be assigned to some thread-shared location before a second thread could use it, and such an assignment will not happen until after the constructor finishes executing.[1]

Now suppose we want to implement a new operation for our stack called `peek` that returns the newest not-yet-popped element in the stack without popping it. (This operation is also sometimes called `top`.) A correct implementation would be:

```
synchronized E peek() {
    if(index==0)
        throw new StackEmptyException();
    return array[index-1];
}
```

[1]This would not necessarily be the case if the constructor did something like `someObject.f = this;`, but doing such things in constructors is usually a bad idea.

If we omit the `synchronized` keyword, then this operation would cause data races with simultaneous `push` or `pop` operations.

Consider instead this alternate also-correct implementation:

```
synchronized E peek() {
  E ans = pop();
  push(ans);
  return ans;
}
```

This version is perhaps worse style, but it is certainly correct. It also has the advantage that this approach could be taken by a helper method outside the class where the first approach could not since `array` and `index` are private fields:

```
class C {
  static <E> E myPeekHelper(Stack<E> s) {
     synchronized (s) {
       E ans = s.pop();
       s.push(ans);
       return ans;
     }
  }
}
```

Notice this version could not be written if stacks used some private inaccessible lock. Also notice that it relies on reentrant locks. However, we will consider instead this WRONG version where the helper method omits its own synchronized statement:

```
class C {
  static <E> E myPeekHelperWrong(Stack<E> s) {
     E ans = s.pop();
```

```
        s.push(ans);
        return ans;
      }
    }
```

Notice this version has no data races because the `pop` and `push` calls still acquire and release the appropriate lock. Also notice that `myPeekHelperWrong` uses the stack operators exactly as it is supposed to. Nonetheless, it is incorrect because a peek operation is not supposed to modify the stack. While the *overall result* is the same stack if the code runs without interleaving from other threads, the code produces an *intermediate state* that other threads should not see or modify. This intermediate state would not be observable in a single-threaded program or in our correct version that made the entire method body a critical section.

To show why the wrong version can lead to incorrect behavior, we can demonstrate interleavings with other operations such that the stack does the wrong thing. Writing out such interleavings is good practice for reasoning about concurrent code and helps determine what needs to be in a critical section. As it turns out in our small example, `myPeekHelperWrong` causes race conditions with all other stack operations, including itself.

`myPeekHelperWrong` and `isEmpty`:

If a stack is not empty, then `isEmpty` should return `false`. Suppose two threads share a stack `stk` that has one element in it. If one thread calls `isEmpty` and the other calls `myPeekHelperWrong`, the first thread can get the wrong result:

```
Thread 1                    Thread 2 (calls myPeekHelperWrong)
--------                    ----------------------------------
                            E ans = stk.pop();
boolean b = isEmpty();
                            stk.push(ans);
                            return ans;
```

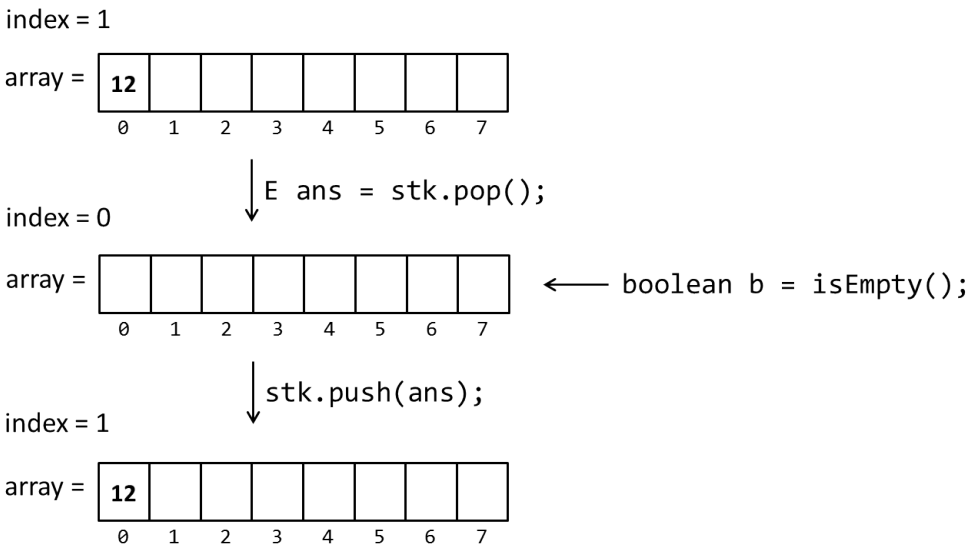

Figure (8.2) A bad interleaving for a stack with a peek operation that is incorrect in a concurrent program.

Figure 8.2 shows how this interleaving produces the wrong result when the stack has one element. Notice there is nothing wrong with what "Java is doing" here. It is the programmer who wrote `myPeekHelperWrong` such that it does not behave like a peek operation should. Yet the code that gets the wrong answer is `isEmpty`, which is another reason debugging concurrent programs is difficult.

`myPeekHelperWrong` and `push`:

Another key property of stacks is that they return elements in last-in-first-out order. But consider this interleaving, where one of Thread 1's `push` operations happens in the middle of a concurrent call to `myPeekHelperWrong`.

```
Thread 1                    Thread 2 (calls myPeekHelperWrong)
--------                    ----------------------------------
stk.push(x);

                            E ans = stk.pop();

stk.push(y);
```

```
                            stk.push(ans);
                            return ans;
E z = stk.pop();
```

This interleaving has the effect of reordering the top two elements of the stack, so `z` ends up holding the wrong value.

`myPeekHelperWrong` and `pop`:

Similar to the previous example, a `pop` operation (instead of a `push`) that views the intermediate state of `myPeekHelperWrong` can get the wrong (not last-in-first-out) answer:

```
Thread 1                    Thread 2 (calls myPeekHelperWrong)
--------                    ----------------------------------
stk.push(x);
stk.push(y);
                            E ans = stk.pop();
E z = stk.pop();
                            stk.push(ans);
                            return ans;
```

`myPeekHelperWrong` and `myPeekHelperWrong`:

Finally, two threads both calling `myPeekHelperWrong` causes race conditions. First, if `stk` initially has one element, then a bad interleaving of the two calls could raise a `StackEmptyException`, which should happen only when a stack has zero elements:

```
Thread 1                    Thread 2 (calls myPeekHelperWrong)
--------                    ----------------------------------
E ans = stk.pop();
                            E ans = stk.pop(); // exception!
stk.push(ans);
return ans;
```

Second, even if the stack has more than 1 element, we can extend the interleaving above to swap the order of the top two elements, which will cause later `pop` operations to get the wrong answer.

```
Thread 1                    Thread 2 (calls myPeekHelperWrong)
--------                    ----------------------------------
E ans = stk.pop();
                            E ans = stk.pop();
stk.push(ans);
return ans;
                            stk.push(ans);
                            return ans;
```

In general, bad interleavings can involve any number of threads using any operations. The defense is to define large enough critical sections such that every possible thread schedule is correct. Because `push`, `pop`, and `isEmpty` are synchronized, we do not have to worry about calls to these methods being interleaved with each other. To implement `peek` correctly, the key insight is to realize that its intermediate state must not be visible and therefore we need its calls to `pop` and `push` to be part of *one* critical section instead of two separate critical sections. As Section 8.5 discusses in more detail, making your critical sections large enough — but not too large — is an essential task in concurrent programming.

8.4.2 Data Races: Wrong Even When They Look Right

This section provides more detail on the rule that data races — simultaneous read/write or write/write of the same field — must be avoided even if it "seems like" the program would be correct anyway.

For example, consider the `Stack` implementation from Section 8.4.1 and suppose we omit synchronization from the `isEmpty` method. Given everything we have discussed so far, doing so seems okay, if perhaps a bit risky. After all, `isEmpty` only reads the `index`

field and since all the other critical sections write to `index` at most once, it cannot be that `isEmpty` observes an intermediate state of another critical section: it would see `index` either before or after it was updated — and both possibilities remain when `isEmpty` is properly synchronized.

Nonetheless, omitting synchronization introduces data races with concurrent `push` and `pop` operations. As soon as a Java program has a data race, it is extremely difficult to reason about what might happen. (The author of this chapter is genuinely unsure if the stack implementation is correct with `isEmpty` unsynchronized.) In the C++11 standard, it is *impossible* to do such reasoning: any program with a data race is as wrong as a program with an array-bounds violation. So data races must be avoided. The rest of this section gives some indication as to *why* and a little more detail on *how* to avoid data races.

Inability to Reason In the Presence of Data Races Let's consider an example that is simpler than the stack implementation but is so simple that we won't motivate why anyone would write code like this:

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1; // line A
    y = 1; // line B
  }
  void g() {
    int a = y; // line C
    int b = x; // line D
    assert(b >= a);
  }
```

```
}
```

Suppose we have two threads operating on the same instance of `C`, one calling `f` and the other calling `g`. Notice `f` and `g` are not synchronized, leading to potential data races on fields `x` and `y`. Therefore, it turns out the assertion in `g` can fail. But there is no interleaving of operations that justifies the assertion failure! Here we prove that all interleavings are correct. Then Section 8.4.2 explains why the assertion can fail anyway.

- If a call to `g` begins after at least one call to `f` has ended, then `x` and `y` are both 1, so the assertion holds.
- Similarly, if a call to `g` completes before any call to `f` begins, then `x` and `y` are both 0, so the assertion holds.
- So we need only concern ourselves with a call to `g` that is interleaved with the first call to `f`. One way to proceed is to consider all possible interleavings of the lines marked A, B, C, and D and show the assertion holds under all of them. Because A must precede B and C must precede D, there are only 6 such interleavings: ABCD, ACBD, ACDB, CABD, CADB, CDAB. Manual inspection of all six possibilities completes the proof.

  A more illuminating approach is a proof by contradiction: Assume the assertion fails, meaning `!(b>=a)`. Then `a==1` and `b==0`. Since `a==1`, line B happened before line C. Since (1) A must happen before B, (2) C must happen before D, and (3) "happens before" is a transitive relation, A must happen before D. But then `b==1` and the assertion holds.

There is nothing wrong with the proof except its assumption that we can reason in terms of "all possible interleavings" or that everything happens in certain orders. We can reason this way *only* if the program has no data races.

Partial Explanation of Why Data Races Are Disallowed To understand fully why one cannot always reason in terms of interleavings requires taking courses in compilers and/or computer architecture.

Compilers typically perform *optimizations* to execute code faster without changing its meaning. If we required compilers to never change the possible interleavings, then it would be too difficult for compilers to be effective. Therefore, language definitions allow compilers to do things like execute line B above before line A. Now, in this simple example, there is no reason why the compiler *would* do this reordering. The point is that it is *allowed to*, and in more complicated examples with code containing loops and other features there are reasons to do so. Once such a reordering is allowed, the assertion is allowed to fail. Whether it will ever do so depends on exactly how a particular Java implementation works.

Similarly, in hardware, there isn't *really* one single shared memory containing a single copy of all data in the program. Instead there are various caches and buffers that let the processor access some memory faster than other memory. As a result, the hardware has to keep track of different copies of things and move things around. As it does so, memory operations might not become "visible" to other threads in the order they happened in the program. As with compilers, requiring the hardware to expose all reads and writes in the exact order they happen is considered too onerous from a performance perspective.

To be clear, compilers and hardware cannot just "do whatever they want." All this reordering is *completely hidden from you* and you never need to worry about it *if* you avoid data races. This issue is irrelevant in single-threaded programming because with one thread you can never have a data race.

The Grand Compromise One way to summarize the previous section is in terms of a "grand compromise" between programmers who want to be able to reason easily about what a program might do and compiler/hardware implementers who want to implement things efficiently and easily:

- The programmer promises not to write data races.
- The implementers promise that if the programmer does his/her job, then the implementation will be indistinguishable from one that does no reordering of memory operations. That is, it will preserve the illusion that all memory operations are interleaved in a

global order.

Why is this compromise the state of the art rather than a more extreme position of requiring the implementation to preserve the "interleaving illusion" even in the presence of data races? In short, it would make it much more difficult to develop compilers and processors. And for what? Just to support programs with data races, which are almost always bad style and difficult to reason about anyway! Why go to so much trouble just for ill-advised programming style? On the other hand, if your program accidentally has data races, it might be more difficult to debug.

Avoiding Data Races To avoid data races, we need synchronization. Given Java's locks, which we already know about, we could rewrite our example from Section 8.4.2 as follows:

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    synchronized(this) { x = 1; } // line A
    synchronized(this) { y = 1; } // line B
  }
  void g() {
    int a, b;
    synchronized(this) { a = y; } // line C
    synchronized(this) { b = x; } // line D
    assert(b >= a);
  }
}
```

In practice you might choose to implement each method with one critical section instead of two, but using two is still sufficient to eliminate data races, which allows us to reason in terms of the six possible interleavings, which ensures the assertion cannot fail. It would even be correct to use two different locks, one to protect `x` (lines A and D) and another to protect `y` (lines B and C). The bottom line is that by avoiding data races we can reason in terms of interleavings (ignoring reorderings) because we have fulfilled our obligations under The Grand Compromise.

There is a second way to avoid data races when writing tricky code that depends on the exact ordering of reads and writes to fields. Instead of using locks, we can declare fields to be *volatile*. By *definition*, accesses to volatile fields *do not count as data races*, so programmers using volatiles are still upholding their end of the grand compromise. In fact, this definition is the reason that `volatile` is a keyword in Java — and the reason that until you study concurrent programming you will probably not encounter it. Reading/writing a volatile field is less efficient than reading/writing a regular field, but more efficient than acquiring and releasing a lock. For our example, the code looks like this:

```
class C {
  private volatile int x = 0;
  private volatile int y = 0;

  void f() {
    x = 1; // line A
    y = 1; // line B
  }
  void g() {
    int a = y; // line C
    int b = x; // line D
    assert(b >= a);
  }
```

```
}
```

Volatile fields are typically used only by concurrency experts. They can be convenient when concurrent code is sharing only a single field. Usually with more than one field you need critical sections longer than a single memory access, in which case the right tool is a lock, not a volatile field.

A More Likely Example Since the previous discussion focused on a strange and unmotivated example (who would write code like `f` and `g`?), this section concludes with a data-race example that would arise naturally.

Suppose we want to perform some expensive iterative computation, for example, rendering a really ugly monster for a video game, until we have achieved a perfect solution or another thread informs us that the result is "good enough" (it's time to draw the monster already) or no longer needed (the video-game player has left the room with the monster). We could have a boolean `stop` field that the expensive computation checks periodically:

```
class MonsterDraw {
  boolean stop = false;
  void draw(...) {
    while(!stop) {
      ... keep making uglier ...
    }
    ...
  }
}
```

Then one thread would execute `draw` and other threads could set `stop` to true as necessary. Notice that doing so introduces data races on the `stop` field. To be honest, even though this programming approach is wrong, with most Java implementations it will probably work. But if it does not, it is the programmer's fault. Worse, the code might work for a long time,

but when the Java implementation changes or the compiler is run with different pieces of code, the idiom might stop working. Most likely what would happen is the `draw` method would never "see" that `stop` was changed and it would not terminate.

The simplest "fix" to the code above is to declare `stop` volatile. However, many concurrency experts would consider this idiom poor style and would recommend learning more about Java's thread library, which has built-in support for having one thread "interrupt" another one. Java's concurrency libraries are large with built-in support for many coding patterns. This is just one example of a pattern where using the existing libraries is "the right thing to do," but we do not go into the details of the libraries in this chapter.

8.5 Concurrency Programming Guidelines

This section does not introduce any new primitives for concurrent programming. Instead, we acknowledge that writing correct concurrent programs is difficult — just saying, "here is how locks work, try to avoid race conditions but still write efficient programs" would leave you ill-equipped to write non-trivial multithreaded programs. Over the decades that programmers have struggled with shared memory and locks, certain guidelines and methodologies have proven useful. Here we sketch some of the most widely agreed upon approaches to getting your synchronization correct. Following them does not make concurrency easy, but it does make it much easier. We focus on how to design/organize your code, but related topics like how to use tools to test and debug concurrent programs are also important.

8.5.1 Conceptually Splitting Memory in Three Parts

Every memory location in your program (for example an object field), should meet *at least one* of the following three criteria:

1. It is *thread-local*: Only one thread ever accesses it.
2. It is *immutable*: (After being initialized,) it is only read, never written.
3. It is *synchronized*: Locks are used to ensure there are no race conditions.

Put another way, memory that is thread-local or immutable is irrelevant to synchronization. You can ignore it when considering how to use locks and where to put critical sections. So the more memory you have that is thread-local or immutable, the easier concurrent programming will be. It is often worth changing the data structures and algorithms you are using to choose ones that have less sharing of mutable memory, leaving synchronization only for the unavoidable communication among threads. We cannot emphasize enough that each memory location needs to meet at least one of the three conditions above to avoid race conditions, and if it meets either of the first two (or both), then you do not need locks.

So a powerful guideline is to make as many objects as you can immutable or thread-local or both. Only when doing so is ineffective for your task do you need to figure out how to use synchronization correctly. Therefore, we next consider ways to make more memory thread-local and then consider ways to make more memory immutable.

More Thread-Local Memory

Whenever possible, do not share resources. Instead of having one resource, have a separate (copy of the) resource for each thread. For example, there is no need to have one shared `Random` object for generating pseudorandom numbers, since it will be just as random for each thread to use its own object. Dictionaries storing already-computed results can also be thread-local: This has the disadvantage that one thread will not insert a value that another thread can then use, but the decrease in synchronization is often worth this trade-off.

Note that because each thread's call-stack is thread-local, there is never a need to synchronize on local variables. So using local variables instead of object fields can help where possible, but this is good style anyway even in single-threaded programming.

Overall, the vast majority of objects in a typical concurrent program should and will be thread-local. It may not seem that way in this chapter, but only because we are focusing on the shared objects, which are the difficult ones. In conclusion, do not share objects unless those objects really have the *explicit purpose* of enabling shared-memory communication among threads.

More Immutable Memory

Whenever possible, do not update fields of objects: Make new objects instead. This guideline is a key tenet of *functional programming*. Even in single-threaded programs, it is often helpful to avoid mutating fields of objects because other too-often-forgotten-about *aliases* to that object will "see" any mutations performed.

Let's consider a small example that demonstrates some of the issues. Suppose we have a dictionary such as a hashtable that maps student ids to names where the name class is represented like this:

```
class Name {
  public String first;
  public String middle;
  public String last;
  public Name(String f, String m, String l) {
    first  = f;
    middle = m;
    last   = l;
  }
  public String toString() {
    return first + " " + middle + " " + last;
  }
}
```

Suppose we want to write a method that looks up a student by id in some table and returns the name, but using a middle initial instead of the full middle name. The following direct solution is probably the BEST STYLE:

```
Name n = table.lookup(id);
return n.first + " " + n.middle.charAt(0) + " " + n.last;
```

Notice that the computation is almost the same as that already done by `toString`. So if the computation were more complicated than just a few string concatenations, it would be

tempting and arguably reasonable to try to reuse the `toString` method. This approach is a BAD IDEA:

```
Name n = table.lookup(id);
n.middle = n.middle.substring(0,1);
return n.toString();
```

While this code produces the right answer, it has the *side-effect* of changing the name's `middle` field. Worse, the Name object is *still in the table*. Actually, this depends on how the `lookup` method is implemented. We assume here it returns a reference to the object in the table, rather than creating a new Name object. If Name objects are not mutated, then returning an alias is simpler and more efficient. Under this assumption, the full middle name for some student has been *replaced* in the table by the middle initial, which is presumably a bug.

In a SINGLE-THREADED PROGRAM, the following workaround is correct but POOR STYLE:

```
Name n = table.lookup(id);
String m = n.middle;
n.middle = m.substring(0,1);
String ans = n.toString();
n.middle = m;
return ans;
```

Again, this is total overkill for our example, but the idea makes sense: undo the side effect before returning.

But if the table is shared among threads, the APPROACH ABOVE IS WRONG. Simultaneous calls using the same ID would have data races. Even if there were no data races, there would be interleavings where other operations might see the intermediate state where the `middle` field held only the initial. The point is that for shared memory, you cannot perform side effects and undo them later: later is too late in the presence of concurrent

operations. This is a major reason that the functional-programming paradigm is even more useful with concurrency.

The solution is to make a copy of the object and, if necessary, mutate the (thread-local) copy. So this WORKS:

```
Name n1 = table.lookup(id);
Name n2 = new Name(n1.first, n1.middle.substring(0,1), n1.last);
return n2.toString();
```

Note that it is no problem for multiple threads to look up the same Name object at the same time. They will all *read* the same fields of the same object (they all have aliases to it), but simultaneous reads never lead to race conditions. It is like multiple pedestrians reading the same street sign at the same time: They do not bother each other.

8.5.2 Approaches to Synchronization

Even after maximizing the thread-local and immutable objects, there will remain some shared resources among the threads (else threads have no way to communicate). The remaining guidelines suggest ways to think about correctly synchronizing access to these resources.

Guideline #0: Avoid data races

Use locks to ensure that two threads never simultaneously read/write or write/write the same field. While guideline #0 is *necessary* (see Section 8.4.2), it is not *sufficient* (see Section 8.4.1 and other examples of wrong programs that have no data races).

Guideline #1: Use consistent locking

For each location that needs synchronization, identify a lock that is always held when accessing that location. If some lock l is always held when memory location m is read or written, then we say that l *guards* the location m. It is extremely good practice to document (e.g., via comments) for each field needing synchronization what lock guards that location. For example, the most common situation in Java would be, "`this` guards all fields of instances of the class."

Note that each location would have exactly one guard, but one lock can guard multiple locations. A simple example would be multiple fields of the same object, but any mapping from locks to locations is okay. For example, perhaps all nodes of a linked list are protected by one lock. Another way to think about consistent locking is to imagine that all the objects needing synchronization are *partitioned*. Each piece of the partition has a lock that guards all the locations in the piece.

Consistent locking is *sufficient* to prevent data races, but is *not sufficient* to prevent bad interleavings. For example, the extended example in Section 8.4.1 used consistent locking.

Consistent locking is *not necessary* to prevent data races nor bad interleavings. It is a guideline that is typically the default assumption for concurrent code, violated only when carefully documented and for good reason. One good reason is when the program has multiple conceptual "phases" and all threads globally coordinate when the program moves from one phase to another. In this case, different phases can use different synchronization strategies.

For example, suppose in an early phase multiple threads are inserting different key-value pairs into a dictionary. Perhaps there is one lock for the dictionary and all threads acquire this lock before performing any dictionary operations. Suppose that at some point in the program, the dictionary becomes fixed, meaning no more insertions or deletions will be performed on it. Once all threads *know* this point has been reached (probably by reading some other synchronized shared-memory object to make sure that all threads are done adding to the dictionary), it would be correct to perform subsequent lookup operations *without synchronization* since the dictionary has *become immutable*.

Guideline #2: Start with coarse-grained locking and move to finer-grained locking only if contention is hurting performance

This guideline introduces some new terms. *Coarse-grained locking* means using fewer locks to guard more objects. For example, one lock for an entire dictionary or for an entire array of bank accounts would be coarse-grained. Conversely, *fine-grained locking* means using more locks, each of which guards fewer memory locations. For example, using a separate

lock for each bucket in a chaining hashtable or a separate lock for each bank account would be fine-grained. Honestly, the terms "coarse-grained" and "fine-grained" do not have a strict dividing line: we can really only say that one locking strategy is "more coarse-grained" or "more fine-grained" than another one. That is, *locking granularity* is really a continuum, where one direction is coarser and the other direction is finer.

Coarse-grained locking is typically easier. After acquiring just one lock, we can access many different locations in a critical section. With fine-grained locking, operations may end up needing to acquire multiple locks (using nested synchronized statements). It is easy to forget to acquire a lock or to acquire the wrong lock. It also introduces the possibility of deadlock (Section 8.6).

But coarse-grained locking leads to threads waiting for other threads to release locks unnecessarily, i.e., in situations when no errors would result if the threads proceeded concurrently. In the extreme, the coarsest strategy would have just one lock for the entire program! Under this approach, it is obvious what lock to acquire, but no two operations on shared memory can proceed in parallel. Having one lock for many, many bank accounts is probably a bad idea because with enough threads, there will be multiple threads attempting to access bank accounts at the same time and they will be unable to. *Contention* describes the situation where threads are blocked waiting for each other: They are *contending* (in the sense of competing) for the same resource, and this is hurting performance. If there is little contention (it is rare that a thread is blocked), then coarse-grained locking is sufficient. When contention becomes problematic, it is worth considering finer-grained locking, realizing that changing the code without introducing race conditions is difficult. A good execution profiler[2] should provide information on where threads are most often blocked waiting for a lock to be released.

Guideline #3: Make critical sections large enough for correctness but no larger. Do not perform I/O or expensive computations within critical sections.

[2]A profiler is a tool that reports where a run of a program is spending time or other resources.

The granularity of critical sections is completely orthogonal to the granularity of locks (guideline #2). Coarse-grained critical sections are "large" pieces of code whereas fine-grained critical sections are "small" pieces of code. However, what matters is *not* how many lines of code a critical section occupies. What matters is how long a critical section takes to execute (longer means larger) and how many shared resources such as memory locations it accesses (more means larger).

If you make your critical sections too short, you are introducing more possible interleavings and potentially incorrect race conditions. For example, compare:

```
synchronized(lk) { /* do first thing */ }
/* do second thing */
synchronized(lk) { /* do third thing */ }
```

with:

```
synchronized(lk) {
  /* do first thing */
  /* do second thing */
  /* do third thing */
}
```

The first version has smaller critical sections. Smaller critical sections lead to less contention. For example, other threads can use data guarded by `lk` while the code above is doing the "second thing." But smaller critical sections expose more states to other threads. Perhaps only the second version with one larger critical section is correct. This guideline is therefore a tough one to follow because it requires moderation: make critical sections no longer or shorter than necessary.

Basic operations like assignment statements and method calls are so fast that it is almost never worth the trouble of splitting them into multiple critical sections. However, expensive computations inside critical sections should be avoided wherever possible for the obvious reason that other threads may end up blocked waiting for the lock. Remember in

particular that reading data from disk or the network is typically orders of magnitude slower than reading memory. Long running loops and other computations should also be avoided.

Sometimes it is possible to reorganize code to avoid long critical sections by repeating an operation in the (hopefully) rare case that it is necessary. For example, consider this large critical section that replaces a table entry by performing some computation on its old value:

```
synchronized(lk) {
  v1 = table.lookup(k);
  v2 = expensive(v1);
  table.remove(k);
  table.insert(k,v2);
}
```

We assume the program is incorrect if this critical section is any smaller: We need other threads to see either the old value in the table or the new value. And we need to compute `v2` using the current `v1`. So this variation would be WRONG:

```
synchronized(lk) {
  v1 = table.lookup(k);
}
v2 = expensive(v1);
synchronized(lk) {
  table.remove(k);
  table.insert(k,v2);
}
```

If another thread updated the table to map `k` to some `v3` while this thread was executing `expensive(v1)`, then we would not be performing the correct computation. In this case, either the final value in the table should be `v3` (if that thread ran second) or the result of `expensive(v3)` (if that thread ran first), but never `v2`.

However, this more complicated version would WORK in this situation, under the CAVEAT described below:

```
boolean loop_done = false;
while(!loop_done) {
  synchronized(lk) {
    v1 = table.lookup(k);
  }
  v2 = expensive(v1);
  synchronized(lk) {
    if(table.lookup(k)==v1) {
      loop_done = true;
      table.remove(k);
      table.insert(k,v2);
    }
  }
}
```

The essence of the trick is the second critical section. If `table.lookup(k)==v1`, then we know what `expensive(v1)` would compute, having already computed it! So the critical section is small, but does the same thing as our original large critical section. But if `table.lookup(k)!=v1`, then our precomputation did useless work and the while-loop starts over. If we expect concurrent updates to be rare, then this approach is a "good performance bet" while still correct in all cases.

As promised, there is an important caveat to mention. This approach does *not* ensure the table entry for key `k` was unmodified during the call to `expensive`. There could have been any number of concurrent updates, removals, insertions, etc. with this key. All we know is that when the second critical section executes, the key `k` has value `v1`. Maybe it was unchanged or maybe it was changed and then changed back.

In our example, we assumed it did not matter. There are many situations where it *does*

matter and in such cases, checking the value is wrong because it does not ensure the entry has not been modified. This is known as an "A-B-A" problem, meaning the value started as A, changed to B, and changed back to A, causing some incorrect code to conclude, wrongly, that it was never changed.

Guideline #4: Think in terms of what operations need to be *atomic*. Determine a locking strategy after you know what the critical sections are.

An operation is *atomic*, as in indivisible, if it executes either entirely or not at all, with no other thread able to observe that it has partly executed.[3] Ensuring necessary operations appear to be atomic is exactly why we have critical sections. Since this is the essential point, this guideline recommends thinking first in terms of what the critical sections are *without thinking about locking granularity*. Then, once the critical sections are clear, how to use locks to implement the critical sections can be the next step.

In this way, we ensure we actually develop the software we want — the correct critical sections — without being distracted prematurely by the implementation details of how to do the locking. In other words, think about atomicity (using guideline #3) first and the locking protocol (using guidelines #0, #1, and #2) second.

Unfortunately, one of the most difficult aspects of lock-based programming is when software needs change. When new critical sections are needed in "version 2.0" of a program, it may require changing the locking protocol, which in turn will require carefully modifying the code for many other already-working critical sections. Section 8.6 even includes an example where there is really no good solution. Nonetheless, following guideline #4 makes it easier to think about why and how a locking protocol may need changing.

Guideline #5: Do not implement your own concurrent data structures. Use carefully tuned ones written by experts and provided in standard libraries.

Widely used libraries for popular programming languages already contain many reusable data structures that you should use whenever they meet your needs. For example, there is

[3]In the databases literature, this would be called atomic and *isolated*, but it is common in concurrent programming to conflate these two ideas under the term *atomic*.

little reason in Java to implement your own hashtable since the `Hashtable` class in the standard library has been carefully tuned and tested. For concurrent programming, the advice to "reuse existing libraries" is even more important, precisely because writing, debugging, and testing concurrent code is so difficult. Experts who devote their lives to concurrent programming can write tricky fine-grained locking code once and the rest of us can benefit. For example, the `ConcurrentHashMap` class implements a hashtable that can be safely used by multiple threads with very little contention. It uses some techniques more advanced than discussed in this chapter, but clients to the library have such trickiness hidden from them. For basic things like queues and dictionaries, do not implement your own.

If standard libraries are so good, then why learn about concurrent programming at all? For the same reason that educated computer scientists need a basic understanding of sequential data structures in order to pick the right ones and use them correctly (e.g., designing a good hash function), concurrent programmers need to understand how to debug and performance tune their use of libraries (e.g., to avoid contention).

Moreover, standard libraries do not typically handle all the necessary synchronization for an application. For example, `ConcurrentHashMap` does not support atomically removing one element and inserting two others. If your application needs to do that, then you will need to implement your own locking protocol to synchronize access to a shared data structure. Understanding race conditions is crucial. While the examples in this chapter consider race conditions on simple data structures like stacks, larger applications using standard libraries for concurrent data structures will still often have bad interleavings at higher levels of abstraction.

8.6 Deadlock

There is another common concurrency bug that you must avoid when writing concurrent programs: If a collection of threads are blocked forever, all waiting for another thread in the collection to do something (which it won't because it's blocked), we say the threads are *deadlocked*. Deadlock is a different kind of bug from a race condition, but unfortunately pre-

venting a potential race condition can cause a potential deadlock and vice versa. Deadlocks typically result only with some, possibly-rare, thread schedules (i.e., like race conditions they are nondeterministic). One advantage compared to race conditions is that it is easier to tell a deadlock occurred: Some threads never terminate because they are blocked.

As a canonical example of code that could cause a deadlock, consider a bank-account class that includes a method to transfer money to another bank-account. With what we have learned so far, the following WRONG code skeleton looks reasonable:

```
class BankAccount {
   ...
   synchronized void withdraw(int amt) { ... }
   synchronized void deposit(int amt) { ... }
   synchronized void transferTo(int amt, BankAccount a) {
     this.withdraw(amt);
     a.deposit(amt);
   }
}
```

This class uses a fine-grained locking strategy where each account has its own lock. There are no data races because the only methods that directly access any fields are (we assume) synchronized like `withdraw` and `deposit`, which acquire the correct lock. More subtly, the `transferTo` method appears atomic. For another thread to see the intermediate state where the withdraw has occurred but not the deposit would require operating on the withdrawn-from account. Since `transferTo` is synchronized, this cannot occur.

Nonetheless, there are interleavings of two `transferTo` operations where neither one ever finishes, which is clearly not desired. For simplicity, suppose `x` and `y` are static fields each holding a `BankAccount`. Consider this interleaving in pseudocode:

```
Thread 1: x.transferTo(1,y)      Thread 2: y.transferTo(1,x)
--------------------------       --------------------------
```

```
acquire lock for x
withdraw 1 from x
                                    acquire lock for y
                                    withdraw 1 from y
                                    block on lock for x
block on lock for y
```

In this example, one thread transfers from one account (call it "A") to another (call it "B") while the other thread transfers from B to A. With just the wrong interleaving, the first thread holds the lock for A and is blocked on the lock for B while the second thread holds the lock for B and is blocked on the lock for A. So both are waiting for a lock to be released by a thread that is blocked. They will wait forever.

More generally, a deadlock occurs when there are threads $T_1, T_2, ..., T_n$ such that:

- For $1 \leq i \leq (n-1)$, T_i is waiting for a resource held by T_{i+1}
- T_n is waiting for a resource held by T_1

In other words, there is a cycle of waiting.

Returning to our example, there is, unfortunately, no obvious way to write an atomic `transferTo` method that will not deadlock. Depending on our needs we could write:

```
void transferTo(int amt, BankAccount a) {
  this.withdraw(amt);
  a.deposit(amt);
}
```

All we have done is make `transferTo` not be synchronized. Because `withdraw` and `deposit` are still synchronized methods, the only negative effect is that an intermediate state is potentially exposed: Another thread that retrieved the balances of the two accounts in the transfer could now "see" the state where the withdraw had occurred but not the deposit. If

that is okay in our application, then this is a sufficient solution. If not, another approach would be to resort to coarse-grained locking where all accounts use the same lock.

Either of these approaches avoids deadlock because they ensure that no thread ever holds more than one lock at a time. This is a sufficient but not necessary condition for avoiding deadlock because it cannot lead to a cycle of waiting. A much more flexible sufficient-but-not-necessary strategy that subsumes the often-unworkable "only one lock at a time strategy" works as follows: For all pairs of locks x and y, either do not have a thread ever (try to) hold both x and y simultaneously *or* have a globally agreed upon order, meaning either x is always acquired before y or y is always acquired before x.

This strategy effectively defines a conceptual partial order on locks and requires that a thread tries to acquire a lock only if all locks currently held by the thread (if any) come earlier in the partial order. It is merely a programming convention that the entire program must get right.

To understand how this ordering idea can be useful, we return to our `transferTo` example. Suppose we wanted `transferTo` to be atomic. A simpler way to write the code that still has the deadlock problem we started with is:

```
void transferTo(int amt, BankAccount a) {
  synchronized(this) {
    synchronized(a) {
      this.withdraw(amt);
      a.deposit(amt);
    }
  }
}
```

Here we make explicit that we need the locks guarding both accounts and we hold those locks throughout the critical section. This version is easier to understand and the potential deadlock — still caused by another thread performing a transfer to the accounts in reverse order — is also more apparent. (Note in passing that a deadlock involving more threads is

also possible. For example, thread 1 could transfer from A to B, thread 2 from B to C, and thread 3 from C to A.)

The critical section has two locks to acquire. If all threads acquired these locks in the same order, then no deadlock could occur. In our deadlock example where one thread calls `x.transferTo(1,y)` and the other calls `y.transferTo(1,x)`, this is exactly what goes wrong: one acquires `x` before `y` and the other `y` before `x`. We want one of the threads to acquire the locks in the other order.

In general, there is no clear way to do this, but sometimes our data structures have some unique identifier that lets us pick an arbitrary order to prevent deadlock. Suppose, as is the case in actual banks, that every `BankAccount` already has an `acctNumber` field of type `int` that is distinct from every other `acctNumber`. Then we can use these numbers to require that threads always acquire locks for bank accounts in increasing order. (We could also require decreasing order, the key is that all code agrees on one way or the other.) We can then write `transferTo` like this:

```
class BankAccount {
  ...
  private int acctNumber; // must be unique
  void transferTo(int amt, BankAccount a) {
    if(this.acctNumber < a.acctNumber)
       synchronized(this) {
         synchronized(a) {
            this.withdraw(amt);
            a.deposit(amt);
         }
       }
    else
       synchronized(a) {
         synchronized(this) {
```

```
            this.withdraw(amt);
            a.deposit(amt);
          }
        }
  }
}
```

While it may not be obvious that this code prevents deadlock, it should be obvious that it follows the strategy described above. And *any* code following this strategy will not deadlock because we cannot create a cycle if we acquire locks according to the agreed-upon partial order.

In practice, methods like `transferTo` where we have two instances of the same class are a difficult case to handle. Even in the Java standard library it does not always work well: The `StringBuffer` class has an `append` method that is subject to race conditions (bad interleavings) because the only way to avoid it would be to create a possible deadlock. With no clear, efficient solution (buffers do not have unique identifiers like bank accounts), the documentation for `StringBuffer` indicates that clients of the library must use their own synchronization to prevent calls to `append` from interleaving with other methods that change the contents or size of a `StringBuffer`.

Fortunately, there are other situations where preventing deadlock amounts to fairly simple rules about the order that locks are acquired. Here are two examples:

- If you have two different types of objects, you can order the locks by the types of data they protect. For example, the documentation could state, "When moving an item from the hashtable to the work queue, never try to acquire the queue's lock while holding the hashtable's lock (the other order is acceptable)."

- If you have an acyclic data structure like a tree, you can use the references in the data structure to define the locking order. For example, the documentation could state, "If holding a lock for a node in the tree, do not acquire other nodes' locks unless they are

descendants in the tree."

8.7 Additional Synchronization Primitives

So far, the synchronization primitives we have seen are (reentrant) locks, `join`, and volatile fields. This section describes two more useful primitives — reader/writer locks and condition variables — in detail, emphasizing how they provide facilities that basic locks do not. Then Section 8.7.3 briefly mentions some other primitives.

8.7.1 Reader/Writer Locks

This chapter has emphasized multiple times that it is *not* an error for multiple threads to read the same information at the same time. Simultaneous reads of the same field by any number of threads is not a data race. Immutable data does not need to be accessed from within critical sections. But as soon as there might be concurrent writes, we have used locks that allow only one thread at a time. *This is unnecessarily conservative.* Considering all the data guarded by a lock, it would be fine to allow multiple simultaneous *readers* of the data provided that any *writer* of the data does so exclusively, i.e., while there are neither concurrent readers nor writers. In this way, we still prevent any data race or bad interleaving resulting from read/write or write/write errors. A real-world (or at least on-line) analogy could be a web-page: It is fine for many people to read the same page at the same time, but there should be at most one person editing the page, during which nobody else should be reading or writing the page.

As an example where this would be useful, consider a dictionary such as a hashtable protected by a single lock (i.e., simple, coarse-grained locking). Suppose that lookup operations are common but insert or delete operations are very rare. As long as lookup operations do not actually mutate any elements of the dictionary, it is fine to allow them to proceed in parallel and doing so avoids unnecessary contention.[4] When an insert or delete operation

[4]A good question is why lookups *would* mutate anything. To *clients* of the data structure, no mutation should be apparent, but there are data-structure techniques that *internally* modify memory locations to im-

does arrive, it would need to block until there were no other operations active (lookups or other operations that mutate the dictionary), but (a) that is rare and (b) that is what we expect from coarse-grained locking. In short, we would like to support simultaneous concurrent reads. Fine-grained locking might help with this, but only indirectly and incompletely (it still would not help with reads of the same dictionary key), and it is much more difficult to implement.

Instead a *reader/writer lock* provides exactly what we are looking for as a primitive. It has a different interface than a regular lock: A reader/writer lock is acquired either "to be a reader" or "to be a writer." The lock allows multiple threads to hold the lock "as a reader" at the same time, but only one to hold it as a writer and only if there are no holders as readers. In particular, here are the operations:

- `new` creates a new lock that initially has "0 readers and 0 writers"
- `acquire_write` blocks if there are currently any readers or writers, else it makes the lock "held for writing"
- `release_write` returns the lock to its initial state
- `acquire_read` blocks if the lock is currently "held for writing" else it increments the number of readers
- `release_read` decrements the number of readers, which may or may not return the lock to its initial state

A reader/writer lock will block an `acquire_write` or `acquire_read` operation as necessary to maintain the following invariants, where we write $|r|$ and $|w|$ for the number of threads holding the lock for reading and writing, respectively.

- $0 \leq |w| \leq 1$, i.e., $|w|$ is 0 or 1

prove asymptotic guarantees. Two examples are move-to-front lists and splay trees. These techniques conflict with using reader/writer locks as described in this section, which is another example of the disadvantages of mutating shared memory unnecessarily.

- $|w| * |r| = 0$, i.e., $|w|$ is 0 or $|r|$ is 0

Note that the name *reader/writer lock* is really a misnomer. Nothing about the definition of the lock's operations has anything to do with reading or writing. A better name might be a *shared/exclusive lock*: Multiple threads can hold the lock in "shared mode" but only one can hold it in "exclusive mode" and only when no threads hold it in "shared mode." It is up to the programmer to use the lock correctly, i.e., to ensure that it is okay for multiple threads to proceed simultaneously when they hold the lock for reading. And the most sensible way to ensure this is to read memory but not write it.

Returning to our dictionary example, using a reader/writer lock is entirely straightforward: The lookup operation would acquire the coarse-grained lock for reading, and other operations (insert, delete, resize, etc.) would acquire it for writing. If these other operations are rare, there will be little contention.

There are a few details related to the semantics of reader/writer locks worth understanding. To learn how a particular library resolves these questions, you need to read the documentation. First, some reader/writer locks give *priority* to writers. This means that an "acquire for writing" operation will succeed before any "acquire for reading" operations that *arrive later*. The "acquire for writing" still has to wait for any threads that *already* hold the lock for reading to release the lock. This priority can prevent writers from *starving* (blocking forever) if readers are so common or have long enough critical sections that the number of threads wishing to hold the lock for reading is never 0.

Second, some libraries let a thread *upgrade* from being a reader to being a writer without releasing the lock. Upgrading might lead to deadlock though if multiple threads try to upgrade.

Third, just like regular locks, a reader/writer lock may or may not be reentrant.[5] Aside from the case of upgrading, this is an orthogonal issue.

Java's synchronized statement supports only regular locks, not reader/writer locks. But

[5]Recall reentrant locks allow the same thread to "acquire" the lock multiple times and hold the lock until the equal number of release operations.

Java's standard library has reader/writer locks, such as the class `ReentrantReadWriteLock` in the `java.util.concurrent.locks` package. The interface is slightly different than with synchronized statements (you have to be careful to release the lock even in the case of exceptions). It is also different than the operations we described above. Methods `readLock` and `writeLock` return objects that then have `lock` and `unlock` methods. So "acquire for writing" could look like `lk.writeLock().lock()`. One advantage of this more complicated interface is that some library could provide some clients only one of the "read lock" or the "write lock," thereby restricting which operations the clients can perform.

8.7.2 Condition Variables

Condition variables are a mechanism that lets threads *wait* (block) until another thread *notifies* them that it might be useful to "wake up" and try again to do whatever could not be done before waiting. (We assume the thread that waited chose to do so because some *condition* prevented it from continuing in a useful manner.) It can be awkward to use condition variables correctly, but most uses of them are within standard libraries, so arguably the most useful thing is to understand when and why this wait/notification approach is desirable.

To understand condition variables, consider the canonical example of a *bounded buffer*. A bounded buffer is just a queue with a maximum size. Bounded buffers that are shared among threads are useful in many concurrent programs. For example, if a program is conceptually a pipeline (an assembly line), we could assign some number of threads to each stage and have a bounded buffer between each pair of adjacent stages. When a thread in one stage produces an object for the next stage to process, it can enqueue the object in the appropriate buffer. When a thread in the next stage is ready to process more data, it can dequeue from the same buffer. From the perspective of this buffer, we call the enqueuers *producer threads* (or just *producers*) and the dequeuers *consumer threads* (or just *consumers*).

Naturally, we need to synchronize access to the queue to make sure each object is enqueued/dequeued exactly once, there are no data races, etc. One lock per bounded buffer

will work fine for this purpose. More interesting is what to do if a producer encounters a full buffer or a consumer encounters an empty buffer. In single-threaded programming, it makes sense to throw an exception when encountering a full or empty buffer since this is an unexpected condition that cannot change. But here:

- If the thread simply waits and tries again later, a producer may find the queue no longer full (if a consumer has dequeued) and a consumer may find the queue no longer empty (if a producer has enqueued).
- It is not unexpected to find the queue in a "temporarily bad" condition for continuing. The buffer is for managing the handoff of data and it is entirely natural that at some point the producers might get slightly ahead (filling the buffer and wanting to enqueue more) or the consumers might get slightly ahead (emptying the buffer and wanting to dequeue more).

The fact that the buffer is bounded is useful. Not only does it save space, but it ensures the producers will not get too far ahead of the consumers. Once the buffer fills, we want to stop running the producer threads since we would rather use our processors to run the consumer threads, which are clearly not running enough or are taking longer. So we really do want waiting threads to stop using computational resources so the threads that are not stuck can get useful work done.

With just locks, we can write a version of a bounded buffer that never results in a wrong answer, but it will be INEXCUSABLY INEFFICIENT because threads will not stop using resources when they cannot proceed. Instead they keep checking to see if they can proceed:

```
class Buffer<E> {
   // not shown: an array of fixed size for the queue with two indices
   // for the front and back, along with methods isEmpty() and isFull()
   void enqueue(E elt) {
      while(true) {
        synchronized(this) {
```

```
            if(!isFull()) {
               ... do enqueue as normal ...
               return;
            }
         }
      }
   }
   E dequeue() {
      while(true) {
        synchronized(this) {
           if(!isEmpty()) {
              E ans = ... do dequeue as normal ...
              return ans;
           }
        }
      }
   }
}
```

The purpose of condition variables is to avoid this *spinning*, meaning threads checking the same thing over and over again until they can proceed. This spinning is particularly bad because it causes contention for exactly the lock that other threads need to acquire in order to do the work before the spinners can do something useful. Instead of constantly checking, the threads could "sleep for a while" before trying again. But how long should they wait? Too little waiting slows the program down due to wasted spinning. Too much waiting is slowing the program down because threads that could proceed are still sleeping.

It would be best to have threads "wake up" (unblock) exactly when they might usefully continue. In other words, if producers are blocked on a full queue, wake them up when a dequeue occurs and if consumers are blocked on an empty queue, wake them up when

an enqueue occurs. This cannot be done correctly with just locks, which is exactly why condition variables exist.

In Java, just like every object is a lock, every object is *also* a condition variable. So we will first explain condition variables as extra methods that every object has. (Quite literally, the methods are defined in the `Object` class.) We will see later that while this set-up is often convenient, associating (only) one condition variable with each lock (the same object) is not always what you want. There are three methods:

- `wait` should be called only while holding the lock for the object. (Typically, the call is to `this.wait` so the lock `this` should be held.) The call will atomically (a) block the thread, (b) release the lock, and (c) register the thread as "waiting" to be notified. (This is the step you cannot reasonably implement on your own; there has to be no "gap" between the lock-release and registering for notification else the thread might "miss" being notified and never wake up.) When the thread is later woken up it will re-acquire the same lock before `wait` returns. (If other threads also seek the lock, then this may lead to more blocking. There is no guarantee that a notified thread gets the lock next.) Notice the thread then continues as part of a *new critical section*; the entire point is that the state of shared memory has changed in the meantime while it did not hold the lock.

- `notify` wakes up *one* thread that is blocked on this condition variable (i.e., has called `wait` on the same object). If there are no such threads, then `notify` has no effect, but this is fine. (And this is why there has to be no "gap" in the implementation of `wait`.) If there are multiple threads blocked, there is no guarantee which one is notified.

- `notifyAll` is like `notify` except it wakes up all the threads blocked on this condition variable.

The term *wait* is standard, but the others vary in different languages. Sometimes *notify* is called *signal* or *pulse*. Sometimes *notifyAll* is called *broadcast* or *pulseAll*.

Here is a WRONG use of these primitives for our bounded buffer. It is close enough to get the basic idea. Identifying the bugs is essential for understanding how to use condition variables correctly.

```
class Buffer<E> {
   // not shown: an array of fixed size for the queue with two indices
   // for the front and back, along with methods isEmpty() and isFull()
   void enqueue(E elt) {
      synchronized(this) {
         if(isFull())
           this.wait();
         ... do enqueue as normal ...
         if(... buffer was empty (i.e., now has 1 element) ...)
           this.notify();
       }
   }
  E dequeue() {
     synchronized(this) {
        if(isEmpty())
          this.wait();
        E ans = ... do dequeue as normal ...
        if(... buffer was full (i.e., now has room for 1 element) ...)
          this.notify();
        return ans;
     }
   }
}
```

Enqueue waits if the buffer is full. Rather than spinning, the thread will consume no resources until awakened. While waiting, it does not hold the lock (per the definition of `wait`), which

is crucial so other operations can complete. On the other hand, if the enqueue puts one item into an empty queue, it needs to call `notify` in case there are dequeuers waiting for data. Dequeue is symmetric. Again, this is the *idea* — wait if the buffer cannot handle what you need to do and notify others who might be blocked after you change the state of the buffer — but this code has two serious bugs. It may not be easy to see them because thinking about condition variables takes some getting used to.

The first bug is that we have to account for the fact that after a thread is notified that it should try again, but before it actually tries again, the buffer can undergo other operations by other threads. That is, there is a "gap" between the notification and when the notified thread actually re-acquires the lock to begin its new critical section. Here is an example bad interleaving with the `enqueue` in Thread 1 doing the wrong thing; a symmetric example exists for `dequeue`.

```
Initially the buffer is full (so Thread 1 blocks)


Thread 1 (enqueue)        Thread 2 (dequeue)      Thread 3 (enqueue)
------------------        ------------------      ------------------
if(isFull())
   this.wait();
                          ... do dequeue ...
                          if(... was full ...)
                              this.notify();
                                                  if(isFull()) (not full: don't wait)
                                                  ... do enqueue ...
... do enqueue ... (wrong!)
```

Thread 3 "snuck in" and re-filled the buffer before Thread 1 ran again. Hence Thread 1 adds an object to a full buffer, when what it should do is wait again. The fix is that Thread 1, after (implicitly) re-acquiring the lock, must again check whether the buffer is full. A second

if-statement does not suffice because if the buffer is full again, it will wait and need to check a third time after being awakened and so on. Fortunately, we know how to check something repeatedly until we get the answer we want: a while-loop. So this version fixes this bug, but is still WRONG:

```
class Buffer<E> {
   // not shown: an array of fixed size for the queue with two indices
   // for the front and back, along with methods isEmpty() and isFull()
   void enqueue(E elt) {
      synchronized(this) {
         while(isFull())
           this.wait();
         ... do enqueue as normal ...
         if(... buffer was empty (i.e., now has 1 element) ...)
           this.notify();
       }
   }
  E dequeue() {
     synchronized(this) {
        while(isEmpty())
          this.wait();
        E ans = ... do dequeue as normal ...
        if(... buffer was full (i.e., now has room for 1 element) ...)
          this.notify();
        return ans;
     }
   }
}
```

The only change is using a while-loop instead of an if-statement for deciding whether to `wait`.

Calls to `wait` *should always be inside while-loops that re-check the condition.* Not only is this in almost every situation the right thing to do for your program, but technically for obscure reasons Java is allowed to notify (i.e., wake up) a thread even if your program does not! So it is mandatory to re-check the condition since it is possible that nothing changed. This is still much better than the initial spinning version because each iteration of the while-loop corresponds to being notified, so we expect very few (probably 1) iterations.

Unfortunately, the version of the code above is still wrong. It does not account for the possibility of multiple threads being blocked due to a full (or empty) buffer. Consider this interleaving:

```
Initially the buffer is full (so Threads 1 and 2 block)

Thread 1 (enqueue)          Thread 2 (enqueue)      Thread 3 (two dequeues)
------------------          ------------------      -----------------------
while(isFull())
   this.wait();
                            while(isFull())
                              this.wait();
                                                    ... do dequeue ...
                                                    this.notify();

                                                    ... do dequeue ...
                                                    // no notification!
```

Sending one notification on the first dequeue was fine at that point, but that still left a thread blocked. If the second dequeue happens before any enqueue, no notification will be sent, which leaves the second blocked thread waiting when it should not wait.

The simplest solution in this case is to change the code to use `notifyAll` instead of `notify`. That way, whenever the buffer becomes non-empty, *all* blocked producers know about it and similarly whenever the buffer becomes non-full, *all* blocked consumers wake

up. If there are n blocked threads and there are no subsequent operations like the second dequeue above, then $n-1$ threads will simply block again, but at least this is correct. So this is RIGHT:

```
class Buffer<E> {
   // not shown: an array of fixed size for the queue with two indices
   // for the front and back, along with methods isEmpty() and isFull()
   void enqueue(E elt) {
      synchronized(this) {
         while(isFull())
           this.wait();
         ... do enqueue as normal ...
         if(... buffer was empty (i.e., now has 1 element) ...)
           this.notifyAll();
       }
   }
  E dequeue() {
     synchronized(this) {
        while(isEmpty())
          this.wait();
        E ans = ... do dequeue as normal ...
        if(... buffer was full (i.e., now has room for 1 element) ...)
          this.notifyAll();
        return ans;
     }
   }
}
```

If we do not expect very many blocked threads, this `notifyAll` solution is reasonable. Otherwise, it causes a lot of probably-wasteful waking up (just to have $n-1$ threads probably

block again). Another tempting solution ALMOST works, but of course "almost" means "wrong." We could have every enqueue and dequeue call `this.notify`, not just if the buffer had been empty or full. (In terms of the code, just replace the if-statements with calls to `this.notify()`.) In terms of the interleaving above, the first dequeue would wake up one blocked enqueuer and the second dequeue would wake up another one. If no thread is blocked, then the `notify` call is unnecessary but does no harm.

The reason this is incorrect is subtle, but here is one scenario where the wrong thing happens. For simplicity, assume the size of the buffer is 1 (any size will do but larger sizes require more threads before there is a problem).

1. Assume the buffer starts empty.
2. Then two threads T_1 and T_2 block trying to dequeue.
3. Then one thread T_3 enqueues (filling the buffer) and calls `notify`. Suppose this wakes up T_1.
4. But before T_1 re-acquires the lock, another thread T_4 tries to enqueue and blocks (because the buffer is still full).
5. Now T_1 runs, emptying the buffer and calling `notify`, *but in a stroke of miserable luck this wakes up* T_2 *instead of* T_4.
6. T_2 will see an empty buffer and wait again. So now T_2 and T_4 are blocked even though T_4 should enqueue and then T_2 could dequeue.

The way to fix this problem is to use *two condition variables* — one for producers to notify consumers and one for consumers to notify producers. But we need both condition variables to be "associated with" the same lock, since we still need all producers and consumers to use one lock to enforce mutual exclusion. This is why Java's choice to associate one condition variable with every lock is somewhat strange. However, the Java standard library has a `ReentrantLock` class that is more flexible. It has a `newCondition` method that

lets you associate as many condition variables as you want with the same lock. You then call `wait` and `notify` on these separate condition objects. In other languages and libraries, it is common for locks and condition variables to be separate things entirely, leaving it up to the programmer to manually associate each condition variable with a lock.

In any case, condition variables are:

- Important: Concurrent programs need ways to have threads wait without spinning until they are notified to try again. Locks are not the right tool for this.
- Subtle: Bugs are more difficult to reason about and the correct programming idioms are less intuitive than with locks.

In practice, condition variables tend to be used in very stylized ways. Moreover, it is rare that you need to use condition variables explicitly rather than using a library that internally uses condition variables. For example, the Java standard library already has a class `java.util.concurrent.ArrayBlockingQueue<E>` that is exactly what we need for a bounded buffer. (It uses the names `put` and `take` rather than `enqueue` and `dequeue`.) It is good to know that any blocking caused by calls to the library will be efficient thanks to condition variables, but we do not need to write or understand the condition-variable code.

8.7.3 Other

Locks and condition variables are not the only synchronization primitives. For one, do not forget `join` from the previous chapter, which synchronizes two threads in a particular way and is very convenient when it is the appropriate tool. You may encounter other primitives as you gain experience with concurrent programming. Your experience learning about locks should make it much easier to learn other constructs. Key things to understand when learning about synchronization are:

- What interleavings does a primitive prevent?
- What are the rules for using a primitive correctly?

- What are the standard idioms where a primitive proves most useful?

Another common exercise is to implement one primitive in terms of another, perhaps inefficiently. For example, it is possible to implement `join` in terms of locks: The helper thread can hold a lock until just before it terminates. The `join` primitive can then (try to) acquire and immediately release this lock. Such an *encoding* is poor style if `join` is provided directly, but it can help understand how `join` works and can be useful if you are in a setting where only locks are provided as primitives.

Here is an incomplete list of other primitives you may encounter:

- *Semaphores* are rather low-level mechanisms where two operations, called P and V for historical reasons, increment or decrement a counter. There are rules about how these operations are synchronized and when they block. Semaphores can be easily used to implement locks as well as barriers.

- *Barriers* are a bit more like `join` and are often more useful in parallel programming closer in style to the problems studied in the previous chapter. When threads get to a barrier, they block until n of them (where n is a property of the barrier) reach the barrier. Unlike `join`, the synchronization is not waiting for *one* other thread to *terminate*, but rather n other threads to *get to the barrier*.

- *Monitors* provide synchronization that corresponds more closely to the structure of your code. Like an object, they can encapsulate some private state and provide some entry points (methods). As synchronization, the basic idea is that only one thread can call any of the methods at a time, with the others blocking. This is *very* much like an object in Java where all the methods are synchronized on the `this` lock, which is exactly Java's standard idiom. Therefore, it is not uncommon for Java's locks to be called monitors. We did not use this term because the actual primitive Java provides is a reentrant lock even though the common idiom is to code up monitors using this primitive.

REFERENCES

[1] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Addison-Wesley, 2006.