Programming with Threads: Synchronization and Communication[*]

Daniel Heck[†]

v1.0β (April 4, 2011)

Contents

1 Introduction

One of the predominant models for writing concurrent applications is *multithreaded programming*. Here, concurrency is achieved by allowing a program to execute multiple code paths—the *threads*—in parallel. Compared to other models of concurrency, the defining property of threads is that all threads in a program have shared access to the program's memory.

Used correctly, threads are a powerful and efficient technique for solving a wide variety of concurrent programming tasks. But programmers new to multithreaded programming face a steep learning curve. First they have to deal with concurrency itself and develop suitable mental models for reasoning about things like parallel execution or the communication of concurrent entities. But in addition, they also have to unlearn proven techniques from sequential programming that are problematic in a concurrent setting. We will see, for example, that dynamic dispatch (one of the cornerstones of object-oriented programming) and the shared address space (one of the cornerstones of imperative programming) are a lot more difficult to handle once concurrency enters the picture.

The main purpose of this article is to collect general techniques for thinking about concurrency and for designing and programming with threads. The first three sections briefly review the fundamentals of multithreaded programming. We begin with a short overview of threads in general: how to start and stop them and how to control their execution. In Section 3, we discuss *mutexes*, which are a fundamental technique for controlling concurrent access to shared resources by preventing two threads from accessing the same resource simultaneously. More powerful synchronization mechanisms, such as waiting for other threads or actively negotiating the order in which shared resources are accessed, can be constructed by combining mutexes with *condition variables* (Section 4).

Although mutexes and condition variables are important building blocks and can be used to solve most synchronization and communication problems, they are fairly low-level techniques, and using them correctly requires a lot of attention to detail. Multithreaded programming becomes more manageable when we give up some of the flexibility of mutexes and condition variables and turn to higher-level synchronization and communication facilities. We will discuss two such facilities in the second half of the article.

[†]dheck@gmx.de

In Section 5 we cover *monitors,* which combine shared data, functions, mutexes, and condition variables into self-contained functional units that are quite similar to classes in object-oriented programming. Like classes, monitors encapsulate state and side effects, but in addition they deal with synchronization. Finally, Section 6 gives a brief introduction to *message passing*, a programming metaphor that decomposes programs into a set of concurrent entities that can interact exclusively by exchanging messages. Message passing is a powerful model for dealing with concurrency in general. In multithreaded programs, it encourages strongly decoupled threads and avoids many of the complications of shared-memory communication.

Examples in this article are presented in *Java*—not because the language is the best language for concurrent programming, but because it is ubiquitous and has solid support for threads. To make the representation self-contained, we briefly explain the threading facilities offered by Java as we go along. Many examples could be simplified by using specialized features provided by Java's standard library, but our focus is on the fundamentals of multithreaded programming, not on idiomatic Java programming. Some prior experience with threads and multithreaded programming is recommended but not strictly required.

2 Threads

Threads are the fundamental unit of execution in most operating systems. Every thread executes its own instruction stream concurrently with all other threads, but it shares resources like main memory, file handles, and sockets with all threads belonging to the same *process.* Processes correspond to programs being started and initially contain only one active thread. Additional threads must be explicitly launched by the program. When a process ends, all its threads—whether they are still active or not—are automatically terminated too.

In Java, threads are declared by deriving from `Thread` and putting the code to be executed into the `run` method (a subset of `Thread`'s interface is shown in Figure 1). Each object of a class derived from `Thread` corresponds to one operating system thread. Every thread obtains local copies of all non-static member variables declared in the class; setting these variables in the constructor makes it possible to pass information into the thread before it starts.

Newly created threads are inactive and must be explicitly brough to life by calling `start`. After that, they remain active until `run` returns or the thread is forcefully aborted. Once a thread has returned from `run` it is dead and cannot be restarted. All thread implementations provide a way to wait for a thread to stop; this operation is often called `join` because it merges two streams of execution.

```
public class Thread {
  Thread();
  void run();    // must be overloaded
  void start();
  void join() throws InterruptedException;
  void yield();
  void sleep(long millis) throws InterruptedException;
  void interrupt();
  boolean interrupted();
  ...
}
```

Figure 1: The core methods in Java's `Thread` class.

```
class ExampleThread extends Thread {
  public ExampleThread(String text, int n) {
    this.text = text;
    this.n = n;
  }
  void run() {
    for (int i=0; i<n; ++i)
      System.out.println(text + " " + i);
  }
  private String text;
  private int n;

  public static void main(String[] args) {
    Thread t1 = new ExampleThread("Hello", 50);
    Thread t2 = new ExampleThread("World", 100);
    t1.start();
    t2.start();
    t1.join();
    t2.join();
  }
}
```

Figure 2: Defining and starting threads.

The program in Figure 2 demonstrates how to define new threads by deriving from `Thread`, how to start them, and how to wait for them to run to completion. In this example, `join` acts as a simple synchronization mechanism: without the two calls to `join` the program would exit immediately, often before the two threads have executed a single line of code.

The operating system's scheduler is responsible for distributing CPU time among all active threads. If there are more threads than physical processors (as is usually the case), active threads are executed alternately for short slices of time, giving the impression of concurrent execution. Operating systems provide system calls to interact with the scheduler and influence how CPU time is assigned to threads. Java exposes only a small number of these system calls, and even those few are rarely used.

The simplest and most common way of interacting with the scheduler is to inform it when the current thread cannot do useful work. Java provides two methods in `Thread` to do this: `sleep` suspends the current thread for a specified number of milliseconds, whereas `yield` merely gives up the rest of the current time slice. Both methods are useful when a thread has to actively wait for an external event: instead of wasting CPU time and energy by checking for this event continuously, we can use `yield` or `sleep` to perform these tests at a more leisurely rate.[1]

A more indirect way of influencing the way a thread is scheduled is to modify its *priority*. The exact interpretation of priorities is system-dependent, but in general threads with a high priority get to run more often than those with a low priority. In Java, a thread's priority can be adjusted by calling `setPriority`, which accepts a parameter between `MIN_PRIORITY` and `MAX_PRIORITY`; new threads start with an intermediate priority `NORM_PRIORITY`. Priorities can sometimes be used for tuning the performance of finished applications, but they are rarely used otherwise.

Occasionally, interrupting threads from the outside can be useful, for example to give users the opportunity to cancel long-running operations. Most threading APIs provide a function to forcibly abort running threads, but doing so can have system-dependent or even undefined results. The `stop` method provided by Java threads is deprecated for this reason and shouldn't be used anymore.

As an alternative, Java provides a more structured way to interrupt a thread A from the outside, namely a method called `interrupt`. If A is currently active, `interrupt` merely sets a Boolean flag that can queried inside A using the `interrupted` method. If A is currently suspended inside a method such as `join`, `sleep`, or `wait`, `interrupt` doesn't set the flag but instead aborts these functions by throwing an `InterruptedException`. These two cases are complementary: either the thread is actively executing code and the flag is set, or the thread is suspended and an exception is thrown. The (few) methods in the JDK that can be aborted in this way are declared as throwing an `InterruptedException`. The following short example demonstrates how to safely check for both kinds of interruptions.

```
while (!interrupted()) {
  try {
    <<do some work>>
  } catch (InterruptedException e) {
    break;
  }
}
```

[1]Even this this weaker form of active waiting is usually frowned upon, however. For many kinds of external events, operating systems and libraries provide notification mechanisms that are cleaner and more efficient.

3 Mutual Exclusion

Since threads almost never run in isolation, they need some way to interact with the rest of the program. The most obvious solution is to use the shared address space for communication and data exchange. Unfortunately, this is also the most error-prone solution: when multiple threads access the same memory locations without proper synchronization, the results are typically undefined. Non-deterministic behavior and irreproducible crashes in multithreaded programs can often be traced back to the incorrect use of shared memory.

The fundamental mechanism for coordinating access to shared memory is the *mutex* (often called a *lock*). (Even though we will focus on synchronizing memory operations, the same arguments and techniques also apply to other types of shared resources such as files, network sockets, or peripheral devices.) Mutexes control access to a resource by "mutual exclusion" (hence the name): Threads can request exclusive access to a resource by *locking* the associated mutex and relinquish exclusive access by *unlocking* it. The code that is executed during the time a mutex is locked is also known as a *critical section*. If a thread attempts to lock a mutex that is already locked, this thread has to wait until the mutex is released by its current owner.

Mutual exclusion provides an easy way to protect shared resources against concurrent access from multiple threads: all we have to do is associate a mutex with the resource and ensure that code accessing the resource is only executed when this mutex is locked.

```
mutex.lock();
<<access resource protected by mutex>>
mutex.unlock();
```

In most cases, the shared resources of interest are simply variables or groups of variables, and mutexes are used to guarantee that these variables remain in a consistent state. Boolean predicates known as *invariants* are often used to specify what it means for data to be in a "consistent state," such as

- "`size` contains the number of elements in `array`;"
- "`list` contains a linked list of numbers in sorted order."

Invariants are basically contracts between different parts of a program. The first invariant, for example, demands that every method modifying either `size` or `array` must also make appropriate changes to the other.

When we say that a mutex "protects" a set of variables, we mean that the mutex is used to ensure that the invariants are true *whenever the mutex is not locked.* Note that invariants may be broken inside critical sections, as long as they are restored before the critical section ends. In this respect, mutex invariants are very similar to other invariants used in computer science:

loop invariants must be true only at the end of every iteration, and class invariants only at the end of every method. In all cases, invariants let us reason about the program's state and its correctness at discrete points in the program execution without having to consider every single intermediate step in the computation.

This focus on data and invariants lets us view mutexes from a slightly different angle. In most practical uses, *the main purpose of mutexes isn't to synchronize code execution but to synchronize data exchange.* Seen in this light, locking a mutex corresponds to acquiring the latest published version of the shared data, and unlocking the mutex to publishing the current contents of the data. This isn't merely a different interpretation of mutexes—it's a fairly accurate description of what mutexes actually do under the hood. To understand this, we need to take a quick look at the behavior of memory in modern computer architectures.

When writing single-threaded programs, memory appears to be a simple linear array of memory cells, and memory operations are performed in *program order*: given a sequence of instructions such as

```
a = b
c = a+1
```

we can be sure that reading `a` in the second line returns the value it was most recently assigned. Memory that behaves in this way is called *sequentially consistent.* Sequential consistency is a very desirable property because it simplifies reasoning about programs and the contents of memory, but it is a theoretical concept: for performance reasons, most modern computer architectures employ *weak memory models* that relax the strict constraints of sequential consistency. In weak memory models, different processors (and threads running on these processors) may disagree on the contents of memory and on the order in which memory operations are performed! Special processor instructions are provided to explicitly synchronize the memory contents when necessary.

Luckily, most programmers are shielded from such low-level details. To ensure that the contents of shared memory are properly synchronized at the beginning and end of critical sections, system libraries execute the correct synchronization instructions inside `lock` or `unlock`. As a result, we can assume that the memory contents are in a consistent state *as long as all accesses to shared memory regions are confined to critical sections.* But accessing shared memory *outside* of critical sections is undefined in most programming environments! Even seemingly simple operations like reading or writing a single shared variable from different threads can give unexpected results unless mutexes are used for synchronization. Due to their non-deterministic nature, program errors caused by unprotected access to shared variables are extremely difficult to reproduce and track down, so using critical sections correctly is mandatory in all multithreaded programs.

```
public interface Lock {
  void lock();
  void unlock();
  Condition newCondition();
  ...
}
```

Figure 3: The `Lock` interface, defined in the `java.util.concurrent.locks` package. This figure omits a few specialized variants of `lock()` that can be interrupted or that can time out.

3.1 Mutexes in Java

The `Lock` interface shown in Figure 3 specifies the methods supported by mutexes in Java. This interface provides `lock` and `unlock` to acquire and release the mutex; we will return to the `newCondition` method in Section 4.1. The `ReentrantLock` class provides the standard implementation of the `Lock` interface.

Let's illustrate the basics of threads and mutexes using a concrete example. Suppose we want to find the first prime number larger than some given lower bound n. The availability of efficient algorithms for primality testing permits an extremely simple solution: we scan the numbers $n+1, n+2, \ldots$ in sequence and stop as soon as the primality test is successful. For small n this search completes very quickly, due the density of primes and the efficiency of the underlying algorithms, but if n has a few hundred digits it becomes worthwhile to distribute the work over all available processors to speed up the search.

Figure 4 shows one possible solution to this problem which uses multiple threads to test individual numbers for primality in parallel. We want every number to be tested only once, and all threads to stop as soon as the desired prime number has been found. To do this, we employ two shared variables: `next` is the next number that hasn't been tested yet, and `solution` is either `null` or the smallest prime that has been found so far. The description of the two variables just given actually serves as the invariant protected by the `mutex` object.

The `main` method starts two threads (for simplicity we assume a two-processor machine), waits for both of them to complete, and finally outputs the solution. We must await the completion of all threads because the threads execute *asynchronously,* so we don't know which one will finish first—and either of them may end up finding the smallest prime. It is worth elaborating on this point. Asynchronous execution of threads means that we know nothing about their relative timing unless we use explicit synchronization. In this case, if p and q are two prime numbers and $p < q$, we do know that the

```
import java.util.concurrent.locks.*;
import java.math.BigInteger;

class PrimeFinder extends Thread {
  private static BigInteger next, solution;
  private static Lock mutex=new ReentrantLock();

  public static void main(String[] args)
    throws InterruptedException
  {
    Thread[] threads = { new PrimeFinder(),
      new PrimeFinder() };
    next = new BigInteger("10").pow(500);
    for (Thread t : threads) t.start();
    for (Thread t : threads) t.join();
    System.out.println(solution);
  }
  public void run() {
    while (true) {
      BigInteger n = getCandidate();
      if (n == null) break;
      else if (n.isProbablePrime(20)) {
        updateSolution(n);
        break;
      }
    }
  }
  static BigInteger getCandidate() {
    mutex.lock();
    BigInteger n = null;
    if (solution == null) {
      n = next;
      next = next.add(BigInteger.ONE);
    }
    mutex.unlock();
    return n;
  }
  static void updateSolution(BigInteger n) {
    mutex.lock();
    if (solution == null ||
        n.compareTo(solution) < 0)
      solution = n;
    mutex.unlock();
  }
}
```

Figure 4: A multithreaded program for finding the smallest prime number greater than a given lower bound. To keep the example simple, we have omitted obvious improvements like checking only odd n for $n > 2$.

primality test of p is started before the test of q since we use explicit synchronization when accessing `next`. But apart from that, the two threads aren't synchronized, so the thread testing q may finish before the thread testing p.

The `run` method contains the main loop of both threads. Inside this loop we repeatedly call an auxiliary method `getCandidate` to fetch the next number that must be tested for primality. If at least one solution has already been found, this method returns `null` and the thread stops. Otherwise we test for primality and, if necessary, update `solution` using a second auxiliary method `updateSolution`.

These two auxiliary methods encapsulate all code that accesses the shared variables. We preserve the invariant stated above by locking the mutex every time either method is entered and by ensuring that it is restored before the mutex is unlocked. The `getCandidate` method is fairly straightforward, but `updateSolution` is worth a second look. Similar to the `main` method, it has to deal with the asynchronous nature of threads: because the order in which the threads complete their work is undefined, we have to explicitly test whether `candidate` is the smallest prime number found so far.

In addition to explicit mutexes based on `Lock`, Java supplies every object with a mutex that can be locked and unlocked using the built-in `synchronized` statement. These mutexes are conventionally used for implementing monitors (see Section 5) and should rarely be used directly.

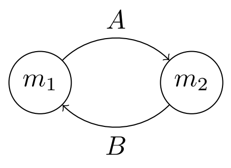
3.2 Deadlocks

One of the most notorious problems in concurrent programming is the *deadlock*, a circular dependency between threads in which every thread acquires exclusive access to a shared resource that is required by the next thread in the cycle. This results in a situation in which *all* threads wait for a resource to become available and prevent each other from making progress. The simplest deadlock involves two threads A and B that compete for two mutexes m_1 and m_2. Consider what happens if the threads lock the mutexes in opposite order, as shown in Figure 5 (a). The threads in this example can deadlock if A locks m_1 and B locks m_2 at approximately the same time: to continue, A has to wait until B releases m_2 and B until A releases m_1, so both threads are stuck.

In this simple example, it is comparatively easy to analyze the deadlock and understand its cause, but for more complex programs, analyzing the relative timing of lock operations to prevent deadlocks quickly becomes infeasible. Fortunately, there is an easier way to analyze deadlock conditions in practice. Figure 5 (b) shows the *dependency graph* of the program in Figure 5 (a). Every node in the graph corresponds to a mutex, and two mutexes x and y are connected by an arrow whenever a

```
void A() {          void B() {
  m1.lock();          m2.lock();
  m2.lock();          m1.lock();
  ...                 ...
}                   }
```

(a)



(b)

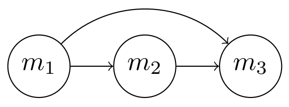
Figure 5: (a) A piece of code that can deadlock when A and B are executed in different threads. (b) Dependency graphs can be used to illustrate deadlocks.

single thread can potentially lock both mutexes at the same time. If this thread locks x before y, the arrow points from x to y and vice versa. In this example, thread A locks m_1 and then attempts to lock m_2, so there is an arrow from m_1 to m_2; because B locks m_2 first, there is also an arrow in the opposite direction.

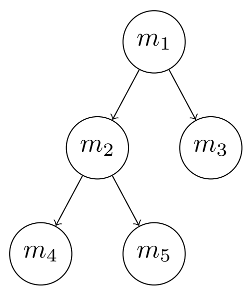
The most interesting property of dependency graphs is the correspondence between potential deadlocks and cycles in the graph. In mathematical terms, cycles are a necessary but not a sufficient condition for deadlocks: even though a cycle does not always lead to a deadlock, all deadlocks are caused by cycles. In other words, we can *prevent* deadlocks in concurrent programs by keeping the graph acyclic! In practice, this can be done by constraining the order in which mutexes are locked: the deadlock in Figure 5, for example, can be eliminated by requiring that A and B lock the two mutexes in the same order. The resulting dependency graph contains only one single arrow (either the one from m_1 to m_2 or the one from m_2 to m_1) but certainly no cycle.

The idea of restricting the order in which mutexes are locked can be extended to more general scenarios. First, consider a problem in which we have a set of shared resources R_i that are protected by mutexes m_i and several methods that work on *subsets* of these resources. For example, if the R_i represent bank accounts, a method responsible for performing money transfers would lock exactly those accounts involved in the transaction. To prevent deadlocks in such a situation, we can use a locking scheme such as the one in Figure 6 (a): the mutexes are put into a fixed linear order and all arrows point to the right, so that m_1 is *always* locked before m_3.

A second example of a general locking scheme is shown in Figure 6 (b). Hierarchical arrangement like this arise naturally when nested function calls access different mutexes. For example, mutex m_1 may be locked by a top-level method which then calls library functions that access mutexes m_2, m_4, m_5 and m_3 respectively. In this case, the edges in the dependency graph roughly correspond to method calls or hierarchical relationships between program modules. In such a hierarchical locking scheme we must be careful to avoid calls in the oppsite direction, for example due to callback mechanisms or cyclic relationships between modules.



(a)



(b)

Figure 6: The dependency graphs of two common locking strategies. (a) A linear lock order: the mutexes are always locked in a fixed order. (b) A hierarchical lock order: different mutexes are locked by different layers in the program hierarchy.

The dependency graphs resulting from either of these two schemes are *obviously* acyclic. That's the main advantage of this kind of analysis: dependency graphs allow us to ignore the complicated dynamic properties of the system (the exact configuration and relative timing that leads to a deadlock) and focus on mostly static ones (the order in which mutexes are used) that are much easier to deal with.

3.3 Contention

By design, mutexes *serialize* all accesses to the shared resource they protect: as long as a mutex is locked, all threads attempting to enter the critical section are temporarily suspended. This happens continually in concurrent program and usually isn't a problem. But when threads regularly impede each other by competing for the same mutex, they start to execute alternately instead of concurrently, and the application's performance can suffer. Such a situation, in which heavy access to a mutex prevents threads from running concurrently, is known as *lock contention.*

Contention is one of the primary reasons for performance issues in multithreaded programs, and it is also the source of the widespread myth that "locks are slow." In normal circumstances an uncontended lock/unlock pair takes only 10–200 clock cycles—few programs perform so many critical sections per second that this becomes a concern (and those that do often need a different design rather than faster locks). Contended locks, on the other hand, are slow. Even if the actual wait time is minimal, the operating system has to suspend the thread, schedule another thread, and start executing that thread, which easily consumes thousands of clock cycles. *If* locking affects a program's performance it's probably due to contention.

The likelihood of contention depends on how frequently the mutex is locked and how long it remains locked. If the average locking frequency is f and the average lock duration is T, the average number of threads that are inside the critical section or in the wait queue of the mutex is $N = fT$ (in queueing theory, this result is known as *Little's law*). As long as $N \ll 1$, the performance impact of locking is negligible, but for $N > 1$ contention will start to be a problem.

Let us briefly return to the prime finder example from Figure 4. The threads in this program use a single set of shared variables to synchronize their operation, and the associated mutex has to be locked and unlocked whenever any thread requires a new work package. Obviously, the frequency at which this mutex is accessed is roughly proportional to the number of threads that work in parallel. As long as there are only a few threads, the risk of contention is low, but the program doesn't *scale* because the likelihood of contention grows with the number of concurrent threads. A highly parallel implementation of the prime finder would therefore require a fundamentally different means of communication.

When designing concurrent applications, the following two guidelines can help reduce the risk of contention by keeping either f or T low:

- *Minimize access to shared resources.* Frequent access to shared resources not only increases the risk of contention but is also an indicator of scalability problems.

- *Unlock mutexes as quickly as possible* by performing only the absolute minimum of operations inside the critical section. For the same reason, library and system calls that can take a long time or even block (such as file I/O, `sleep`, or other synchronization functions) should be avoided inside critical sections.

Of course, the usual caveats regarding premature optimization also apply to lock contention—probably even more so, since errors in concurrent programs are so much harder to fix.

3.4 Recursive Locking

The mutexes provided by Java are so-called *recursive* (or *reentrant*) mutexes. The defining property of such mutexes is that they can be locked multiple times by the same thread. Figure 7 illustrates this by calling the `common` method from `example`'s critical section. If `mutex` is non-recursive, the second call to `lock` inside `common` suspends the thread and waits (indefinitely) for the mutex to be released. If `mutex` is recursive, however, the second call to `lock` detects that the current thread is already inside the critical section and returns without deadlocking.

```
public void example() {   public void common() {
  mutex.lock();             mutex.lock();
  common();                 <<shared code>>
  mutex.unlock();           mutex.unlock();
}                         }
```

Figure 7: Recursive mutexes can be locked multiple times from the same thread. If `mutex` is non-recursive, the call to `common` deadlocks the thread.

Recursive mutexes maintain an internal counter that is increased when `lock` is called and decreased when `unlock` is called. A call to `lock` enters the critical section only if this counter is currently zero, and a call to `unlock` leaves the critical section only if the counter returns to zero. In all other cases `lock` and `unlock` merely modify the counter. It is worth iterating that this only applies to repeated locking operations from the same thread—attempting to lock a mutex currently held by another thread always blocks as expected.

At first sight, recursive mutexes seem to be attractive: they prevent a certain class of deadlocks, facilitate code reuse, and—as we will see when we discuss monitors in Section 5—because they can greatly simplify certain concurrent code. Nevertheless, it's often better to use non-recursive mutexes and to structure the code in such a way that locks need not be acquired multiple times. As we will now discuss, the semantics of non-recursive mutexes may appear inconvenient at first sight, but they often lead to simpler code and stricter locking policies.

The first advantage of non-recursive mutexes is, in fact, *that* they can cause deadlocks. The reason is that *accidentally* locking mutexes more than once is a relatively common error in multithreaded programming, especially in object-oriented code with complex call hierarchies. The deadlock caused by non-recursive mutexes makes such problems trivial to detect and debug. Even though the likelihood of such locking mistakes can be reduced by keeping the program's control flow simple (for example, by not calling unknown code such as external libraries or virtual methods inside critical sections), the little bit of extra safety provided by non-recursive mutexes is often helpful.

The second and more important point is that recursive locking makes it harder to keep track of data invariants. As we have already mentioned, one crucial feature of these invariants is that we have to consider them at two discrete points only: when the mutex is acquired and when it is released. Because `lock` and `unlock` are used in pairs, it is usually evident from the code where the critical section begins and where it ends. This is no longer true when recursive locking is used: a recursive mutex may already be locked when `lock` is called, and it may remain locked after `unlock` has finished. It is no longer sufficient to examine localized pieces of code when reasoning about invariants—instead, we have to analyze a program's call graph and the interaction of different methods.

Both of these arguments against recursive mutexes stem from the fact that the temporal extent of critical sections is easier to control and easier to understand if recursive locking isn't used. Unfortunately, several modern programming languages (such as Java and C#) provide only recursive mutexes—here, it is the programmer's responsibility to avoid nested locking and complicated critical sections.

4 Condition Variables

Multithreaded programs often require synchronization mechanisms that go beyond mutual exclusion. We already encountered one such operation in Section 2: `join` allows one thread to wait for another thread to stop. In this section we will discuss a far more general method for coordinating threads that builds on two fundamental operations: a *wait* operation that suspends threads until a certain event occurs and a *notify* operation that signals the occurrence of such an event.

There a several different synchronization primitives that provide these two basic operations, such as *semaphores*, *events*, and *condition variables*. Semaphores are a flexible synchronization primitive provided by almost every operating system, but their semantics are a bit tricky and can easily cause programmer errors. Events are mainly used on the Windows platform; their main disadvantage is that they can lead to subtle race conditions, especially when data needs to be exchanged along with the notification. Condition variables avoid both of these problems and are the simplest and safest choice for most applications.

The canonical example for using condition variables are message queues that transfer data between threads in a first-in first-out (FIFO) fashion. The *sender* thread appends items to the end of the queue, and the *receiver* thread removes items from the front. We assume that the queue can grow arbitrarily long so that sending an object always succeeds. But what happens when the receiver encounters an empty list? Condition variables allow us to handle this case by using *wait* to suspend the thread when the queue is empty. The sender can then use *notify* to wake up a sleeping receiver whenever it makes a new object available.

Of course, we have to ensure that only one thread at a time can access the shared queue data structure. If we use a mutex for this purpose, it must interact correctly with the *wait* and *notify* operations. Most importantly, the receiver must unlock the mutex before suspending itself using *wait*—otherwise, the sender wouldn't be able to enter the critical section and insert new items into the queue.

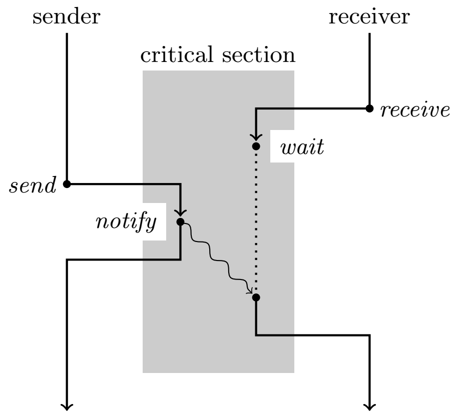


Figure 8: This figure illustrates the interaction of threads when using condition variables. Time increases from top to bottom. When the *receive* operation encounters an empty queue, it calls *wait*. This suspends the thread and unlocks the mutex protecting the critical section, as indicated by the dotted line. The sender thread can now enter the critical section and insert a new message into the queue. The notification emitted by the sender doesn't reactivate the receiver immediately but is delayed until the sender leaves the critical section.

The beauty of condition variables is that they automate this process: *wait* automatically unlocks a mutex before suspending the thread and reacquires it after being woken up. This means we can call *wait* in the middle of a critical section, temporarily grant exclusive access to the shared data to another thread, and when the execution resumes after the thread is notified, we are back inside the critical section where we left off. This is the crucial idea behind condition variables: they give a controlled means of relinquishing exclusive access to a shared resource while waiting for notifications. Figure 8 illustrates the operation of condition variables and their interaction with critical sections.

Internally, condition variables maintain a list of threads that are currently waiting for notification. The *notify* operation takes one thread from this list and wakes it; if there are multiple waiting threads, which of them is chosen is implementation-defined. Note that waking a thread doesn't imply that it becomes active immediately, only that it is made eligible for execution; the exact time it is resumed is determined by the operating system.

In addition to *notify*, condition variables also provide an operation called *broadcast* that wakes all waiting threads. We will discuss this in detail in Section 4.3.

4.1 Condition Variables in Java

Java provides an interface called `Condition` for dealing with condition variables (Figure 9). This interface provides the equivalents of *wait*, *notify*, and *broadcast*,

```
public interface Condition {
  void signal();
  void signalAll();
  void await() throws InterruptedException;
  ...
}
```

Figure 9: The three most common methods in Java's `Condition` interface. In addition to the three shown methods, the interface also contains a few variants of `await` that can time out or that cannot be interrupted.

although it calls them `await`, `signal`, and `signalAll`[2]. Condition variables cannot be instantiated directly but must be created by calling the `newCondition` method of an existing mutex. The returned condition variable is inextricably bound to this mutex, and every call to `await` automatically performs the necessary unlock and lock operations on it.

Figure 10 shows a Java implementation of a *bounded queue* using condition variables. A bounded queue is a message queue as discussed in the previous section, but with the additional property that it has a maximum size: if the queue is full, sending a message is delayed until the receiver consumes one or more messages. The example uses two condition variables: `notfull` represents the event that the queue isn't full, and `notempty` the event that it isn't empty.

The `receive` method removes a single element from the front of the queue and returns it. If it encounters an empty queue it uses the `notempty` condition to wait until new items become available. Otherwise, it removes the front element from the queue and generates a `notempty` notification to inform other threads that there is now at least one empty slot in the queue. The `send` method is an almost exact mirror image of `receive`, except that we wait until the queue isn't *full* before we append a new item and that we signal `notempty` afterwards.

In Section 2 we discussed Java's mechanism for interrupting threads using `Thread.interrupt`. Since the `await` method also belongs to the set of functions that can be interrupted, we must handle potential `InterruptedExceptions`. One basic guarantee of condition variables is that the associated mutex is *always* locked when `await` returns, even if it returns prematurely due to an interruption. We therefore use a `finally` block to ensure that `mutex` is properly unlocked at the end of `receive` and `send`.

[2]As we will see in Section 5.1, all classes derived from `Object` contain `wait` and `notify` methods. Be careful to use `await` and `signal` when dealing with `Conditions`.

```
class BoundedQueue<Data> {
  int maxsize;
  Queue<Data> queue = new LinkedList<Data>();
  Lock mutex = new ReentrantLock();
  Condition notfull = mutex.newCondition();
  Condition notempty = mutex.newCondition();

  BoundedQueue(int maxsize) {
    this.maxsize = maxsize;
  }
  Data receive() throws InterruptedException {
    mutex.lock();
    try {
      while (queue.peek() == null)
        notempty.await();
      Data d = queue.poll();
      notfull.signal();
      return d;
    } finally {
      mutex.unlock();
    }
  }
  void send(Data d) throws InterruptedException {
    mutex.lock();
    try {
      while (queue.size() == maxsize)
        notfull.await();
      queue.offer(d);
      notempty.signal();
    } finally {
      mutex.unlock();
    }
  }
}
```

Figure 10: Implementation of a bounded message queue using one mutex and two condition variables.

4.2 Waiting in Detail

Alert readers may have noticed that the code in Figure 10 encloses the calls to `await` in a `while` loop, even though a simple `if` statement would appear to be sufficient. This is not an accident: in Java, and many other programming environments, using only an `if` statement is an error. This is one of the trickiest aspects of condition variables, and a consequence of the internal operation of *wait*.

The condition variable's *wait* operation performs three tasks: leave the critical section, wait for an event, and re-enter the critical section after waking up, as in the following piece of code (the hypothetical `SuspendAndWait` method suspends the thread and waits for a notification):

```
mutex.unlock();
SuspendAndWait();
mutex.lock();
```

Unfortunately, however, this naive implementation is prone to a race condition and can sometimes miss notifications.

The problem is that calls to `unlock` and `SuspendAndWait` are separate operations, allowing another thread to enter the critical section before the current thread is suspended. All notification sent via *notify* during this short time interval are lost because the thread hasn't been added to the list of waiting threads yet! In the case of the `BoundedQueue` in Figure 10 this means that a thread executing `receive` could miss a `notempty` notification and suspend itself even though a new item has just been inserted into the queue. In the worst case, no additional messages are sent later on and the thread remains suspended indefinitely.

To guarantee that notifications are never lost, condition variables must implement *wait* by releasing the mutex and suspending the thread in one indivisible step.

The second half of *wait* still consists of two separate steps, however: the thread wakes up, and then attempts to reacquire the mutex. A thread awoken by a notification isn't granted privileged access to the mutex but competes with all other threads that are trying to enter the critical section. This has an important consequence: conditions that were true the moment the notification was sent are *not* guaranteed to be true when the notified thread eventually returns from *wait*: another thread may have obtained the lock in the meantime and modified any shared variable. It is easy to guard against such surprises by re-checking the wait condition after *wait* returns. In most cases we already have to check the condition *before* calling *wait*, so we can simply change the surrounding `if` statement into a loop. This is the first reason for using a `while` loop in Figure 10.

But what if there simply is no third thread that could modify shared variables between the call to *notify* in one thread and the return from *wait* in the other thread–do we still have to re-check the wait condition in this case? The surprising answer is yes, but for a completely unrelated, unexpected reason: in most implementations of condition variables, *wait* may return *spontaneously*, without any thread calling *notify*! These unexpected returns from *wait* are called *spurious wake-ups.* Even though they occur rarely in practice, spurious wake-ups are real and can cause real problems if they aren't handled correctly. For this reason *all* calls to *wait* should be enclosed in a loop checking the wait condition:

```
while (!expression)
  condition.await();
```

Spurious wake-ups may appear to be a nuisance, but they are ultimately a good thing: having to explicitly write down the wait conditions before every call to *wait* forces us to specify exactly how threads interact and which shared variables they use for communication. When this information is only available implicitly—in the way the different threads operate and modify shared variables—concurrent programs are much harder to understand. Another advantage of wait conditions is that they clearly state the precondition for the code following the call to *wait*. In the `BoundedQueue` example, we don't need to investigate other parts of the program to convince ourselves that the queue isn't empty or isn't full in the code following `await`—it is evident. Even if spurious wake-ups didn't exist, it would be good programming practice to check the wait condition in a loop.

4.3 Broadcasts

So far we have only briefly mentioned the *broadcast* operation but haven't explained how to use it. Broadcasting means that *all* threads currently waiting for a notification from a condition variable are awoken. After waking up, all these threads attempt to lock the mutex associated with the condition variable, but the order in which they eventually enter the critical section is undefined.

One common application of condition variables is to manage limited resources in a program: in the `BoundedQueue` from Figure 10, for example, the `notempty` condition tracks the occupied queue slots, and the `notfull` condition the empty slots. Threads that want to acquire such resources can *wait* for a notification if none are currently available. When *exactly one* resource becomes available, *notify* can be used to inform one waiting thread. But when multiple of the managed resources become available at once, we *have* to use the *broadcast* operation instead of *notify*.

Consider an extension of `BoundedQueue` that contains an additional `receiveAll` method that removes all available items from the queue (see Figure 11), thereby freeing all of the message slots at once. If multiple threads are waiting for a `notfull` notification, calling `signal` will activate only one of them and leave the others suspended—in spite of the fact that we just emptied the complete queue and (potentially) all of them could progress. In this case, we obviously have to use `signalAll` to ensure correct operation.

The situation in which a thread waits forever for an external event is known as *starvation.* In the `BoundedQueue` example we just discussed, threads waiting for `notfull` can starve if `receiveAll` doesn't use `signalAll`: it's possible in principle that such a thread won't receive a notification for the rest of its lifetime. Starvation and deadlocks are two forms of *liveness failures*, conditions in which part(s) of the program cannot make progress.

The second situation in which *broadcasts* are commonly used is when condition variables are used to monitor state changes. This technique is illustrated in Figure 12, which shows a variant of the bounded queue from Figure 10. This implementation combines the two

```
class BoundedQueue {
  ...
  public Queue<Data> receiveAll() {
    mutex.lock();
    Queue<Data> r = queue;
    queue = new LinkedList<Data>();
    notfull.signalAll();
    mutex.unlock();
    return r;
  }
}
```

Figure 11: Broadcasts are used to inform multiple threads when more than one shared resource becomes available. In this example, calling `signal` instead of `signalAll` could cause starvation if multiple threads are waiting for a notification from `notfull`.

condition variables `notfull` and `notempty` into one single condition variable `state`. This condition variable is then used to send notification whenever the internal state of the message queue changes, i.e., when an item is appended or removed.

In this case we have to use `signalAll` instead of `signal` because—now that the condition variables are combined—a notification sent by `receive` can potentially wake up either a thread waiting inside `send` (this is what we want) or a thread waiting inside `receive` (this isn't)[3]. In general, keeping track of waiting threads and notifications is more difficult when condition variables are used to monitor state rather than single resources. For this reason it's advisable to *always* broadcast state changes, at least if more than two threads are involved.

In principle, it is possible to avoid *notify* altogether and always use *broadcast* for sending notifications. As long as wait conditions are checked correctly, the worst thing that can happen is that *broadcast* unnecessarily wakes up threads that cannot do useful work and immediately call *wait* again. The advantage of using *broadcast* exclusively is that it avoids the liveness failures that can result when too few threads or the wrong threads are awoken, as in the two examples we just discussed.

The main disadvantage of using only *broadcast* is that it can cause performance issues if many threads are involved or the broadcasts happen frequently. When a large group of threads wakes up simultaneously and competes for CPU time and the shared mutex, the associated overhead can keep the whole computer busy for a noticeable amount of time; this is also called, quite fittingly, the *thundering herd problem.*

The second argument against using *broadcast* indiscriminately is that *notify* can communicate the programmer's intention more clearly. This is the same argument we used to argue in favor of non-recursive mutexes and explicit wait conditions: every bit of additional information about the way threads interact makes concurrent programs easier to understand.

[3]It is surprisingly difficult to show that that it's even possible for two threads to be suspended in `receive` and `send` at the same time; the construction relies on the mutexes being *unfair*, a property which we won't discuss in this article.

```
class BoundedQueue<Data> {
  Queue<Data> queue = new LinkedList<Data>();
  Lock mutex = new ReentrantLock();
  Condition changed = mutex.newCondition();
  int maxsize

  BoundedQueue(int maxsize) {
    this.maxsize = maxsize;
  }
  Data receive() throws InterruptedException {
    mutex.lock();
    try {
      while (queue.peek() == null)
        changed.await();
      Data d = queue.poll();
      changed.signalAll();
      return d;
    } finally {
      mutex.unlock();
    }
  }
  void send(Data d) throws InterruptedException {
    mutex.lock();
    try {
      while (queue.size() == maxsize)
        changed.await();
      queue.offer(d);
      changed.signalAll();
    } finally {
      mutex.unlock();
    }
  }
}
```

Figure 12: A variant of the bounded queue from Figure 10 that uses a single condition variable instead of two.

The present discussion can be summarized as follows. Use *notify* if only one thread needs to be awoken and if you are sure that all of the waiting threads can handle the notification correctly (if condition variables are used to track individual resources, these two conditions are often fulfilled). In all other cases, or if you are unsure whether *notify* suffices, use *broadcast*, even if this potentially wakes too many threads. Like lock contention and other performance issues, the overuse of *broadcast* is best handled at the design level and not by micro-optimizations that potentially compromise the correctness of the program.

```
class Monitor {
  // Fields should always be private.
  private int field;

  // Constructors do not need to be synchronized.
  public Monitor() { ... }

  // Only one thread at at time can execute
  // any of the synchronized methods.
  public synchronized void method1() { ... }
  public synchronized void method2() { ... }
}
```

Figure 13: Declaring monitors in Java.

5 Monitors

There are many ways to use mutexes incorrectly such as locking or unlocking them at the wrong time (or not at all), or locking multiple mutexes in the wrong order. Because mutexes are only loosely coupled to the resources they protect, programming tools and languages are of limited help with preventing or tracking down such problems. *Monitors* offer a more structured alternative to mutexes and prevent many programming errors by making mutually exclusive access simpler or even (with a little language support) automatic.

A monitor consists of a set of shared variables and methods that manipulate these variables—similar to a class in object-oriented programming, but with thread synchronization weaved in. We say that a thread *enters* the monitor by calling one of its methods and *leaves* it when the method returns. The main property of a monitor is that *only one thread may be inside the monitor at any time.* In other words, monitors provide a form of mutual exclusion that, in contrast to mutexes, doesn't rely on manual *lock* and *unlock* operations but is tied to the duration of method calls. Simply by calling one of the monitor's methods a thread obtains exclusive access to the monitor for as long as this method is active.

5.1 Monitors in Java

Java's built-in support for monitors is tightly integrated with the class system. Every Java class can be turned into a monitor by adding the `synchronized` keyword to every method declaration and every object instantiated from such a class is an independent monitor. Only one thread at a time is allowed to execute the synchronized methods in a monitor. The member variables of a monitor should always be declared as private to prevent unsynchronized access from the outside. The general structure of monitors in Java is shown in Figure 13.

We already met two monitors in disguise in previous sections, namely the two `BoundedQueue` classes in Figures 10 and 12. Both classes used manual locking to

```
class BoundedQueue {
  private Queue<Data> queue =
    new LinkedList<Data>();
  private int maxsize;

  BoundedQueue(int maxsize) {
    this.maxsize = maxsize;
  }
  public synchronized
  Data receive() throws InterruptedException {
    while (queue.peek() == null)
      wait();
    Data d = queue.poll();
    notify();
    return d;
  }
  public synchronized
  void send(Data d) throws InterruptedException {
    while (queue.size() == maxsize)
      wait();
    queue.offer(d);
    notify();
  }
}
```

Figure 14: A bounded queue implemented using Java's built-in monitor support. This implementation is equivalent to the code in Figure 12, but this time Java generates the boilerplate code for locking, unlocking, and handling exceptions automatically.

guarantee that only one thread at a time could execute either the `send` or the `receive`. The `synchronized` keyword allows a significantly more compact implementation, as shown in Figure 14.

Internally, monitors implement mutual exclusion using a hidden mutex: calling a synchronized method locks this mutex, and returning from the method (or throwing an exception) unlocks it. We can therefore think of a synchronized method like

```
synchronized void method() {
  <<method body>>
}
```

as a shorthand for the following synchronization code:

```
void synchronized_method() {
  Lock mtx = <mutex associated with object>;
  mtx.lock();
  try {
    <<method body>>
  } finally {
    mtx.unlock();
  }
}
```

In addition to the hidden mutex, every Java monitor contains a hidden condition variable that can be accessed using the following methods defined in `Object`.

```
class Object {
  ...
  public void wait() throws InterruptedException;
  public void notify();
  public void notifyAll();
}
```

These methods correspond to `await`, `signal`, and `signalAll` in the `Condition` interface. Java only provides a single condition variable per monitor, so there is no direct translation of our first implementation of bounded queues in Figure 10 which used two condition variables `notempty` and `notfull`.

The mutexes associated with Java monitors are recursive, so once inside the monitor it's possible to call other methods in the same monitor without risking a deadlock. Deeply nested method calls inside monitors should be avoided, however, for the same reasons we cautioned against complex critical sections throughout Section 3: needless complexity increases the likelihood of contention and makes it harder to keep track of invariants.

Calls to synchronized methods lock the monitor for the entire duration of the method call, even if only a few instructions access the monitor's state. This *coarse-grained* locking is one of the main reasons monitors are easier to use and safer than mutexes, but it can occasionally lead to performance problems. Java therefore also features a `synchronized` *statement* that protects only selected blocks of code:

```
synchronized (this) {
  <<code>>
}
```

This statement enters the monitor and executes the code inside the curly braces. Control flow operations that leaves the code block (like exceptions, `return`, or `break`) automatically also leave the monitor. Because `synchronized` blocks are little more than plain critical sections, they return the full responsibility for synchronizing access to shared variables back to the programmer. For this reason they should be used sparingly, and only if they offer measurable performance benefits.

The `synchronized` statement is actually more general than we have indicated: it can be used to lock arbitrary objects, not just `this`. This feature is mainly a remnant from the earlier days of Java where explicit locks like `ReentrantLock` weren't available. Today, it is better to reserve the `synchronized` notation for monitor operations locking `this` and use explicit locks in all other cases.

5.2 Monitors and Classes

The discussion of the mechanics of monitors in the previous section may have left you with the impression that monitors are little more than syntactic sugar for mutexes and condition variables. Even though this is true to a certain extent, it misses the point: the main advantage of monitors is that they combine data and functionality into functional units that we can use to structure concurrent programs. We will explore this aspect in the current and the following section.

Monitors were originally inspired by classes in Simula, so it isn't surprising that the two concepts share many similarities (and, in the case of Java, almost the same syntax). Due to this close relationship, monitors are sometimes referred to as "thread-safe classes," but we avoid this term because the term "thread-safe" is imprecise and because the addition of synchronization is a such a significant departure from regular class semantics that it becomes important to use a distinctive name.

The main conceptual difference between classes and monitors is that it's much harder to make assumptions about the internal state of a monitor. Let's illustrate this with a simple example. In a sequential program, it is reasonable to write the following code to replace the top-most element of the stack with a modified copy.

```
if (!stack.isEmpty()) {
  Data x = stack.pop();
  <<modify x>>
  stack.push(x);
}
```

The implicit assumption in the example (which is trivially true in sequential programs) is that the stack doesn't spontaneously become empty after `isEmpty` has returned `false`, and that no new elements are inserted between the calls to `pop` and `push`. Unfortunately, this kind of reasoning breaks down in a concurrent program, even if we turn `stack` into a monitor by making the three methods mutually exclusive. The problem is that a nonempty stack *can* spontaneously become empty and new elements *can* suddenly appear if another thread happens to become active at the "wrong" time. The whole operation must be enclosed in a single critical section to make it work correctly.

In a certain respect, the design goals for classes and monitors are therefore almost opposite. For classes, it is often desirable to keep number of methods to a minimum and their functionality orthogonal because classes with simple semantics and interfaces are usually easier to extend and compose than high-level, monolithic classes. Monitors, on the other hand, usually gravitate towards high-level functionality. Even though it would be desirable to provide the same level of extensibility and composability, the presence of concurrency puts much stronger constraints on the public interface of monitors.

For this reason, monitors that expose low-level functionality or internal state should be regarded with suspicion: as in the example we just discussed, monitors with a low level of abstraction are prone to hidden race conditions. The existence of predicates such as `isEmpty`,

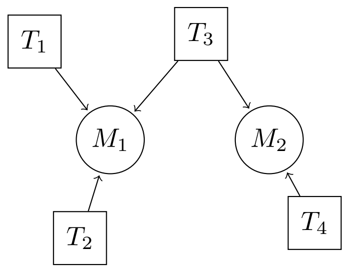


Figure 15: A simple model for managing the interaction of threads $T_1, \ldots, T_4$ using two monitors M_1, M_2.

"getter" and "setter" functions, iterators, or the need for external locking often betrays such half-baked monitors.

5.3 Using Monitors

In concurrent programs, monitors are often used to glue together cooperating threads, as illustrated schematically in Figure 15: the threads $T_1, \ldots, T_4$ run autonomously and interact *exclusively* using the monitors M_1, M_2 connecting them. The main benefit of this organization is that it neatly decouples the *communication logic* from the rest of the program logic. If all data exchange and synchronization is encapsulated inside monitors, the threads behave very much like sequential processes that interact with the rest of the program only through the interfaces provided by the monitors.

The prime searching example from Section 3.1, for example, can be rewritten in this style by moving the shared data and the methods accessing it into a new monitor `State` and using this monitor from both threads. The resulting program is shown in Figure 16. Note how nicely the `PrimeFinder` class is isolated from the mechanics of thread interaction.

Autonomous monitors, such as the `State` class, are most useful when the data protected by the monitor is shared symmetrically between multiple threads. When data isn't shared symmetrically but owned by one of the threads, it can be convenient to integrate the monitor into the class implementing the thread. An example is shown in Figure 17. The main disadvantage of this widely used approach is that it blurs the distinction between shared and non-shared variables. One typical mistake to watch out for is accessing shared variable directly from unsynchronized methods. Inside `BackgroundThread.run`, for example, it is an error to access `isPaused` without synchronization because the variable resides in shared memory. Instead, the synchronized `paused` method must be used.

Monitors occasionally have to interact with other code such as external libraries, the operating system, or other classes and monitors defined in the same program. When calling such code from a monitor we must be careful to avoid contention, deadlocks, and race con-

```
class PrimeFinder extends Thread {
  class State {
    private BigInteger next, solution;

    State(BigInteger startValue) {
      next = startValue;
    }
    synchronized BigInteger getSolution();
    synchronized BigInteger getCandidate();
    synchronized void updateSolution(BigInteger);
  }
  private State state;

  public PrimeFinder(State state) {
    this.state = state;
  }
  public void run() {
    while (true) {
      BigInteger n = state.getCandidate();
      if (n == null) break;
      else if (n.isProbablePrime(10)) {
        state.updateSolution(n);
        break;
      }
    }
  }

  public static void main(String[] args)
      throws InterruptedException {
    State s = new State(new BigInteger("100000"));
    Thread[] threads = { new PrimeFinder(s),
      new PrimeFinder(s) };
    for (Thread t : threads) t.start();
    for (Thread t : threads) t.join();
    System.out.println(s.getSolution());
  }
}
```

Figure 16: The example from Figure 4 rewritten using a monitors that encapsulates thread interaction.

ditions. This is usually manageable, as long as it is clear what side effects the code has and how it interacts with threads and synchronization. But from sequential programs we are used to a large array of techniques for dynamic dispatch, such as callbacks, function pointers, virtual functions, and several *design patterns*. Combining such techniques with monitors can easily lead to trouble.

To illustrate the potential problems, let's consider the well-known *Observer* pattern. In sequential programs, the purpose of this pattern is to inform multiple interested parties (the *observers*) about changes to an object of interest (the *observable*) by calling their `update` method. To use this pattern in a concurrent program, we could try to define a monitor `SyncObservable` that informs a list of observers when `notifyObservers` is called. Figure 18 shows a possible implementation.

```
class BackgroundThread extends Thread {
  private boolean isPaused;

  public synchronized
  void setPaused(boolean p) {
    isPaused = p;
    notify();
  }
  private synchronized boolean paused() {
    return isPaused;
  }

  public void run()
      throws InterruptedException {
    while (true) {
      // don't access isPaused directly!
      while (paused())
        wait();
      <<do some work>>
    }
  }
}
```

Figure 17: When data owned by individual threads must be protected, it can be convenient to combine the thread implementation and the monitor into a single class. But only methods declared as synchronized may access the shared variables!

```
class SyncObservable {
  private List<Observer> observers
    = new ArrayList<Observer>();

  public synchronized
  void notifyObservers() {
    for (Observer o : observers)
      o.update(this);
  }
  public synchronized
  void addObserver(Observer o) {
    observers.add(o);
  }
  ...
}
```

Figure 18: An attempt to implement the *Observer* pattern as a monitor; the class roughly follows the interface of Java's `java.util.Observable`.

The main problem with `SyncObservable` is that it is not clear how the observers and their `update` methods must behave to guarantee correctness. Is it safe for the observers to access the `SyncObservable` class they observe, even though it is technically still locked when `update` is called? What happens if one of the observers blocks or waits for another thread? In addition, since the observers run their code in the same thread that modified the observable, how do we guarantee that the execution of `update` cannot cause race conditions or deadlocks? As long as we do not know exactly what the observers do inside `update`, it's impossible to make guarantees about the safeness and correctness of `SyncObservable`.

The purpose of the original Observer pattern is to decouple observables and observers, so why does the implementation as a monitor lead to much tighter coupling? When sequential programs use dynamic dispatch, the unknown code is usually called for its side effects. The problem is that in multithreaded programs side effects are hard, and composing code with side effects is a harder still. Because the main purpose of monitors is to encapsulate such side effects, it's hardly surprising that they do not mix particularly well with dynamic dispatching.

6 Message Passing

Most threads don't run in isolation and need to communicate with the rest of the program. With the tools we have discussed so far, this communication is often indirect: to pass information from one thread to another, we store it in shared memory and send a notification to the receiver that new data is available. In this section, we discuss *message passing,* a programming model that focuses on direct communication between threads.

Message passing is an abstract programming model: its basic units are the *actors* that communicate (the term *process* is also common), the *messages* that are being exchanged, and the *channels* that transfer the messages. The crucial property of message passing models is that the actors are autonomous self-contained units: their state and internal operation is strictly encapsulated and the only way for them to interact is by exchanging messages. Note that this description doesn't mention threads, shared memory, or even computers at all: the ideas behind message passing apply to the interaction of concurrent objects in general, whether they are mathematical abstractions, physical devices, computers in a network, or threads in a program.

As an example, Figure 19 shows how to model the operation of a simple telephone using message passing. We assume that there are three concurrent actors: the user, the carrier, and the phone. Pressing a button on the phone sends a message such as "digit" or "hang up,"

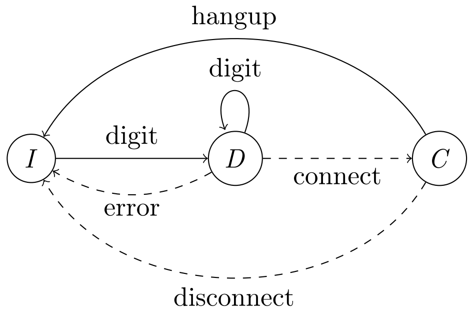


Figure 19: A simplified state machine model of a telephone. The phone has three internal states: *idle (I)*, *dialing (D)*, and *connected (C)*. External messages by the user (solid edges) or the carrier (dashed edges) cause transitions between the phone's states.

to the phone; likewise, the carrier sends messages such as "connect" or "disconnect."

The telephone starts in the *idle* state I, but when it receives a *digit* message, it transitions into its *dialing* state D. Depending on the messages it receives in this state, it may either return to the idle state—for example if the user dialed an invalid telephone number or if the signal is too weak—or move to the *connected* state C. The three states I, D, and C represent the three main modes of operation of the telephone and determine how it interacts with the other actors in the system. State machines such as this one are often used to model the behavior of actors whose behavior changes over time.

Since we assumed that messages are the only way for actors to interact with the rest of the system, an actor's internal state is fully determined by the sequence of messages it has received. In this sense, each individual actor behaves deterministically. It is important to realize, however, that *groups* of actors can still behave non-deterministically. As a somewhat contrived example, if multiple users dial concurrently using the same phone, the resulting telephone number will be a random interleaving of the pressed digits. This form of non-determinism is a fundamental and unavoidable aspect of concurrent computation, regardless of design metaphor and programming language!

6.1 (A)Synchronous Channels

Implementing the message-passing model in multithreaded applications is fairly straightforward: actors correspond to threads, messages are are regular data structures, and channels are typically implemented using monitors that transfer messages from one thread to another. To mirror the encapsulation provided by actors, we adopt a strict model of data ownership: threads may only access their own private data and data contained in the messages they receive.

An obvious consequence is that messages must not contain references to shared data structures or data owned by other threads—only copies of data or references to immutable data. Of course, there are cases where data must be shared—either because copies are expensive in terms of memory consumption or CPU time, or because the data really *is* global to the whole program. In such cases it can be necessary to supplement message passing with monitors that manage access to those shared resources. But this should be the exception: strict data ownership is one of the core strengths of the message passing model.

Different forms of message passing can be implemented by using appropriate channels. We will use the following `Channel` interface to represent different kinds of channels.

```
interface Channel<Msg> {
  void send(Msg m) throws InterruptedException;
  Msg receive() throws InterruptedException;
}
```

All channels have the same two fundamental operations for sending and receiving messages, but different implementations can provide different semantics for these methods.

There are two predominant forms of message passing. *Asynchronous* (non-blocking) message passing corresponds to depositing sent messages in a mailbox and leaving them for the receiver to pick up at a later time. The sender continues immediately after posting the message; see Figure 20 (a). Asynchronous channels are implemented most easily using bounded or unbounded queues. The sender appends messages to the end of the queue and the receiver retrieves them from the front, so that messages are received in the same order in which they were sent. Not entirely by accident, the queue implementations presented so far already conform to the `Channel` interface.

Synchronous (blocking) message passing, on the other hand, postpones sending the message until the receiver is ready to accept it; this is also called *rendezvous* message passing because both parties must "meet" to exchange a message; see Figure 20 (b). Synchronous channels must implement the *rendezvous* operation by suspending the sender until the receiver picks up the current message. Obviously, only one message at a time can be transmitted through such a channel. After the message is transmitted the sender and the receiver continue independently—the sender only waits for the receiver to *receive* the message, not to finish *processing* it. A possible implementation is shown in Figure 21.

When do we use asynchronous and when synchronous message passing? The short answer is: in multithreaded applications, asynchronous message passing is usually the better choice. The main problem with synchronous messages is that blocking on every send operation easily leads to latency problems, which are particularly undesirable in interactive applications. In languages with

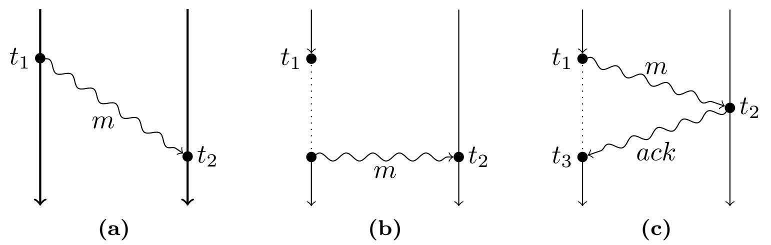


Figure 20: (a) Sending and receiving a message are independent operations in asynchronous message passing and may occur at different times t_1 and t_2. (b) In contrast, sending and receiving occur at the same time in synchronous message passing: the sender is paused until the receiver is ready to accept the message. (c) For most practical applications, a synchronous message can be replaced by a pair of asynchronous messages. The sender waits for an explicit acknowledgement message *ack* returned by the receiver.

```
class SynchronousChannel<Msg>
    implements Channel<Msg> {
  private Msg message;

  public synchronized
  void send(Msg m) throws InterruptedException {
    assert(message == null);
    message = m;
    notify();
    while (message == m)
      wait();
  }
  public synchronized
  Msg receive() throws InterruptedException {
    while (message == null)
      wait();
    Msg m = message;
    message = null;
    notify();
    return m;
  }
}
```

Figure 21: Implementation of a synchronous channel. If the sender is early, it waits for the receiver to pick up the message; similarly, if the receiver is early it waits for the sender.

built-in support for concurrency this is rarely a problem since it's easy to introduce new threads that hide these latencies, but in today's mainstream languages this is an arduous task. In cases where we need confirmation that a message was received, we can (asynchronously) send explicit acknowledgement messages as illustrated in Figure 20 (c).

6.2 Examples

After having discussed the abstract ideas behind message passing, we now turn to a few practical examples.

6.2.1 The Client-Server Model

The client-server model is well-known from network programming where a *server* is a central computer or application that provides certain services to all *clients* in the same network. The same idea works just as well in multithreaded applications: here, a server is a central thread that provides certain services to the rest of the application.

The following list illustrates a few possible uses for server threads:

- Load files from hard drive or network;
- Create and cache document previews in a file browser;
- Handle data exchange with an external program;
- Manage interaction with a database;
- Process a batch of background jobs like printing or rendering documents.

In each case the server thread either manages a shared resource or performs operations that can run independently from the rest of the application.

Consider an application such as a file manager or an FTP client that transfers files from one location to another. Instead of starting a new thread for every file being transferred, we can introduce a central server thread responsible for copying files. Centralizing this operation has many benefits: we can more easily estimate the total time required, avoid thrashing the hard drive or congesting the network, and manage the list of active transfers. Figure 22 shows the outline of a very simple implementation of such a server thread. Clients, i.e. threads that need files to be copied, can request a new copy operation by sending a message over the server's input channel. The server processes the requests in order and sends progress messages to its output channel.

Unbounded channels are usually a good default for implementing client-server models. Bounded channels can be used to prevent the client from getting too far

```
class FileCopyServer extends Thread {
  static class WorkMsg {
    String oldfile, newfile;
  }
  static class ProgressMsg {
    ...
  }

  private Channel<WorkMsg> inChan;
  private Channel<ProgressMsg> outChan;

  public FileCopyServer(Channel<WorkMsg> in,
      Channel<ProgressMsg> out) {
    this.inChan = in;
    this.outChan = out;
  }
  public void run() {
    while (true) {
      WorkMsg msg = inChan.recv();
      while (<<more data to copy>>) {
        <<do some work>>
        outChan.send(new ProgressMsg());
      }
    }
  }
}
```

Figure 22: A thread for copying files in the background. Two channels connect the thread with the rest of the program: `inChan` is used for receiving work request and `outChan` for sending progress information.

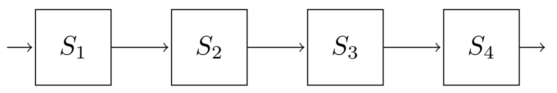


Figure 23: Illustration of a pipeline. The output of one stage is the input of the next stage in the chain.

ahead of the server or to limit the amount of memory consumed by the queue.

6.2.2 The Pipeline Model

As an alternative to the centralized client-server model, multiple threads can also be joined to form a pipeline in which each thread performs one step in a longer computation. The general structure of such a design is illustrated in Figure 23: data enters the pipeline at the left and is transformed by each step on its way to the right. In multithreaded programs, pipelines are easy to implement using message passing. Each stage in the pipeline is mapped to one thread, and consecutive stages are connected using channels.

To exploit this pipeline organization, we must be able to split up both the computation itself and the data being processed into multiple pieces. Consider for example the problem of printing a document in a word processing application. In general, parallelizing algorithms in a word processor will be difficult since the underlying data structures are quite complex. But for printing, we can assume that individual *pages* can be processed independently from each other, so we can send single pages down the following pipeline:

1. Prepare next page;
2. Perform color separation on page;
3. Rasterize page;
4. Send rasterized page to printer.

Since the individual steps of the computation are independent, we can run all of them in parallel: while one page is still undergoing color separation, the first stage can already prepare the next page.

Since pages pass through the pipeline sequentially, the time required to print a single page is the sum of the time it spends in each stage T_i

$$T = T_1 + T_2 + T_3 + T_4.$$

For documents that contain only a single page we obviously haven't gained anything. But if there are enough pages to keep all stages of the pipeline busy, the *average* time required to print a page is reduced to

$$T = \max\{T_1, T_2, T_3, T_4\},$$

We see that the performance of a pipeline is limited by its slowest stage. Pipelines therefore work most efficiently when every step takes approximately the same amount of time.

6.3 Discussion of Message Passing

Overall, communication through message passing has the following main advantages, especially in comparison to interaction based on shared memory or monitors.

- *Encapsulation.* The internal behavior and state of an actor is completely hidden to the outside world, unless we explicitly provide a messaging protocol to query this information.
- *Explicit Interaction.* In multithreaded programs, the thorniest bugs often result when threads interact by accident, for example by accessing a shared memory location without proper synchronization. In such cases, the program appears to misbehave spontaneously. Problems like this are almost nonexistent in programs based on message passing since all interaction between actors requires the explicit exchange of messages.
- *Synchronization.* A message can only be received *after* it is sent. This observation isn't as trivial as it sounds: especially subtle, rare race conditions in

multithreaded programs are caused by the lack of a causal relationship between the threads, which is implicitly provided by message passing.

- *Weak coupling.* Message passing introduces only a weak form of coupling between actors. As long as we know what kinds of messages an actor can send and receive at a particular time, we don't need to know anything about its internal operation. For example, the telephone doesn't know how the user or the carrier makes its decisions—it only needs to correctly handle the messages they send. Compare this to interaction via shared memory: unless every part of the program accessing a shared variable knows in detail how that variable is used in other parts of the program, there is a risk of data races or locking errors.

Message passing isn't the final answer to all concurrency problems, however. As we mentioned, nondeterminism is still an issue, as are race conditions and deadlocks. Furthermore, message passing can be more cumbersome than other communication mechanisms, especially in programming languages without built-in syntactic support for sending, receiving, and parsing messages (which, unfortunately, still includes all current mainstream programming languages).

Nevertheless it is a very general and flexible metaphor that lends itself well to designing and reasoning about concurrent programs. It also prevents many of the common problems that plague multithreaded programs.

7 Conclusion

The main distinction between threads and other models of concurrency is that threads automatically share all process resources, most importantly the memory space. This is the biggest advantage of threads, but also their most severe drawback. On the one hand, being able to transparently share memory between threads is very efficient and allows threads to easily interact with existing code. On the other hand, concurrent access to this memory is highly non-deterministic on all modern computer architectures and conflicts with proven techniques from sequential programming. The fact that this shared memory is unsafe by default and must be tamed with suitable synchronization operations is clearly the main reason for the perceived (and actual) difficulty of multithreaded programming.

The primary reason for using threads is performance: threads can increase the *throughput* of a program by working on multiple subproblems in parallel, or its *responsiveness* by offloading time-consuming activities to separate threads without blocking the rest of the program. Performance is a noble goal, but adding more threads makes programs more complex and often harder to understand, test, and debug. It is therefore rarely wrong to err on the side of simplicity and keep the number of threads as low as possible. This isn't meant as a warning against threads per se—it's a warning against a particularly dangerous form of premature optimization.

But even though threads require more diligence than typical sequential programming techniques, using concurrency correctly can also lead to simple, elegant solutions that are harder to achieve in purely sequential programs. Thinking about programs as collections of independent, concurrent units may be unfamiliar to programmers new to concurrency, but it's often more natural than the idea of an endless, linear stream of instructions from sequential programming. The benefits of concurrency as a programming metaphor are often overlooked.

The thread model is often (rightly) criticized because even trivial operations such as as reading or writing a variable in shared memory can introduce unintended interactions between threads. Dealing with the inherent difficulty of threads requires a solid understanding of the basic techniques such as condition variables, monitors, and message passing. But more importantly, it requires programmers to

- Devise clear models of data and resource ownership;
- Decompose a program into well-defined components that run concurrently; and
- Conceive simple and safe protocols for communication between these components.

Solving three problems is not merely a matter of applying a fixed repertoire of programming techniques but a design activity that, ultimately, can only be learned through experience.

My hope is that this article served you as a starting point for further excursions into this challenging and fascinating subject.

8 Bibliographic Notes

Threads. Concurrency has become a hot topic in recent years; the term "concurrency revolution" was popularized by Sutter [34] and Sutter and Larus [35]. They argue that "the free lunch is over" and a major shift from sequential to concurrent programming will be necessary to exploit future hardware advances. For a slightly different analysis ("lunch might not be exactly free, it *is* practically all-you-can- eat"), refer to the article by Cantrill and Bonwick [10].

Threading APIs for different platforms and languages differ in many subtle and not-so-subtle ways. I only discussed the fundamentals of Java threads; for a much more exhaustive treatment turn to the books by Goetz et al. [14] and Lea [26]. Butenhof [9] is the classic reference for POSIX threads, the threading API used on Unix-based systems, and Duffy

[13] treats Windows and .NET concurrency in detail. Introductions to multithreaded programming can be found in Birrell's article [6] (which, incidentally, is a good complement to this article) and the book by Andrews [2].

The idea of using invariants for reasoning about programs goes back to Hoare and his work on program verification [20].

Mutexes. Deadlocks in connection with mutexes are a special case of the general deadlock problem, which crops up in many situations in which limited resources are dynamically allocated to multiple concurrent entities. The deadlock problem was first discussed by Coffman et al. [11]; their paper listed necessary preconditions for deadlocks (circular dependency must be supplemented by three other conditions in the general case) and ways for detecting and preventing deadlocks. Deadlocks are covered in every textbook on operating systems [32, 36].

The book by Goetz et al. [14] devotes several sections on contention in multithreaded programs and ways to deal with such problems. Special processor instructions called *atomic operations* can sometimes be used to update individual shared variables without locking. For some data structures, algorithms exist that either work correctly without locking (*lockless algorithms*) or that at least guarantee that locks are only held briefly (*wait-free algorithms*). Designing such data structures and algorithms and implementing them correctly is very difficult, even for experts, but readers interested in this fascinating topic will enjoy the book by Herlihy and Shavit [18].

No discussion of recursive mutexes would be complete without mentioning Butenhof's classic newsgroup posting [8].

Monitors. The concept of monitors was developed in the early 1970s by Brinch Hansen [7] and Hoare [21]; condition variables were also introduced in this context. Most modern variants of monitors and condition variables can be traced back to the Mesa programming language [29]. Lampson and Redell [25] describe the design considerations that went into Mesa monitors; their text is still an insightful read because it explains why modern threading primitives behave the way they do.

I only briefly touched upon the problems associated with calling external code from monitors. *Nested monitor calls* are a special case where one monitor calls methods in another monitor. This question has received a lot of attention; the article by Andrews and Schneider [3] contains a discussion and a detailed bibliography.

Our discussion of the Observer pattern was inspired by Lee's classic critique of threads [27]. Lee correctly observes that "this very simple and commonly used design pattern" is hard to implement correctly in a multithreaded setting, but his conclusion isn't that side effects or dynamic dispatch are hard, but that threads per se are unmanageable. I would encourage you to read his article and form your own opinion.

Message passing. This article used message passing only for communication between threads, but the abstract idea is extremely general. The "Message Passing Interface" (MPI), for example, is a standard API for constructing distributed parallel applications [33]. Many mathematical models of concurrency are based on message passing, such as the *actor model* by Hewitt et al. [19], Milner's *π calculus* [28], and Hoare's *concurrent sequential processes* [22]. I especially recommend Hoare's book which is an insightful treatment of the theory and practice of concurrent computation using synchronous message passing.

As I already mentioned, message passing can be cumbersome in languages that do not provide syntactic support for operations such as sending, receiving, or replying to messages. Many languages that were specifically designed for concurrent computing have strong support for message passing, among them occam [31], Erlang [5], and more recently Scala [16] and Go [15]. Armstrong's book on *Programming Erlang* [4] contains several real-world examples of concurrent programming using message passing. The article by Karmani et al. [23] reviews several Java and JVM frameworks for message passing.

Further reading. I briefly noted on the complicated behavior of shared memory in current computer architectures. For example, memory operations are performed by one processor can appear to occur in a different order when observed from another processor. Such *weak memory models* are inconvenient for programmers but have been necessary so far for performance reasons. General overviews can be found in Adve and Gharachorloo's tutorial on shared memory consistency models [1] and in textbooks on computer architecture [17]. The actual memory models are extremely hardware dependent and are typically hidden inside the technical manuals of each processor model.

Another huge area I have passed over is the problem of designing *parallel algorithms* that perform computations more efficiently by distributing them over multiple processors; a good introduction to this subject is Roosta's book [30]. The *Sourcebook of Parallel Computing* [12] contains a large collection of articles on parallel computing with an emphasis on scientific problems. Recently, the emergence of massively parallel graphics processors (GPUs) has led to the development of specialized languages targeted at parallel computation, such as CUDA and OpenCL; the book by Kirk and Hwu [24] is a nice introduction to this topic.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12): 66–76, Dec. 1996.

[2] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[3] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, Mar. 1983.

[4] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG.* Prentice Hall, second edition, Jan. 1996.

[6] A. D. Birrell. An introduction to programming with C# threads. Technical Report TR-2005-68, Microsoft Corporation, 2005.

[7] P. Brinch Hansen. *Operating System Principles.* Prentice-Hall, 1973.

[8] D. Butenhof. Re: recursive mutexes. `comp.programming.threads`, May 2005. `http://groups.google.com/group/comp.programming.threads/msg/d835f2f6ef8aed99`.

[9] D. R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[10] B. Cantrill and J. Bonwick. Real-world concurrency. *ACM Queue*, 6(5):16–25, Sept. 2008.

[11] E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.

[12] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors. *The Sourcebook of Parallel Computing.* Morgan Kaufmann, 2003.

[13] J. Duffy. *Concurrent Programming on Windows.* Addison-Wesley, 2008.

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

[15] *The Go Programming Language Specification.* Google, Inc., Mar. 2011. `http://golang.org/doc/go_spec.html`.

[16] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, Feb. 2009.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, fourth edition, 2006.

[18] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[19] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[20] C. A. R. Hoare. Proof of a program: FIND. *Commun. ACM*, 14:39–45, Jan. 1971.

[21] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.

[22] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985. Electronic version at `http://www.usingcsp.com/`.

[23] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, 2009.

[24] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 2010.

[25] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, Feb. 1980.

[26] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, second edition, Nov. 1999.

[27] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[28] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[29] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1979.

[30] S. H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation.* Springer, 1999.

[31] SGS-THOMSON Microelectronics Limited. *Occam 2.1 Reference Manual*, 1995. `http://www.wotug.org/occam/documentation/oc21refman.pdf`.

[32] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts.* Wiley, eighth edition, 2008.

[33] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*, volume 1. MIT Press, second edition, Sept. 1998.

[34] H. Sutter. The free lunch is over. *Dr. Dobb's Journal*, 30(3), Mar. 2005. URL `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[35] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, Sept. 2005.

[36] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Upper Saddle River, NJ, USA, third edition, 2008.