Extensions to Polyphonic C#

Joel Barciauskas
Professor Edward Sciore, Advisor
Boston College
Computer Science Department

Abstract

Synchronization mechanisms in concurrent programming have progressed very slowly over the years. Semaphores and monitors are still the most widely used tools for solving these problems. We discuss in this paper a new type of synchronization mechanism called "chords." Previous papers have discussed a few areas in which chords are well-suited to replace semaphores and monitors, including asynchronous method execution and return value retrieval and the reader-writer problem. We will attempt to solve a number of other traditional synchronization problems using chords, and make suggestions as to possible extensions to the rules governing chords that would make them even more useful.

Introduction: What Are "Chords"?

In order to understand this paper, one must have a reasonable understanding of how chords function. We will be using as a reference the language Cω, or "Comega." Comega is a language developed by researchers at Microsoft Research that presents a concrete implementation of, among other experimental language features, chords. The Comega implementation of chords introduces two major pieces of syntax to a language. The first is the creation of an "async" keyword. The async keyword is a new return type for methods and essentially ensures that the current thread will not block waiting for the method to return anything. In effect, an async method returns instantaneously. An example of a class with a simple async method looks like this:

```
public class ClassUsingAsync
{
      public async m1();

      when m1()
      {
```

```
                //method stuff
        }
}
```

The other new keyword here, "when," means that when m1() is called, the code in the body of the when clause is executed asynchronously from the calling thread, usually in a new thread or a thread from a shared pool. Asynchronous method code may also be declared in-line, but I use the slightly more complicated version for the sake of consistency when we explore the next syntax addition of chords.

The second new syntax that is enabled in chords is the ability to "join" methods with each other. Two or more methods joined with each other are what we refer to as a "chord." All method calls are kept in a queue, and the body of a chord is executed when all the joined methods in a chord are present in the queue. The syntax for joining a chord is simply listing all the methods in the chord separated by an ampersand ("&"). Going back to the previous example,

```
public class ClassUsingAsyncVersion2
{
        public async m1();
        public async m2();

        when m1() & m2()
        {
                //method stuff
        }
}
```

As is the case here, chords may be composed entirely of asynchronous methods. In this case, each call to m1() or m2() completes instantaneously as normal and the calling thread continues immediately following the call. Each object has an underlying queue in which it stores all calls to methods that have not been part of a completed chord. When all the methods of a chord are found in this queue, and the chord is composed of only asynchronous methods as in the example above, the chord body is not executed in any of the calling threads. After all, how would we decide which one to interrupt? Instead, a new thread is created, or an available thread is drawn from a thread pool, depending on the implementation.

It is also possible for a chord to return a value if one of the methods in the chord is synchronous. That is, only one method may have a return type other than "async." When a thread makes a call to the synchronous method in a chord, it blocks until the chord is completed, at which point the body of the chord is executed in the same thread of the synchronous method. This is important to note: a chord that contains a synchronous method does not result in the creation of a new thread. Building on the previous example, here is a class with a synchronous chord:

```
public class ClassUsingAsyncVersion3
{
      public async m1();
      public async m2();

      when m1() & m2()
      {
            //method stuff
      }

      public int m3() & m1()
      {
            return 0;
      }
}
```

If a thread makes a call to m3() without any previous calls to m1() by any other threads, it will block and wait to be notified. It will be notified when another thread makes a call to m1() and the chord is complete, and the thread that called m3() will then execute the code in the body of the chord, in this case simply returning "0."

There is one final important rule to consider about the behavior of chords. We must decide what to do when there is more than one possible chord that could be executed in the queue, but the two chords share a method. For example, in the third version of the ClassWithAsync, as seen above, m1() is part of two different chords. In the program above, it might be the case that there are calls to m2() and m3() already made, and a call to m1() completes two chords at the same time. We must decide which chord executes and which must wait for another call to the shared method. In the original Polyphonic C# paper, it is stated that generally this decision will be non-deterministic, i.e. one of them will be chosen at random, and you should not write your program to rely on a specific behavior. A major subject of this paper is an alternative to this non-determinism that might make chords more useful.

Further Examples

Some kinds of problems that can be solved very elegantly using chords. For instance, a single-cell buffer might be implemented using chords like this:

```
public class OnePlaceBuffer {
   private async empty();
   private async contains(string s);

   public OnePlaceBuffer() {
      empty();
   }

   public void Put(string s) & empty() {
      contains(s);
      return;
   }

   public string Get() & contains(string s) {
      empty();
      return s;
   }
}

public class Demo {
  static OnePlaceBuffer b = new OnePlaceBuffer();

  static async Producer() {
    for(i=0; i<10; i++) {
      b.Put(i.ToString());
      Console.WriteLine("Produced!{0}", i);
```

```
        }
      }

      static async Consumer(){
        while(true){
          string s = b.Get();
          Console.WriteLine("Consumed?{0}",s);
        }
      }

      public static void Main() {
        Producer();
        Consumer();
      }
    }
```

This is a very simple example in which a single string is held by the object on a call to Put(), and returned upon a call to Get(). A call to Get() without any item in the buffer blocks until a call to Put() is made, and a call to Put() against a full buffer blocks until a call to Get() is made. The object notifies the queue that a Put() operation has completed and that a call to Get() may be completed by posting the contains() message on the queue. The contains() message is used not only to notify a previous Get() call, but also as a way of storing the string parameter. The queue may be used to save state this way and provide synchronized access to data without use of a shared variable.

Another common task that can be completed easily with chords is an asynchronous callback situation. The idea is that a call is made to an asynchronous method, and then later a method is called to block and wait for the retrieval of the result of the computation. This is very useful in situations where a complex computation might be made, or a long read or write operation is being performed. As one might imagine given the presence of the "async" keyword in chords, this is very easy:

```
public class AddAsyncClass {
      public async asyncAdd(int i, int j);
      public async addComplete(int n);

      when asyncAdd(int i, int j)
      {
            int n = i+j;
            addComplete(n);
      }

      public int getVal() & addComplete(int n)
      {
            return n;
      }
}

public class Demo {

  public static void Main() {
      AddAsyncClass add = new AddAsyncClass();
      add.asyncAdd(2, 2);

      //some code you want executed

      int result = add.getVal();
  }
}
```

The second message, addComplete, saves the result of the computation as a parameter within message on the queue, much as the contains() method does in the previous example. We then use getVal() to retrieve that message and its stored parameter later.

Semaphores in Chords

Another point that should also be made is that any solution that can be achieved using semaphores can be imitated using chords. This is proven by the fact that we can write our own chord-semaphore class like this,

```
public class Semaphore {
      public async Signal();
      public void Wait() & Signal() {}
}

public class Demo {
  static async task(int n, Semaphore s);

  public static void Main() {
    Semaphore s = new Semaphore();

    task(1,s);
    task(2,s);

    for(int i=0; i<10; i++) {
      Console.WriteLine("Main");
```

```
        Thread.Sleep(10);
      }
      Console.WriteLine("Waiting for tasks to finish");
      s.Wait(); // wait for one to finish
      s.Wait(); // wait for another to finish
      Console.WriteLine("both tasks finished");
      Console.ReadLine();
    }

    when task(int n, Semaphore s) {
       for(int i=0; i<10; i++) {
        Console.WriteLine("Task {0}",n);
        Thread.Sleep(n*10);
       }
       s.Signal(); // say we've finished
    }

}
```

We see the incredible simplicity of the implementation of a Semaphore class here, as well as a simple program that makes use of it. The client code makes a couple of asynchronous calls, and waits for the threads the tasks are spawned in to signal their completion. We are, however, much more interested in novel solutions that differ significantly from the semaphore-based solutions that have already been established.

The "async" Keyword and Thread Creation

In C#, starting a new thread is a fairly complicated affair. It requires writing a delegate that conforms to the ThreadStart delegate format. This format is,

```
public delegate void ThreadStart();
```

The class c we want to control the new thread must therefore have a method that takes no parameters and has a void return type. It might look something like this,

```
public class ThreadController
{
      private int num;

      public ThreadController(int x)
      {
            num = x;
      }
      public void threadStart() { //do something with 'num' }
}
```

We create a new Thread instance by declaring,

```
ThreadController tc = new ThreadController(3);
Thread t = new Thread(new ThreadStart(tc.threadStart));
```

Finally, we start the thread by declaring,

```
t.Start();
```

Unfortunately, there is no way to pass a value directly into the thread delegate, as the method that is the basis of the delegate cannot take any parameters. If we want to achieve this, we must pass in the data somehow before we start the thread. This is most commonly achieved through passing data into the constructor, as shown in this example assigning the value passed into the constructor to the class' private 'num' variable.

By contrast, the async keyword allows us to abstract away this creation of threads. We are still constrained by the lack of a return type, but we can arbitrarily define parameters to a method that we want to cause a new thread to spawn. Also, we can define multiple methods and chords that cause new threads to be created, rather than being required to create a new class with a new ThreadStart delegate every time we want to start a thread with a different set of parameters. Using the async keyword, we might define a class similar to the one above like this,

```
public class ThreadController
{
      public async threadStart(int num)
      {
            //executes in a new thread
            //do something with num
      }
}

ThreadController tc = new ThreadController();
tc.threadStart(3);
```

Instead of requiring a constructor and a private variable to hold the value passed to the constructor, we can define parameters for the async method and pass values and objects to the new thread that way.

Furthermore, in addition to being the ability to eliminate the "new Thread(new ThreadStart())" ugliness, there is another interesting facet of this code. In the above definition of ThreadController, the class definition requires that a new thread is created on every call to threadStart(). In the previous definition, a call to threadStart() might be as a result of a new thread being created with threadStart as the ThreadStart delegate, or it might just be called arbitrarily by a client. If there is some reason we want to guarantee that a new thread is created on every call to a particular method, we must use the async keyword solution. The async solution can also be modified so that the client code has the option of using the functionality of the async method without the expense of spawning a new thread, like this:

```
public class ThreadController
{
      public async threadStart() { nonThreadStart(); }
      public void nonThreadStart() { //original async code here }
}
```

We can put the same functionality in two different method headers when we want to give a client of a ThreadController instance the ability to decide if it wants to spawn a new thread. We can achieve this by making an asynchronous wrapper for another synchronous method. The names used in this example are a little ambiguous – in reality, the nonThreadStart() code is being executed in a new thread when it is called from threadStart(). We can easily change this code back to guaranteeing a new thread is created by making nonThreadStart a private rather than public method. The async keyword gives us the flexibility of ensuring that a new thread on every call or leaving it up to the client, an ability we lack in normal C# and that we may find desirable in solving certain concurrency problems.

A Simple Bouncing Ball and Flow Control

When we look at a very simple use of the async keyword and chords, we already begin to see some interesting things. We wrote a simple bouncing ball program, where a ball is drawn in a window and moves and bounces off the edges of the window. The ball is controlled by a toggle button that either stops the ball or starts the ball back up after the user has stopped the ball. The full code for both the C# and Comega implementations of this program can be found in the appendicies.

This functionality requires that the ball be operating in a different thread than the toggle button, in order to maintain the button's responsiveness. In the C# implementation of the bouncing ball program, this means that we get to use the .NET Thread library,

```
Controller ball = new Controller(g);
t = new Thread(new ThreadStart(ball.start));
t.Start();
```

Now, to control the ball's movement, we are going to use a simple while loop within the BallThread object's start() method,

```
while(go)
{
      ball.move();
      ball.paint(formGraphics);
      Thread.Sleep(5);
}
```

The move() method updates the position of the ball, and the paint() method erases the previous image of the ball and draws the ball at the new location. Lastly, we make a call to Thread.Sleep in order to slow the animation to a reasonable speed.

In order to stop the thread, we add a stop() method to the BallThread class with access to the private boolean variable "go," in which we simply set the variable to false. The while loop will fail on the next test and the thread will exit. It is a very simple and straightforward multi-threaded program.

We can modify this program to use the async keyword and eliminate the Thread library code very easily. We simply redefine the start method as async rather than void. We can then eliminate the two lines,

```
t = new Thread(new ThreadStart(ball.start));
t.Start();
```

And replace them with a direct call,

```
ball.start();
```

These sections are bolded and italicized in Appendix A, which contains the code for the implementation of the BouncingBallForm class. The chord solution works, and is marginally more sensical than the regular C# version, but is not really a significant use of the features presented with chords in Comega. Instead of using a variable to maintain state and control the thread, we want to try to take advantage of the chord mechanism to achieve the same functionality.

In order to do this, we will need to add a few more private methods to the BallThread class. First of all, we are going to eliminate the while loop and the boolean "go" variable. Instead, we will use three chords to control the movement of the bouncing ball. The chord definitions will be,

```
private void move() & checkStop()
public async stop() & checkStop()
private void move() & killMove()
```

The first chord will perform the normal movement of the ball, updating its position and calling the paint() method to redraw the ball in its new position, the second chord stops the looping of the first chord, and the last chord is used to clear the message queue of the spare move() message that remains after the ball's movement has been stopped. One might also wonder what happened to the start() method amongst all these chords. In order to eliminate the state variable, and by extension, the while loop, from the program, we instead use recursive method calls. However, we do not want to recursively call a method defined as async, because that would incur the overhead of creating a new thread on every single call. Instead, we want to make a single async call that then starts a normal, single-threaded recursive loop. The move() method is the recursive call we will make from start(), and we will also post a checkStop() message, which makes sense in that it is possible to have the toggle button clicked twice fast enough that start() and stop() are called in succession and we never get around to actually moving the ball.

Let's take a look inside the move() & checkStop() chord, and explain exactly what the purpose of the checkStop() message is.

```
private void move() & checkStop()
{
      checkStop();
      ball.move();
      ball.paint();
      Thread.Sleep(5);
      move();
}
```

The middle three method calls are familiar; they are lifted directly from the C# version of our program. The final move() message is the recursive loop we just discussed. So, what is the checkStop() message? Well as was alluded to earlier, it essentially serves to check and see if a stop() message has been posted. When the ball is moving, the event handler associated with the toggle button is written to post an asynchronous stop() message to the BallThread object (or, all instances of BallThread, in the case of multiple simultaneous bouncing balls). We have joined checkStop() with both move() and stop(). When a stop message is posted, it will either match and steal the checkStop message in the queue, or will sit there until the next time a checkStop message is posted. Notice there should never be more than one checkStop message in the queue at a time, and in the time between the removal of the checkStop message from the queue upon the matching of a move with a checkStop message, and the placing of a new checkStop message on the queue, there may be not be a checkStop message in the queue. This is not a problem, as the call to stop() is asynchronous, and if it arrives before the new checkStop message is posted, it will sit in the queue until the checkStop message is in fact posted.

At this point, the checkStop message that was previously used to complete the move and checkStop chord will instead complete the stop and checkStop chord. When the thread arrives at its next call to move(), it will find that it has no checkStop message to match with, and will block. In the meantime, the body of the stop and checkStop chord posts a kill() message, which will in turn match with the one chord we have not yet discussed, the move and kill chord. The body of this chord is empty, and its function is essentially to provide a way to escape the recursive move() loop and allow the ball movement thread to safely exit.

This sounds like the code should be airtight, but there's a subtlety about the way chords are matched and threads are notified that is hidden here, but becomes apparent when the order of execution of the move and checkStop chord body is changed around so that checkStop is executed after the call to Thread.Sleep() rather than before. The problem is that the way chords are implemented in Comega, a thread may be notified that a match has been made, but no guarantee is made that that match will be saved until the thread is executed. The original Polyphonic C# paper gives the following illustration of the problem we encounter,

```
class Foo {
        void m1() & async s() & async t() {. . . }
        void m2() & async s() {. . . }
        void m3() & async t() {. . . }
}
```

and the following global execution trace, with four threads running in parallel:

Thread 1. calls *m1*() and blocks.

Thread 2. calls *m2*() and blocks.

Thread 0. calls *t*() then *s*(), awaking Thread 1.

Thread 3. calls *m3*() and succeeds, consuming *t*().

Thread 1. retries *m1*() and blocks again. (Benton, et al. 22)

The problem here is that while Thread 1, which, in conjunction with Thread 0, completes the first chord and Thread 1 is notified, but it does not execute right away. The thread scheduler instead goes ahead and allows Thread 3 to call m3(). Thread 1 was notified, but it is up to the notified thread to consume the messages from the queue, and since it never got a chance to do that, Thread 3 was able to preempt Thread 1's match and steal the t() message. Once Thread 1 was finally able to execute, it looked back at the queue and saw it no longer was able to execute, and was forced to block again.

The bouncing ball program basically benefits from a timing trick that prevents this sort of stealing from happening, or really just makes it extremely unlikely. When checkStop() is called before Thread.Sleep(), the stop message will almost always enter during the execution of ball.move(), paint(), or during the Sleep(), when the move()

thread is blocked. Because the ball movement thread is very likely to be blocked when stop and checkStop match and the button's thread will be free to execute, the button thread will have enough time to consume the checkStop message before the ball movement thread is woken up and has a chance to post its move message. However, if the checkStop message is posted after the call to Thread.Sleep(), the stop message will likely be on the queue before either checkStop or move are put back on the queue. Then, checkStop and move will be placed on the queue almost simultaneously. When the checkStop message is posted, the button thread will be notified, but it may not have a chance to execute as the move message is posted immediately following the checkStop message, and the ball movement thread may be notified and executed before the button thread can consume the checkStop message. Let's look at the possible orders of execution of these threads. Thread 0 will represent the button thread calling start() and stop(), Thread 1 will represent the thread spawned by the call to the async start() method, and Thread 2 is the thread created by the asynchronous call to stop() joined with the by Thread 0.

1)		Thread 0 calls start()
2)	Thread 1 is spawned	
3)	Thread 1 calls checkStop()	
4)	Thread 1 calls move()	
5)	Thread 1 consumes move() and checkStop()	
6)	Thread 1 calls checkStop()	
7)	Thread 1 calls ball.move()	
8)	Thread 1 calls ball.paint()	
9)	Thread 1 calls Thread.Sleep(5) and blocks	
10)		Thread 0 makes async call to stop()
11)		Thread 0 consumes stop() and checkStop()
12)		Thread 0 spawns Thread 2 to execute the body of stop() & checkStop()
13)		Thread 2 executes kill()

14)	Thread 1 wakes up and calls move()	
	Thread 1 consumes move() and kill()	
	Thread 1 exits	

1)		Thread 0 calls start()
2)	Thread 1 is spawned	
3)	Thread 1 calls checkStop()	
4)	Thread 1 calls move()	
5)	Thread 1 consumes move() and checkStop()	
	Thread 1 calls ball.move()	
6)	Thread 1 calls ball.paint()	
7)	Thread 1 calls Thread.Sleep(5) and blocks	
8)		Thread 0 calls stop()
9)		Thread 0 is notified and spawns
	Thread 1 wakes up and calls checkStop()	Thread 2
		to execute the body of
10)		stop() & checkStop()
	Thread 1 calls move()	
11)	Thread 1 consumes move() and checkStop()	
		Thread 2 fails to consume stop() &
12)		checkStop()
13)		
14)	Thread 1 loops back and calls ball.move()	

Now, there is no reason why the first case could not fail with a very poorly timed stop() message that enters just after Thread 1 wakes up, but it is much more unlikely, whereas in the second case almost virtually guarantees a race between Thread 1 and Thread 2 to consume the checkStop message. The solution of placing the checkStop message before Thread.Sleep works in a practical sense, but does not give us the guarantee of correctness we might require in our programs.

The Unisex Bathroom Problem

The above example is not a great case study. It is very easy to argue that the use of chords is inappropriate, and we can get the correct functionality much more easily by scrapping the chords and using the original while() loop code. The use of the async keyword may be preferable to some, but so far we are talking about a net difference of one or two lines of code. Of course, lines of code do not tell us the whole story either. The code for creating a thread in C# is very dense and unintuitive, especially compared to the use of the async keyword. Let us examine a more meaningful example, and consider options for making more complex solutions more intuitive as well.

The Unisex Bathroom problem is a common problem, found in almost all introductory concurrent programming textbooks. The rules are relatively simple. There is a single bathroom, with a finite number of stalls. It may be used by males or females, but not by both at the same time. If the bathroom is empty, either sex may enter, but once it is occupied, only members of the same sex may enter, and only then while there are open stalls. If the bathroom is full, or is occupied by the opposite sex, the person attempting to enter must wait in line. For the purposes of our first solution, we will not really keep an orderly, first in, first out line, but more like a waiting room. This is inefficient and unfair, but presents an interesting problem nevertheless.

The chorded solution to the unfair bathroom problem is interesting in its intuitiveness and readability. We will define four chords to handle the major cases.

```
enterFemale(Person p) & open() & females(){ … }
enterFemale(Person p) & empty() & open()  { … }
enterMale(Person p) & open() & males()    { … }
enterMale(Person p) & empty() & open()    { … }
```

Each of these methods is defined as being asynchronous using the async keyword. We can literally read off the chords to understand the cases they handle. When we have a female attempting to enter the bathroom, and there is an open stall, and the bathroom is occupied by females, allow the female to enter. When we have a female attempting to enter the bathroom, and there is an open stall, and the bathroom is empty, allow the female to enter. Repeat these cases for the males. If we compare this to the semaphore-based solution provided by Downey,

```
femaleLouie.lock (empty)
      femaleMultiplex.wait ()
            bathroom code here
      femaleMultiplex.signal ()
femaleLouie.unlock (empty)
```

The solution is not longer or more complicated, but it requires a deeper consideration of the possible cases to understand its correctness. The chorded solution simply defines the cases and handles them as they arise.

The code inside of these chords requires a bit of explanation. In particular, the use of the males() and females() messages is important. Immediately after each call to one of the enterXXX chords is made, a males() or females() message is placed on the queue. This works in an affirmative way, not so much working to prevent the opposite sex from entering as much as it only allows the current sex to enter. This affirmative quality is important, however, because it leads to a small loophole that can cause some very big problems. The problem arises because there is no guarantee that the males() or females() message will be cleared from the queue in a timely fashion. The body of the exit method called by each person exiting the bathroom looks like this:

```
public void exit(Person.Gender gender)
{
      open();
      countWait();
      count--;
      if(count == 0)
      {
            empty();
            if(gender == Person.Gender.Female)
                  clearFemales();
            else
                  clearMales();
            Console.WriteLine("Bathroom is empty");
```

```
        }
        countSignal();
}
```

Eventually, the exit thread that called empty() will get to its call of clearMales() or clearFemales() to clear the old message. However, it may not happen for a long time, if the thread scheduler decides to make that chord wait immediately after it consumes its messages off the queue. It is also possible that the clearGender() & gender() chord will be continually pre-empted by the corresponding enterGender() & open() & gender() chord. We showed back in the Bouncing Ball example that there is no guarantee about how chords will act when there is a possibility of multiple chord completions.

Let's walk through an example of how we may end up in an undesired state with the current attempt at a solution to the Unisex Bathroom Problem. Say that the bathroom is currently occupied by males, and a thread has just consumed enterMale(), males(), and open() messages, but then the scheduler forces the thread to block. If this occurs, and everyone currently in the bathroom leaves, a condition will occur where the empty() message will be posted and a female may be allowed to enter the bathroom and post her own affirmative females() message. Say this happens, and then finally the first thread that we had intended to put up its males() message right away is activated. Now we have both males() and females() messages up, and mayhem in the bathroom. Eventually, the thread executing the exit() method will post the clearMales() message, but, as with the stop() & checkStop() issue above, there is no guarantee that the clearMales() & males() chord will execute immediately, leaving open the possibility of continued mingling of both genders in the bathroom.

In order to solve this problem, we need to be able to assure two things. First of all, we must assure that the posting of the empty() message and the posting of the clearMales() message are performed atomically. We can accomplish this very simply, by obtaining a lock on the object. The new exit() code looks like this,

```
public void exit(Person.Gender gender)
{
      open();
      lock(this)
      {
            count--;
            if(count == 0)
            {
                  empty();
                  if(gender == Person.Gender.Female)
                        clearFemales();
                  else
                        clearMales();
                  Console.WriteLine("Bathroom is empty");
            }
      }
}
```

Secondly, and much more difficultly, we want to guarantee that when clearMales() is posted, it is matched with the last remaining males() message on the queue and that chord is consumed, at the exclusion of any other enterMales() & open() & males() chord that may also be completed in the queue. This is a much more complicated matter that requires that we re-write the way the queuing mechanism works in the Comega implementation of chords. We no longer want the non-deterministic aspect of chord resolution that made the implementation relatively simple. Instead, we want to be able to dictate priority levels and possibly implement any number of different conflict resolution schemes. We might want to assign weights to various methods, where the chord with the highest weight always executes ahead of all lower-weighted chords. We might want a simpler, first-in first-out idea, where the chord with the method that has been in the queue the longest is always matched and consumed first. The point is that in order to get the most out of chords, we want greater control over the order of consumption than the current Comega implementation allows.

In the particular case, first-in first-out will not cut it. There may be a number of enterGender() messages that are quite old, and we do not care about them. We want to give the clearMales() and clearFemales() chords a priority level greater than enterGender() & open() & gender() chords and guarantee that clearMales() and clearFemales() chords are always consumed first. We may achieve this through assigning methods weights, or simply assigning each chord a priority value as a whole. In this case, it would suffice to give the chords clearMales() & males() and clearFemales() and females() a priority level of one, and all other chords a priority level of zero. This, coupled with the object lock, would guarantee the correctness of this solution to the Unisex Bathroom Problem.

Bouncing Ball Redux

Before we move on, it would also be helpful to note that the Bouncing Ball problem, which we described in detail, would greatly benefit from some type of priority enhancement of the existing chord implementation as well. As was previously stated, the problem with the stop() & checkStop() chord was that that chord was never guaranteed to execute ahead of move() & checkStop(). A priority value idea that gave the stop() & checkStop() chord a higher priority would work here, but even more simply, a first-in first-out guarantee would work as well. In the worst case, stop() would enter immediately after move(), but it would be guaranteed to be the oldest chord in the queue on the next call to checkStop(). In any case, either solution would work, but if we are able to implement either one of them, the priority value solution seems to be more flexible. Let us go back to the Unisex Bathroom, however.

An Attempt at Fairness

We should not be satisified with the solution we've constructed for the Unisex Bathroom Problem. It is a very unfair solution that can quickly lead to starvation for one gender. There is no mechanism to stop one gender from dominating the bathroom, continually replacing each other without allowing the opposite gender to get its foot in the door, so to speak. One very simple idea for preventing this type of behavior is to, as soon as a member of the sex opposite that of the current bathroom occupant enters the line (or "waiting room," a more apt analogy), everyone is prevented from entering the bathroom and the opposite sex is given the chance to take control. In a balanced-gender scenario, this may lead to a lot of see-sawing; it is not particularly efficient. It is a relatively simple improvement, however, so let us take a look anyway.

To achieve the desired behavior, we only need to add three chords:

```
enterFemale(Person p) & males() { //code }
enterMale(Person p) & females() { //code }
private void waitEmpty() & empty() { //no code }
```

The purpose of the first two chords is to remove the affirmative sign from the door, without replacing it with an empty() sign. This has the effect of not allowing any of the other four entry chords to execute any more. Once that is done, we want to wait for the bathroom to empty out, then post the new gender's message on the queue, allowing threads/people of that gender to enter. The bodies of the first two chords are essentially identical, and consist of a call to the blocking method waitEmpty(), followed by posting their own gender() message and then repeating the call to enterGender(). The body of the waitEmpty() & empty() chord is blank, as it serves only to remove the empty() message from the queue.

In order for the first two chords to properly “steal” the gender() messages off the queue, we want to go back to the previous priority discussion. An enterMale() & females() chord may (and almost certainly will, at some point) be on the queue at the same time as an enterFemales() & females() & open() chord. However, the rules of fairness we are working with state that as soon as the member of the opposite gender arrives, everyone is locked out until the bathroom empties and the opposite gender is given a chance to control the bathroom. This means that we want to always consume the enterMale() & females() chord instead of the enterFemales() & females() & open() chord. Therefore, in a chord implementation with our priority idea, we want to give the former chord some arbitrarily higher priority value than the latter.

The call to waitEmpty() within the first two chords above provides two functions: it blocks the entering thread until the bathroom empties out, then consumes the empty() message. Again, we run into a situation where an important message may find itself as part of more than one possible chord. As a result, we go back to the priority idea we were discussing earlier. The solution is very similar to the clearMales() solution. We simply assign the waitEmpty() & empty() chord a higher priority than the enterGender() & empty() & open() chord, to guarantee the proper behavior, i.e. we always want empty() to match with waitEmpty(), if waitEmpty() is in the queue at the time. The use of the suggested priority extension in both of these cases makes writing a fairer solution to the Unisex Bathroom Problem much easier.

The Smokers’ Problem and the Lack of a Novel Solution

The smokers' problem in concurrent programming is defined as something like this: there are three smokers, and a dealer that has three ingredients necessary to create a cigarette. The ingredients are paper, tobacco, and matches. We assume that the dealer has an unlimited quantity of each of these materials. Each smoker has one of the ingredients, and we want that smoker to pick up the two ingredients he needs when the dealer puts them on the table. In the case that the dealer knows what which smoker has which ingredient and takes the initiative to notify that smoker, the solution becomes trivial using semaphores. Essentially, the smoker code will be something like:

```
matches.wait();
dealer.getTobacco();
dealer.getPaper();
```

The dealer, in turn, will simply call a `matches.notify()` when it decides to make tobacco and paper available. Rather than putting the materials out for anyone, this version only notifies the smoker the dealer knows needs the materials he is about to make available. We can mirror this solution using chords, in which case the trivial solution looks something like this:

```
when tobacco() & paper()
{
      SmokerA.notify();
}
```

Again, the code is essentially identical for the two other smokers, and again we see that we are in a situation where the dealer is aware of exactly which smoker is looking for what, rather than the dealer merely presenting ingredients to the smokers for them to take themselves. It's nice to be able to read "when tobacco and paper are available, notify a particular smoker," it is certainly more readable than the semaphore solution. It is not, however, particularly novel compared to using semaphores.

Only when we make the smokers themselves check the table do we encounter a dangerous deadlock situation that we need some amount of cleverness to resolve.

The naïve non-solution that results in deadlock looks like this:

```
Smoker with matches:

      tobacco.wait ()
      paper.wait ()
      agentSem.signal ()

Dealer A:

      agentSem.wait ()
      tobacco.signal ()
      paper.signal ()
```

This is a classic deadlock situation. Downey explains the situation in his Little Book of Semaphores,

> Imagine that the agent puts out tobacco and paper. Since the smoker with matches is waiting on tobacco, it might be unblocked. But the smoker with tobacco is waiting on paper, so it is possible (even likely) that it will also be unblocked. Then the first thread will block on paper and the second will block on match. Deadlock! (Downey, 103)

In order to arrive at a solution to the non-trivial problem using chords, we have two options. On the one hand, since we know how to use chords to write our own semaphores, we can essentially copy Downey's solution. We're more interested in the possibility of a non-trivial solution that is new and novel. In order for this to be possible, the solution must make use of the queuing feature of chords. In particular, we might envision a Table class that uses chords like this:

```
when putTobacco()
      & putPaper()
      & getTobacco(Smoker s)
      & getPaper(Smoker s)
{
```

```
            s.notify();
      }
```

Right away, however, we run into a grammar problem. We have two instances of the Smoker class with the same name, "s." Of course, we really want these to be the same instances, so maybe that's not such a bad thing. When we think about the smoker with matches and the smoker with paper both calling table.getTobacco(), however, we realize that there is a serious problem. We cannot guarantee that the queue manager will match the call to getTobacco() by the smoker with matches with the call to getPaper() by the smoker with matches. In fact, if the smoker with paper calls getTobacco() after the smoker with matches calls it, and before he calls getPaper(), we expect that the queue manager will identify and run the pattern. We need a way of making sure that each pair of calls made by each smoker are made atomically, and this basically brings us back to the traditional semaphore solution, which looks like this:

```
      tobacco.wait ()
      mutex.wait ()
            if (isPaper)
                  isPaper = false
                  matchSem.signal
            else if (isMatch)
                  isMatch = false
                  paperSem.signal
            else
                  isTobacco = true
      mutex.signal ()
```

It does not seem that we will be able to improve on this solution using chords.

The H2O Problem and Joining a Method with Itself

Programming with chords is much better suited for the H2O problem than the smokers problem. Basically, this is a very simple problem with just a couple of rules. We have a barrier and two types of threads allowed to enter the barrier: oxygen and hydrogen. When two hydrogen threads and an oxygen thread are all present in the barrier, the three threads are all supposed to call “bond()” and exit the barrier. Using chords, a first instinct might be to construct something that looks like this:

```
public void enterOxygen(Oxygen o)
      & enterHydrogen(Hydrogen i)
      & enterHydrogen(Hydrogen h)

{
      //call bond
}
```

However, at least in the Polyphonic C# implementation, it is grammatically incorrect to join a method with itself. Luckily, there is a very simple workaround.

```
when enterHydrogen(Hydrogen h) & enterOxygen(Oxygen o) & done()
{
      waitingForHydrogen(h, o);
}

public void waitingForHydrogen(h2, o) & enterHydrogen(Hydrogen h)
{
            //call bonds
            done();
}
```

Rather than trying to join all three events that are necessary to bond a water molecule, we create a new event and split the four events into two pairs. The new event, “waitingForHydrogen,” is now triggered by a single hydrogen and oxygen pairing. We do not care about the order in which the hydrogen and oxygen events occur, because all the chord mechanism queues them all for us. The way the language is constructed guarantees that when we have two hydrogen events and an oxygen event, a blocking call on the body of waitingForHydrogen() will be executed. The blocking part is important, because the last rule of the H2O problem is that we want to guarantee that the calls to bond() all form a new water molecule. That is to say, we want to avoid an extra hydrogen from another thread calling bond() before oxygen has a chance to complete the molecule. The done() message ensures that no other molecule is formed before the previous atoms are finished bonding. In the previous theoretical example, we used the done() message to enforce this rule. Another solution would be something like,

```
when enterOxygen(Oxygen o)
      & enterHydrogen(Hydrogen i)
      & enterHydrogen(Hydrogen h)

{
      bondAll(o, i, h);
}

public void bondAll(Oxygen o, Hydrogen i, Hydrogen h)
{
      lock(this)
      {
            //calls to bond()
      }
}
```

This would also guarantee that the three calls to bond() occur in the correct order, preventing an extra oxygen or hydrogen from calling bond() prematurely by locking the barrier object.

That is something of a moot example since, as we observed earlier, joining a method with itself is not allowed. This is likely due to the use of bitsets to represent the chord combinations. It is interesting to note that C#, and all .NET languages, are allowed to use metadata around code as instructions to the compiler. This is, for instance, how async functions are defined. There is a [OneWay] attribute that can be placed in front of any method that does not require the calling thread to wait for a return value, and this signals to the compiler that it may take and run this method in a new thread or one from the thread pool. In a similar fashion, we might imagine using attributes as a way of attaching metadata to the chord compiler. In particular, we might be able to use attributes to specify how many instances of a chord event we want in order to trigger execution of the chord's body. In the H2O example, we might use such a feature in the following way,

```
when enterOxygen(Oxygen o) & enterHydrogen(Hydrogen h) [2]
{
      bondAll(o, i, h);
}
```

The absence of an attribute from enterOxygen() would imply to the chord matching algorithm a single call, and any positive, non-zero integer would be a valid value for an attribute following a method we wish to join with itself.

Taking a careful look at this scenario, however, we see a recurrence of a problem we ran into in the smokers' problem. While the body of the chord above contains a call to bondAll with parameters o, i, and h, the variable i has disappeared! Whoops. This should also be a fairly trivial problem to solve, however. One solution would be to follow suit of scripting languages like Perl that use parameter collections, and make each parameter actually a reference to a collection of the parameters of the same name. For instance, in the case of the H2O problem,

```
when enterOxygen(Oxygen o) & enterHydrogen(Hydrogen h) [2]
{
            bondAll(o, h[0], h[1]);
}
```

This solution will seem obvious to those familiar with less strongly-typed languages, but may not be intuitive otherwise. The transformation in type of the "h" variable is a possible cause of confusion for someone reading this code, but so long as the behavior is well-documented, it could be an acceptable solution.

The River Crossing Problem

This problem is similar to the previous one, but contains a new twist. The situation is phrased like this: we have a dock and some rowboats, and people want to use the rowboats to cross from Seattle to Redmond. There are two types of people who use this dock: Microsoft serfs and Linux hackers. Each rowboat has four seats. A boat may not contain only a single serf or a single hacker. Rather, boats may contain all serfs, all hackers, or two serfs paired with two hackers. This is, so the story goes, so the lonely serf or hacker is not beaten up on the trip across while being outnumbered on the boat. A half-and-half boat keeps the peace because there would be no clear winner.

Again, our first impulse using chords would be grammatically incorrect. We might be tempted to try something like,

```
when enterSerf(Serf s1)
      & enterSerf(Serf s2)
      & enterSerf(Serf s3)
      & enterSerf(Serf s4)
{
      //board boat
}
```

And,

```
when enterSerf(Serf s1)
      & enterSerf(Serf s2)
      & enterHacker(Hacker h1)
      & enterHacker(Hacker h2)
{
      //board boat
}
```

Of course, the situation of a boat full of hackers is identical to the boat full of serfs. Now, it is very tempting to try to apply the solution to this situation. However, as we try different ideas for breaking down the sets of events into other events, it becomes clear the workaround found in the H2O problem is not a general solution to problems that are well-suited to joining a method with itself. The biggest problem is another grammar rule that makes sense when examined: a chord cannot be a subset of another chord. For instance, we might want to say,

```
when enterSerf()
{
      waitForSecondSerf();
}
public void waitForSecondSerf() & enterSerf()
{
      waitForThird();
}
public void waitForThird() & enterSerf()
{
      waitForFourthSerf();
}
& enterHacker()
{
      waitForFourthHacker();
}

public void waitForFourthSerf() & enterSerf()
{
      //load
}
public void waitForFourthHacker() & enterHacker()
{
      //load
}
```

The problem with this partial solution is that enterSerf() cannot be an event by itself as well as part of another chord. If this were allowed, there could be no guarantee that any of the other chords would ever execute. The language would be explicitly allowing a situation we know leads directly to a starvation problem. Chords do not guarantee freedom from starvation by any means, but this is not a grammar rule we want to find a way around. It is present for a very specific reason.

It is also interesting to note that this rule implies something non-obvious about the above proposal about allowing attributes to specify the quantity of calls to a particular method that need to be present in the queue to satisfy the chord. The rule about subsets must apply without regard to the quantity attribute. For instance, the following chord definitions,

```
when method1() & method2() {}
when method1() [2] & method2() [3] {}
```

would be illegal. Even though they are unique in the sense that one requires a greater quantity than the other, we must also observe that it is possible, and indeed very likely, that the second chord will never be activated, no matter how hard a user of the class that contains this chord definition tries. We must therefore be very careful with how we define and combine our chords and quantities. It is nice, though, that through this language construct we are able to prevent a certain kind of starvation at the compiler level, saving us from ourselves just a little.

Further Consideration: Atomic Execution of Chords Relative to Each Other

In the course of searching for a solution to the Unisex Bathroom Problem, it seemed that a special kind of monitor would be useful. In the end, the idea did not solve the problem we had hoped it solved, but it seems like it might be useful in other problems we have not yet considered. The behavior we would like to achieve is the mutual exclusion of chords being consumed. We do not just want to lock the object and prevent multiple chord bodies from executing in different threads at the same time. Instead, we may want to guarantee that certain other chords are not consumed from the queue until another chord is consumed and executed. With a traditional monitor lock, the chord will consume all the messages from the queue, then lock and wait, whereas we may want the ability to leave those messages on the queue to be consumed by a non-synchronized chord if the synchronized chord is being forced to block. The idea for this arose from the syntax for a monitor in Java, which looks like,

```
	synchronized public void m1() {}
```

This is simply a shorthand way of writing,

```
	public void m1() { synchronized(this) {} }
```

In Java and C#, we do not care if we have entered the method and then blocked, or blocked before we enter the method. Instead, executing a chord body has the effect of consuming messages that we may not really want to consume if we have to block right away. With this idea of not consuming messages if we will be forced to block, we might think of the synchronization code as looking more like,

```
	synchronized(this) {
		public void m1() & m2() {}
	}
```

We invert the synchronization with the chord, because we only want to consume the chord if we can first get a lock on the object. Otherwise, we want to leave the messages on the queue for others.

Conclusion

With a couple of simple extensions, chords allow us to solve problems in a more straight-forward and easily-understood way than complicated semaphore synchronization. Various priority ideas, including explicit priority values and first-in first-out, give us greater control over the resolution of conflicts between multiple chords fighting over the same method in a queue. Allowing a method to join with itself, subject to certain restrictions, maintains the integrity of the grammar while making solutions certain types of problems, like the river crossing problem and the H2O problem, nearly trivial. Chords, particularly with the suggested extensions, allow us to solve a particular class of problems elegantly, while the remaining swath of concurrency problems are at worst no more complicated than they were previously using semaphores.

Appendix A – Common Bouncing Ball Code (Windows Form and Ball classes)

(Note: There are two snippets of code that differ in the BouncingBallForm class. The C# implementation is in **bold** and the Comega implementation is in *italics*. The two implementations are mutually exclusive; only one or the other is used, never both. The Ball class is identical in both cases.)

```
public class BouncingBallForm : Form
{
    Button test, newBalls;
    Button bye;
    bool drawing = false;

    ArrayList balls = new ArrayList();

    public ComegaForm () {

        this.test = new System.Windows.Forms.Button();
        this.SuspendLayout();

        this.test.Location = new System.Drawing.Point(8, 300);
        this.test.Name = "Test";
        this.test.TabIndex = 0;
        this.test.Text = "Start Balls";
        this.test.Size = new System.Drawing.Size(80, 24);

        this.newBalls = new System.Windows.Forms.Button();

        this.newBalls.Location = new System.Drawing.Point(100, 300);
        this.newBalls.Name = "NewBalls";
        this.newBalls.TabIndex = 0;
        this.newBalls.Text = "New Ball";
        this.newBalls.Size = new System.Drawing.Size(80, 24);

        test.Click += new EventHandler(this.OnButtonClick);
        newBalls.Click += new EventHandler(this.createNewBall);
        test.Visible = false;
        newBalls.Visible = true;

        this.Controls.AddRange(new Control[]{ test });

        this.Controls.AddRange(new Control[]{ newBalls });

    }

    void createNewBall(object sender, EventArgs e)
    {
        Graphics g = this.CreateGraphics();
        Controller ball = new Controller(g);
        balls.Add(c);
        if(drawing)
        {
            //C# version
            Thread t = new Thread(new ThreadStart(ball.start));
            t.Start();

            //Comega version
            ball.start();

        }
    }

    void OnButtonClick(object sender, EventArgs e)
    {
         if(!drawing)
         {
              startAll();
              test.Text = "Stop Ball";
            }

            else
            {
                  stopAll();
                  test.Text = "Start Ball";
            }
            drawing = !drawing;
      }

      public void startAll()
      {
            foreach((Controller)ball in balls)
            {
                  //C# version
                  Thread t = new Thread(new ThreadStart(ball.start));
                  t.Start();

                  //Comega version
                  ball.start();
            }
      }

      public void stopAll()
      {
            foreach((Controller)ball in balls)
            {
                  ball.stop();
            }
      }
}

public class Ball
{
      // position in window
      protected Point   location, lastLocation;
      // radius of ball
      protected int radius;
      // horizontal change in ball position in one cycle
      protected double changeInX = 0.0;
      // vertical change in ball position in one cycle
      protected double changeInY = 0.0;
      // colour of ball
      protected Color   colour = Color.Blue;
      private Graphics g;

      // constructor for new ball
      public Ball( Point p, int rad, Color col, Graphics h)
      {
            location = p;
            radius = rad;
            colour = col;
            g = h;
      }

      // methods that   set   attributes of ball
      public void setColor( Color   newColor ) { colour     = newColor;
      }
      public void setRadius( int rad ) { radius =     rad; }
      public void setLocation( Point p ) { location = p; }
      public void setMotion( double dx, double dy     )
      {
            changeInX = dx;
            changeInY = dy;
      }

      // methods that   get   attributes of ball
      public int getRadius() { return     radius;     }
      public Point getLocation() { return location; }
      public Color getColour() { return colour; }

      // methods to move the ball
      public void reflectVert() {   changeInX = -changeInX; }
      public void reflectHorz() {   changeInY = -changeInY; }
      public void moveTo(     int   x, int y )
      {
            location.X = x;
            location.Y = y;
      }
      public void move() {
            lastLocation = location;
            location.Offset((int)changeInX,     (int)changeInY);
            if(location.X < g.VisibleClipBounds.Left || location.X >
g.VisibleClipBounds.Right)
            {
                  reflectVert();
            }
            if(location.Y < g.VisibleClipBounds.Top || location.Y >
g.VisibleClipBounds.Bottom)
            {
                  reflectHorz();
            }
      }

      // method to display ball
      public void paint(  )
      {

            //clear     the   old   ball
            g.FillEllipse(new SolidBrush( Color.White ),
                  lastLocation.X - radius,
                  lastLocation.Y - radius,
                  2*radius,
                  2*radius);

            g.FillEllipse( new SolidBrush( colour ),
                  location.X - radius,
                  location.Y - radius,
                  2*radius,
                  2*radius );
      }
}
```

Appendix C – C# Implementation of Controller class

```
public class Controller
{
      bool go;
      Ball ball;

      public Controller(Graphics g)
      {
            ball = new Ball(new Point(0,0), 10, Color.Blue, g);
            ball.setMotion(1, 1);

      }

      public void start()
      {
            go = true;
            while(go)
            {
                  ball.move();
                  ball.paint();
                  Thread.Sleep(5);
            }
      }

      public void stop()
      {
            go = false;
      }
}
```

Appendix D – Comega Implementation of Controller class

```
public class Controller
{
      public async checkStop();
      public async kill();
      public async start();

      bool go;
      Ball ball;
      public Controller(Graphics g)
      {
            ball = new Ball(new Point(0,0), 10, Color.Blue, g);
            ball.setMotion(1, 1);

      }

      when start()
      {
            checkStop();
            move();
      }

      private void move() & checkStop()
      {
            checkStop();
            ball.move();
            ball.paint();
            Thread.Sleep(5);
            move();
      }

       & kill()
      {
      }

      public async stop() & checkStop()
      {
            kill();
      }
}
```

Bibliography

Benton, Nick, Luca Cardelli, and Cédric Fournet. "Modern Concurrency Abstractions for C#." ACM Transactions on Programming Languages and Systems. 26 (2004): 769–804.

Downey, Allen B. Little Book of Semaphores. 2003. <http://allendowney.com/semaphores/downey03semaphores.pdf>

Wood, Tim. A Chorded Compiler for Java. London: Imperial College, 2004. <http://www.doc.ic.ac.uk/teaching/projects/Distinguished04/TimWood.pdf>