CS61B, Spring 2003 **Discussion #17** **Amir Kamil**
UC Berkeley **5/12/03**

Topics: Threading, Synchronization

1 Threading

Suppose we want to create an automated program that hacks into a server. Many encryption schemes use a very large product of two primes as a key. The product is publicized and can be used to encrypt data, but you can only decrypt data if you know what the two primes are. So let's have our program try factoring the key in order to break the encryption.

Unfortunately, factoring is hard, exponentially so ($O(2^n)$ in the number of bits in the key; keys of hundreds or thousands of bits are typical). So the our program will sit there forever trying to break the key, while we want it to try other hacking techniques as well. We can use *threading* in order to allow it to do both at the same time.

A *thread* is a stream of execution within a program. A program can consist of multiple threads that appear to run simultaneously, so it is possible for a program to do multiple things at once. For example, if you are writing a chess program, you can have the computer player "think" while it is the human player's turn. This act of thinking would be done in a different thread than the main program so that the human player can input his move.

In Java, a thread is represented by a `Thread` object. Associated with a particular thread is a method that runs when the thread is started. This method is analagous to the `main()` method that gets executed when a program is started and must have the signature `public void run()`.

In order to have a piece of code that runs in its own thread, a class that either extends `Thread` or implements `Runnable` must be defined. The piece of code then must be placed in the `run()` method of that class. In the case of a class that extends `Thread`, the thread is started by calling its `start()` method (inherited from the `Thread` class). In the case of a `Runnable`, the class must be wrapped by a generic `Thread` object, and the `start()` method of that `Thread` object called. The `start()` method does the actual thread creation and calls the appropriate `run()` method.

We can now modify our program to run the factoring in a separate thread and try other hacking methods at the same time.

2 Synchronization

While we now have programs that can execute multiple pieces of code at once, we have now introduced *synchronization* issues that must be dealt with. We may have certain pieces of code that we only want one thread to execute at a time, or resources that we don't multiple threads to simultaneously access. A simple example is a file that we are reading and writing. If two threads simultaneously write a file, then the written data can be garbled in unexpected ways. Think of what happens if two threads print something to the screen at the same time. The result will be unintelligible.

2.1 The `synchronized` Keyword

In the case of code fragments that we want to restrict to a single thread, the Java keyword `synchronized` is sufficient. A block of code that is `synchronized` can only be executed by one thread at a time. Any block can be `synchronized`, including entire methods, in which case the method can be declared `synchronized`. We will not cover `synchronized` blocks.

```
class Hacker {
  int factorKey(int key) {
    for (int i = 2; i < key; i++) {
      if (num % i == 0) {
        return i;
      }
    }
    return 0;
  }
}
```

Figure 1: A program that factors numbers.

In the example in figure 3, only one thread at a time can be in either the `writeFile()` method or the `synchronized` block in the `readFile()` method. The exact rules for `synchronized` are a bit complicated, and are as follows:

1. if two static methods of a class are `synchronized`, then only one thread can be in either of them at once
2. if two instance methods of a class are `synchronized`, then only one thread can be in either of them at once for a particular instance of the class
3. if an instance method of a class is `synchronized`, then a thread can be in the method for one instance at the same time another thread is in the method for another instance
4. if an instance and a static method of a class are both `synchronized`, then one thread can be in one and another in the other at the same time

To illustrate these rules, consider the program in figures 4 and 5. It creates four threads, with two calling a `synchronized` instance method (`Synch.foo()`) on the same instance, the third calling the same method on another instance, and the last calling a static `synchronized` method (`Synch.bar()`) in the same class. The methods just wait forever, so a thread that enters one of the methods claims the synchronization forever. What does the program print? The result is

```
Thread-0 executing foo() on object #0
Thread-3 executing bar()
Thread-2 executing foo() on object #1
Press enter to exit...

Exiting...
```

modulo the order of the first four statements (it varies in each run, due to the nondeterministic order in which the threads get run). So you see that thread 1 doesn't execute `foo()` since it tries calling it on the same instance as thread 0, which has claim to the instance's synchronization. But thread 2 executes `foo()` at the same time on a different instance. And thread 3 executes the static method `bar()` at the same time. But thread 4 cannot execute the instance method `baz()` at the same time thread 0 executes `foo()` on the same instance.

As you can see, the effect of the `synchronized` keyword is complicated. Without a thorough knowledge of how it works, it is very easy to make mistakes. Perhaps we can use higher level constructs to abstract away the details of how the keyword works.

2.2 Locks

There are many different synchronization constructs, and perhaps the simplest is a *lock*. A lock can be *acquired* and *released*, but as long as one thread owns a lock, any other thread that attempts to acquire it must wait until the first one releases it.

```
class Factor extends Thread {
  int number, factor;
  boolean running = true;
  Factor (int number) {
    this.number = number;
  }
  public void run() {
    factor = findFactor(number);
    running = false;
  }
  int findFactor(int num) {
    for (int i = 2; i < num; i++) {
      if (num % i == 0) {
        return i;
      }
    }
    return 0;
  }
}

class Hacker {
  Factor f;
  void hack(Server s) {
    f = new Factor(s.getKey());
    f.start(); // creates a new thread, calls f.run()
    while (f.running) {
      trySomethingElse(s);
    }
    // now we know the key, so we can crash the server
    crashServer(s, f.factor);d
  }
  void trySomethingElse(Server s) {
    sendVirus(s);
    ...
  }
  ...
}
```

Figure 2: A modified program that both factors and tries other hacking methods at the same time.

```
  public synchronized void writeFile(String s) {...}

  public synchronized String readFile() {...}
```

Figure 3: Examples of synchronized code.

```
class Synch {
  static int nextId = 0;
  int id = nextId++;    // the unique ID of this instance
  static Synch shared = new Synch(); // a shared instance of this class
  static boolean RUN = true;

  /** Waits forever, preventing other threads from claiming syncronization on
   *  this instance */
  synchronized void foo() {
    System.out.println(Thread.currentThread().getName() +
                       " executing foo() on object #" + id);
    while (RUN) {
      try {
        Thread.sleep(Long.MAX_VALUE);
      } catch (Exception e) {}
    }
  }

  /** Waits forever, preventing other threads from claiming syncronization on
   *  this class */
  synchronized static void bar() {
    System.out.println(Thread.currentThread().getName() +
                       " executing bar()");
    while (RUN) {
      try {
        Thread.sleep(Long.MAX_VALUE);
      } catch (Exception e) {}
    }
  }

  /** Waits forever, preventing other threads from claiming syncronization on
   *  this instance */
  synchronized void baz() {
    System.out.println(Thread.currentThread().getName() +
                       " executing baz() on object #" + id);
    while (RUN) {
      try {
        Thread.sleep(Long.MAX_VALUE);
      } catch (Exception e) {}
    }
  }

}
```

Figure 4: A class to test the effects of the `synchronized` keyword

```
class SynchTest {

  /** Tests the effects of the 'synchronized' keyword */
  public static void main(String[] args) throws java.io.IOException {
    class A extends Thread {
      /** Runs a synchronized method on the shared instance */
      public void run() {
        shared.foo();
      }
    };
    class B extends Thread {
      /** Runs a synchronized method on a different instance */
      public void run() {
        new Synch().foo();
      }
    };
    class C extends Thread {
      /** Runs a synchronized class method */
      public void run() {
        Synch.bar();
      }
    };
    class D extends Thread {
      /** Runs a different synchronized method on the shared instance */
      public void run() {
        shared.baz();
      }
    };

    Thread t0 = new A();
    Thread t1 = new A();
    Thread t2 = new B();
    Thread t3 = new C();
    Thread t4 = new D();

    t0.start();
    t1.start();
    t2.start();
    t3.start();
    t4.start();

    // Wait for the user to be satisfied that nothing more is going to happen.
    System.out.println("Press enter to exit...");
    System.in.read();
    System.out.println("Exiting...");
    System.exit(0);
  }
}
```

Figure 5: A program to test the effects of the `synchronized` keyword.

```
  private Lock fileLock = new Lock();
  public void writeFile(String s) {
    fileLock.acquire();
    ...
    fileLock.release();
  }
  public String readFile() {
    fileLock.acquire();
    ...
    fileLock.release();
  }
```

Figure 6: Using a lock to provide synchronized access to a file.

```
class Lock {
  private boolean locked;
  public synchronized void acquire() {
    while (locked) {}
    locked = true;
  }
  public void release() {
    locked = false;
  }
}
```

Figure 7: A lock class that can be used to restrict access to resources.

Using a lock, we can rewrite the `writeFile()` and `readFile()` methods to use locks instead of the `synchronized`. This can be done be using a lock that must be acquired before the file can be accessed. Only one thread can hold a lock, and all other threads must wait until the owner releases the lock. The rewritten code is in figure 6.

Unfortunately, the Java API does not have a lock implementation, so we'll have to implement our own. Using `synchronized` blocks, it isn't too difficult. The `acquire()` method must be `synchronized` so only one thread can acquire a lock at a time. We also need a flag in order to determine whether or not the lock has been acquired. A lock implementation is shown in figure 7.

Unfortunately, our lock code in figure 7 is inefficient. The loop in the `acquire()` method is an example of *busy waiting*, where a thread executes useless instructions in order to pass time. (Even if we add a call to `Thread.sleep()` as in figure 4, if we don't sleep long enough, it will still be inefficient, but if we sleep too long, then the thread won't acquire the lock immediately after it is released.) Instead, Java provides methods to allow threads to wait without using CPU time. Calling the `wait()` method of an object puts a thread to sleep on that object until another thread calls `notify()` or `notifyAll()` on that object (`notify()` wakes one thread while `notifyAll()` wakes all). An object relinquishes all claims on `synchronized` blocks when it sleeps and must reacquire them before continuing. We can rewrite `Lock` to be more efficient using these methods.

In addition, our lock code in figure 7 has protection issues. Any thread can call the `release()` method, even one that doesn't own the lock. Using the `Thread.currentThread()` method, we can prevent this from happening. The revised lock code is in figure 8.

2.3 Equivalence of `synchronized` and Locks *(Optional)*

An interesting question is whether or not locks are equivalent to the `synchronized` keyword. By *equivalent*, we mean it is possible to do everything you can with locks that you can with `synchronized`, and vice versa. Since we implemented locks using `synchronized`, we have proven that it is possible to do everything with

```
class Lock {
  private boolean locked;
  Thread owner;
  public synchronized void acquire() {
    while (locked) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    locked = true;
    owner = Thread.currentThread();
  }
  public synchronized void release() {
    if (owner == Thread.currentThread()) {
      locked = false;
      owner = null;
      notifyAll();
    }
  }
}
```

Figure 8: A more efficient lock. `acquire()` must catch `InterruptedException` since `wait()` can throw it. `release()` must be `synchronized` to allow `notifyAll()` to be called. Note that threads can attempt to acquire the lock while others are sleeping since they give up their synchronization claims.

`synchronized` that you can do with locks. All that is left is to prove the reverse.

This is more than just a theoretical exercise. If we can show that locks and `synchronized` are equivalent, perhaps we can better understand how `synchronized` works throught the equivalence relation. If the relation is simple enough, we won't have to remember all the complicated rules that define how `synchronized` works.

In order to prove the equivalence, we will try to show how an arbitrary use of `synchronized` can be replaced by use of a lock. (Actually, since we are only discussing `synchronized` methods, we will only go as far as showing how such methods can have the same behavior using locks. It is not much harder to show how `synchronized` blocks can be replaced with locks as well.) To start with, recall how we reimplemented our file access methods using a lock. To do so, we had to acquire a lock at the beginning of each method, and release it at the end. Let's try generalizing this to arbitrary `synchronized` methods.

The main question then, is what lock should we use in each method? Let's try to answer this question by analyzing each rule above for how `synchronized` works:

1. In order to prevent multiple threads from executing a class's static methods at once, all static methods of a particular class should use the same lock. But static methods of different classes should use different locks.

2. In order to prevent multiple threads from executing instance methods concurrently, all instance methods of a particular instance should use the same lock.

3. In order to allow different threads to execute the same instance method on different instances at the same time, instance methods of different instances should use different locks.

4. In order to allow different threads to execute a static and an instance method at the same time, static and instance methods should use different locks. (This also follows from the rule above.)

The simplest scheme to accomplish these goals is to associate a lock with each class, and a lock with each instance. Then the formerly `synchronized` static methods would use the class's lock, and the instance

```
class Synch {
  static int nextId = 0;
  int id = nextId++;    // the unique ID of this instance
  static Synch shared = new Synch(); // a shared instance of this class
  static boolean RUN = true;

  private static Lock ourLock = new Lock(); // lock for class methods
  private Lock myLock = new Lock();         // lock for instance methods

  /** Waits forever, preventing other threads from claiming syncronization on
   *  this instance */
  void foo() {
    myLock.acquire();
    System.out.println(Thread.currentThread().getName() +
                       " executing foo() on object #" + id);
    while (RUN) {
      try {
        Thread.sleep(Long.MAX_VALUE);
      } catch (Exception e) {}
    }
    myLock.release();
  }

  /** Waits forever, preventing other threads from claiming syncronization on
   *  this class */
  static void bar() {
    ourLock.acquire();
    System.out.println(Thread.currentThread().getName() +
                       " executing bar()");
    while (RUN) {
      try {
        Thread.sleep(Long.MAX_VALUE);
      } catch (Exception e) {}
    }
    ourLock.release();
  }

  /** Waits forever, preventing other threads from claiming syncronization on
   *  this instance */
  void baz() {
    myLock.acquire();
    System.out.println(Thread.currentThread().getName() +
                       " executing baz() on object #" + id);
    while (RUN) {
      try {
        Thread.sleep(Long.MAX_VALUE);
      } catch (Exception e) {}
    }
    myLock.release();
  }

}
```

Figure 9: The `Synch` class implemented with locks instead of the `synchronized` keyword.

```
class Lock {
  private int locked = 0;
  Thread owner;
  public synchronized void acquire() {
    if (Thread.currentThread() == owner) {
      locked++;
      return;
    }
    while (locked > 0) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    locked = 1;
    owner = Thread.currentThread();
  }
  public synchronized void release() {
    if (owner == Thread.currentThread()) {
      locked--;
      if (locked == 0) {
        owner = null;
        notifyAll();
      }
    }
  }
}
```

Figure 10: A lock that allows the owner to call `acquire()` multiple times.

methods would use the instance's lock. Figure 9 illustrates this translation for the `Synch` class. The result of running `SynchTest` is exactly the same as before.

There is a small problem with our translation. What happens if methods `A()` and `B()` are both `synchronized`, and `A()` calls `B()`? A thread will acquire the lock at the beginning of `A()`, and then again at the beginning of `B()`. But the lock is taken (by that thread), so the thread will go to sleep. Since the owner is asleep waiting for the owner to wake it up, it will sleep forever.

We can solve this problem by checking if the thread that calls `acquire()` is the owner, and not going to sleep if it is. But then what happens when `B()` returns? The thread will release the lock, even though `A()` has not finished executing. This violates the synchronization requirement. A fix will require keeping track of how many times the owner of a lock has called `acquire()`, and forcing it to release the lock the same number of times. (The term for this is a *semaphore*, but we won't make that distinction here.) The resulting lock implementation is in figure 10.

We have now shown that locks and the `synchronized` keyword are equivalent. This means that we can accomplish any synchronization allowed by Java using only locks. Also, since the equivalence relation is so simple, we don't need to know the rules of how `synchronized` works. Instead, we can just remember that the equivalent is that `synchronized` static methods use a class lock, and that `synchronized` instance methods use an instance lock.

2.4 Conclusion

The introduction of parallelism into programs can potentially result in faster programs. This, however, comes at a price. Parallel programs can interfere with each other, since sharing resources can be a problem. With the use of proper synchronization, this issue can be resolved, though an extra level of complexity is added to the program.

Synchronization issues are particularly important in operating systems. While it may be unlikely that two threads execute the same code at the exact same time, we don't want our computers to crash when it does happen. It's errors like these that are the most difficult to fix, since they are difficult to reproduce. If you are running a `Windows` machine and want to see an example of synchronization, try opening a file in `Microsoft Word` and then in `WordPad` at the same time. You'll see that the operating system won't let you.