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.

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`.

```
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. We say However, one thread can be in one method while another thread is in the other, which can still cause unpredictable results. What we really want is to allow only one thread at a time access to the file. 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. Using `synchronized` blocks, locks are easy to define.

Unfortunately, our lock code in figure 4 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. 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 4 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.

Synchronization issues are 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.

```
class Factor extends Thread {
  int number, factor;
  boolean running = false;
  Factor (int number) {
    this.number = number;
  }
  public void run() {
    running = true;
    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 String readFile() {
    synchronized {...}
  }
```

Figure 3: Examples of synchronized code.

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

Figure 4: A lock class that can be used to restrict access to resources.

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

  public String readFile() {
    lock.acquire();
    ...
    lock.release();
  }
```

Figure 5: Using a lock to provide synchronized access to a file.

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

Figure 6: A more efficient lock. acquire() can throw an InterruptedException since wait() can. 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.