

Synchronization via Locking in C++

In a multi-threaded program (in Java, C, C++, or any language for that matter), it is necessary to provide mechanisms in the language so that a programmer can ensure *serialized* access to shared variables, objects, or methods. Serialization is achieved using various synchronization mechanisms. A common synchronization mechanism is called a *mutual exclusion lock*.

The basic technique is to ensure that the lock is set before a shared variable/object/method is accessed and that the lock is reset when access is complete. In general, a lock should be held no longer than necessary. A very common bug in concurrent programming is to forget to release the lock in some thread, in which case all threads waiting for the lock cannot progress! For example:

```
int x = 0; // a shared variable
...
void synchronized_procedure()
{
    static mutex m; // a "mutual exclusion" lock object

    m.lock();
    x = x + 1; // access shared variabl
    if (x < 5) {
        ...
    }
    else
        return; // THIS CAUSES A PROBLEM, WHY!?
    ...
    m.unlock();
}
```

Failure to release a lock usually occurs when there is more than one exit point from a procedure.

A Simple C++ Lock Object

Assuming we already have a mutex lock type implemented, we can easily implement a C++ class for a lock object as follows:

```
class Lock {
    mutex& m;
public:
    Lock(mutex& x) : m(x) { m.lock(); }
    ~Lock() { m.unlock(); }
};
```

The constructor grabs the lock and the destructor releases the lock, thereby *guaranteeing* that the lock is always released when the scope in which a Lock object is declared is exited. This is really pretty neat, and ensures that you never forget to release the lock.

The only problem is that you might not want to always set the lock at the start of the scope and release it at the end. You might want to have a finer granularity of locking. In this case, you need to just use the lock directly, always remembering to unlock it as soon as you are done accessing the shared information. You could also allocate a lock object from the heap, as long as you remember to delete it at the right time. To complicate matters, if exception handling is being used in a multi-threaded program, and there is some procedure that set a lock, but does not catch the exceptions, then the lock will not be released when the stack is unwound as part of throwing an exception. So, the safest thing to do in C++ is to use the Lock object as a stack allocated object associated with some scope.

Question: What happens to the lock in the following situation, when cond is true?

```
for (;;) {
    static mutex m;
    Lock lock(m);
    if (cond)
        break;
    ...
}
```

Synchronization in C++ using Locks tied to a Scope

It is often useful to associate a lock an entire lexical block (e.g., a procedure), which provides mutual exclusion over the entire scope of the block and there is a guarantee that no matter where the scope is exited, the lock is always released.

How might you implement a locking strategy in C++ so that the lock is set when a scope is entered, and unlocked when the scope is exited, independent of the point at which you exit the scope?

```
void synchronized_procedure()
{
    static mutex m; // why is this a static object?
    Lock lock(m); // associate a lock with the entire block scope

    x = x + 1; // access shared variabl
    if (x < 5) {
        ...
    }
    else
        return; // lock implicitly released
    ...
} // lock implicitly released at end of scope
```

In this case, the granularity of the lock is the entire scope of the `synchronized_procedure`. In C and C++, you can define an arbitrary scope, and declare a Lock variable local to that scope to effect mutual exclusion for a single statement or a group of statements. E.g.,

```
#define MUTEX_BEGIN() do { static mutex m; Lock lock(m);
#define MUTEX_END() } while(0)
...
MUTEX_BEGIN();
x = x + 1;
MUTEX_END();
```

Locking in Java using Synchronized Methods

Java does not have an explicit mutex or lock object, but lock synchronization is implemented by the virtual machine. A lock in Java is applied to an entire object. So, locking in Java applies to an entire object. The lock itself is part of the implementation of the Object class, which all other classes inherit from. So, all objects in Java have a lock associated with them.

To make life easy (depending on how you look at it), Java provides the **synchronized** modifier as the mechanism the programmer uses to indicate that the lock for the object should be set when a method is called by a thread, and implicitly released when the method returns (normally or abnormally). This way, the programmer is protected from forgetting to release the lock. **Forgetting to release a lock is a common bug in multi-threaded programming.**

A simple example:

```
class BankAccount {
    private double balance;

    public synchronized void withdrawal (double amount) {
        balance = balance - amount;
    }

    public synchronized void deposit (double amount) {
        balance = balance + amount;
    }
}
```

When a method executing in thread 1 calls withdrawal, the object is locked, and no other thread can execute the withdrawal or deposit methods. Other threads can however call the method, but the Java virtual machine will make sure the thread is blocked from executing the method until the object is unlocked.

If an exception is thrown from a synchronized method, then the object lock is implicitly released.

The Java “synchronized” Statement

The synchronized modifier for methods is really just a convenience of notation. Java also provides a **synchronized** statement that can be used to lock an object without calling any methods. In effect, when you call a synchronized method, such as the `BankAccount.withdrawal` method in the previous example, it is equivalent to calling:

```
class BankAccount {
    private double balance;

    public void withdrawal (double amount) {
        synchronized(this) {
            balance = balance - amount;
        }
    }
    public void deposit (double amount) {
        synchronized (this) {
            balance = balance + amount;
        }
    }
}
```

The synchronized statement takes a non-null object reference as an argument, and locks the object if the object is not already locked. What do you think happens in the following case, called **recursive locking**?

```
class Test {
    public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
            synchronized(t) {
                System.out.println("This is a test");
            }
        }
    }
}
```

The Reader Thread

```
class Reader extends Thread {
    private Client client;

    public Reader(Client c) {
        super("Client Reader"); // pass identifying string to Thread(String) constructor
        this.client = c;
    }

    public void run() {
        BufferedReader in = null;
        String line;
        try {
            in = new BufferedReader(new InputStreamReader(client.socket.getInputStream()));
            while (true) {
                line = in.readLine();
                if (line == null) {
                    System.out.println("server closed connection");
                    break;
                }
                System.out.println(line);
            }
        } catch (IOException e) {
            System.err.println("Reader: " + e);
        } finally try {
            if (in != null) in.close();
        } catch (IOException e) { System.err.println(e); System.exit(0); }
    }
}
```

Using Threads to Implement a Multi-Threaded Network Client

It is often useful to write a network client that contains two threads: a Reader thread and a Writer thread. The Reader thread listens for data from the server and sends it to the console or a file. The Writer thread takes input from the user or a file, and sends it to the server.

```
package ReaderWriter;
import java.net.*;
import java.io.*;
public class Client {
    Socket socket;
    private Thread reader, writer;

    public Client (String host, int port) {
        try {
            socket = new Socket(host, port); // Socket constructor connects to host on port
            reader = new Reader(this);
            writer = new Writer(this);
            // give the reader higher priority
            reader.setPriority(6);
            writer.setPriority(5);
            reader.start();
            writer.start();
        } catch (IOException e) { System.err.println(e); }

        // main just creates the client object, which starts the reader/writer threads
        public static void main(String[] args) {
            try { new Client(args[0], Integer.parseInt(args[1])); }
            catch (NumberFormatException e) { System.err.println(e); System.exit(-1); }
        }
    }
}
```

The Writer Thread

```
class Writer extends Thread {
    private Client client;

    public Writer(Client c) {
        super("Client Writer");
        this.client = c;
    }

    public void run() {
        BufferedReader in = null;
        PrintWriter out = null;
        try {
            String line;
            in = new BufferedReader(new InputStreamReader(System.in));
            out = new PrintWriter(client.socket.getOutputStream());
            while (true) {
                line = in.readLine();
                if (line == null) {
                    break;
                }
                out.println(line);
            }
        } catch (IOException e) {
            System.err.println("Writer: " + e);
        } finally try { if (out != null) out.close(); }
        catch (IOException e) { System.err.println(e); System.exit(0); }
    }
}
```