

Synchronization in Java

We often want threads to co-operate, typically in how they access shared data structures.

Since thread execution is asynchronous, the details of how threads interact can be unpredictable.

Consider a method

```
update() {  
    n = n+1;  
    val = f(n);  
}
```

that updates fields of an object.

If two or more threads execute **update** concurrently, we might get unexpected or even illegal behavior.

(Why?)

A Java method may be *synchronized*, which guarantees that at most one thread can execute the method at a time. Other threads wishing access, are forced to wait until the currently executing thread completes.

Thus

```
void synchronized update() { ... }
```

can safely be used to update an object, even if multiple threads are active.

There is also a **synchronized** statement in Java that forces threads to execute a block of code sequentially.

```
synchronized(obj) {  
    obj.n = obj.n+1;  
    obj.val = f(obj.n);  
}
```

Synchronization Primitives

The following operations are provided to allow threads to safely interact:

- `wait()` Sleep until awakened
- `wait(n)` Sleep until awakened or until `n` milliseconds pass
- `notify()` Wake up one sleeping thread
- `notifyAll()` Wake up all sleeping threads

Using these primitives, correct concurrent access to a shared data structure can be programmed.

Consider a `Buffer` class in which independent threads may try to store or fetch data objects:

```
class Buffer {
    private Queue q;
    Buffer() { q = new Queue(); }
    public synchronized void
    put(Object obj) {
        q.enqueue(obj);
        notify(); //Why is this needed?
    }
    public synchronized Object
    get() {
        while (q.isEmpty()) {
            //Why a while loop?
            wait();
        }
        return q.dequeue();
    }
}
```

Locks, Semaphores and Monitors

Java's synchronization mechanisms are based upon the notion of a *lock*. A lock is a special value that can be held by at most one thread.

If a thread holds a lock, it has permission to do some "critical" operation like writing a shared variable or restructuring a shared data object.

If a thread wants to do an operation but doesn't hold the necessary lock, it must wait until it gets the lock.

In Java there is a lock associated with each run-time object.

Lock Granularity and Access

Though each Java object has a lock, you often don't want to lock and unlock each object you access.

If you are manipulating a large data structure (like a tree or hash table), acquiring the lock for each object in the tree or table can be costly and error-prone.

Instead, it is common to create a lock corresponding to a group of objects. Hence holding the lock to the root of a tree may give you permission to access the whole tree.

There is a danger though—if all or most of a large structure is held by one thread, then other threads won't be able to access that structure concurrently.

For example, a large shared data base (like one used to record current bookings for an airline) shouldn't be held exclusively by one thread—hundreds of concurrent threads may want to access it at any time. An intermediate lock, like all reservations for a single flight, is more reasonable. There is also an issue of how long you hold a lock. The ideal is to have exclusive access for as short a period as is necessary. Work that is not subject to interference by other threads (like computations using local variables) should be done before a lock is obtained. Hence Java's **synchronized** statement allows a method to get exclusive access to an object for a limited region, enhancing shared access.

Deadlock

A variety of programming problems appear in concurrent programs that don't exist in ordinary sequential programs.

The most serious of these is *deadlock*:

Two or more threads hold locks that other threads require. Each waits for the other thread to release a needed lock, and no thread is able to execute.

As an example of how deadlock may occur, consider two threads, τ_1 and τ_2 . Each requires two files, a master file and a log file. Since these files are shared, each has a lock.

Assume τ_1 gets the lock for the master file while τ_2 (at the same instant) gets the lock for the log file.

Now each is stuck. Each has one file, and will wait forever for the other file to be released.

In Java deadlock avoidance is wholly up to the programmer. There are no language-level guarantees that deadlock can't happen.

Some languages have experimented with ways to help programmers avoid deadlock:

- If all locks must be claimed at once, deadlock can be avoided. You either get all of them or none, but you can't block other threads while making no progress yourself.
- Locks (and the resources they control) can be ordered, with the rule that you must acquire locks in the

proper order. Now two threads can't each hold locks the other needs.

- The language can require that the largest set of locks ever needed be declared in advance. When locks are requested, the operating system can track what's claimed and what may be needed, and refuse to honor unsafe requests.

Fairness & Starvation

When one thread has a lock, other threads who want the lock will be suspended until the lock is released.

It can happen that a waiting thread may be forced to wait indefinitely to acquire a lock, due to an unfair waiting policy. A waiting thread that never gets a lock it needs due to unfair lock allocation faces *starvation*.

As an example, if we place waiting threads on a stack, newly arrived threads will get access to a lock before earlier arrivals. This can lead to starvation. Most thread managers try to be fair and guarantee that all waiting threads have a fair chance to acquire a lock.

How are Locks Implemented?

Internally, Java needs operations to acquire a lock and release a lock. These operations can be implemented using the notion of a *semaphore*.

A semaphore is an integer value (often just a single bit) with two atomic operations: *up* and *down*.

up(s) increments **s** atomically.

down(s) decrements **s** atomically.

But if **s** is already zero, the process doing the **down** operation is put in a wait state until **s** becomes positive (eventually some other process should do an **up** operation).

Now locks are easy to implement.

You do a **down(lock)** to claim a lock. If someone else has it, you are forced

to wait until the lock is released. If the lock value is > 0 you get it and all others are "locked out."

When you want to release a lock, you do `up(lock)`, which makes `lock` non-zero and eligible for another thread to claim.

In fact, since only one thread will ever have a lock, the lock value needs to be only one bit, with 1 meaning currently free and unlocked and 0 meaning currently claimed and locked.

Monitors

Direct manipulation of semaphores is tedious and error-prone. If you acquire a lock but forget to release it, threads may be blocked forever.

Depending on where **down** and **up** operations are placed, it may be difficult to understand how synchronization is being performed.

Few modern languages allow direct use of semaphores. Instead, semaphores are used in the implementation of higher-level constructs like *monitors*.

A monitor is a language construct that guarantees that a block of code will be executed synchronously (one thread at a time).

The Java **synchronized** statement is a form of monitor.

When

```
synchronized(obj) { ... }
```

is executed, “invisible” **getLock** and **freeLock** operations are added:

```
synchronized(obj) {  
  getLock(obj)  
  ...  
  freeLock(obj);  
}
```

This allows the body of the **synchronized** statement to execute only when it has the lock for **obj**. Thus two different threads can never simultaneously execute the body of a **synchronized** statement because two threads can't simultaneously hold **obj**'s lock.

In fact, synchronized methods are really just methods whose bodies are enclosed in an invisible **synchronized** statement.

If we execute

```
obj.method( )
```

where **method** is synchronized, **method**'s body is executed as if it were of the form

```
synchronized(obj) {  
    body of method  
}
```

Operations like **sleep**, **wait**, **notify** and **notifyAll** also implicitly cause threads to release locks, allowing other threads to proceed.

Reading Assignment

- A Comparative Overview of C#
(linked from class web page)

Java 1.5 (Tiger Java)

Recently, Java has been extended to include a variety of improvements, many drawn from functional languages.

Added features include:

- Parametric polymorphism.

Classes and interfaces may be parameterized using a type parameter.

```
class List<T> {  
    T head;  
    List<T> tail;  
}
```

Interfaces may also be parameterized.

- Enhanced loop iterators.

```
for (v : myArray) {  
    // each element of myArray  
    // appears as a value of v }  
}
```

- Automatic boxing and unboxing of wrapper classes.

Conversion from `int` to `Integer` or `Integer` to `int` is now automatic.

- Typesafe enumerations.

```
public enum Color {RED, BLUE, GREEN};
```

- Static imports.

You may import all static members of a class and use them without qualification. Thus you may now write `out.println` rather than `System.out.println`.

- Variable argument methods.

- Formatted output using `printf`:

```
out.printf("Ans = %3d", a+b);
```

C#

C# is Microsoft's answer to Java. In most ways it is very similar to Java, with some C++ concepts reintroduced and some useful new features.

Similarities to Java include:

- C# is object-based, with all objects descended from class `object`.
- Objects are created from classes using `new`. All objects are heap-allocated and garbage collection is provided.
- All code is placed within methods which must be defined within classes.
- Almost all Java reserved words have C# equivalents (many are identical).
- Classes have single inheritance.

- C# generates code for a virtual machine to support cross-platform execution.
- Interfaces are provided to capture functionality common to many classes.
- Exceptions are very similar in form to Java's.
- Instance and static data within an object must be initialized at point of creation.

C# Improves Upon Some Java Features

- Operators as well as methods can be overloaded:

```
class Point {  
    int x, y;  
    static Point operator + (  
        Point p1, Point p2) {  
        return new Point(p1.x+p2.x,  
                          p1.y+p2.y);  
    }  
}
```

- Switch statements may be indexed by string literals.
- In a switch, fall throughs to the next case are disallowed (if non-empty).
- Goto's are allowed.
- Virtual methods must be marked.

- Persistent objects (that may be stored across executions) are available.

C# Adds Useful Features

- *Events* and *delegates* are included to handle asynchronous actions (like keyboard or mouse actions).
- *Properties* allow user-defined read and write actions for fields. You can add `get` and `set` methods to the definition of a field. For example,

```
class Customer {
    private string name;
    public string Name {
        get { return name; } }
}
Customer c; ...
string s = c.Name;
```


- *Indexers* allow objects other than arrays to be indexed. The `[]` operator is overloadable. This allows you to define the meaning of `obj[123]` or `obj["abc"]` within *any* class definition.
- Collection classes may be directly enumerated:
`foreach (int i in array) ...`
- Fields, methods and constructors may be defined within a *struct* as well as a class. Structs are allocated within the stack instead of the heap, and are passed by value. For example:

```
struct Point {  
    int x,y;  
    void reset () {  
        x=0; y=0; }  
}
```

- When an object is needed, a primitive (**int**, **char**, etc.) or a **struct** will be automatically *boxed* or *unboxed* without explicit use of a wrapper class (like **Integer** or **Character**). Thus if method **List.add** expects an object, you may write

```
List.add(123);
```

and **123** will be boxed into an **Integer** object automatically.
- *Enumerations* are provided:

```
enum Color {Red, Blue, Green};
```
- *Rectangular* arrays are provided:

```
int [,] multi = new int[5,5];
```
- Reference, out and variable-length parameter lists are allowed.
- Pointers may be used in methods marked **unsafe**.