Thread Synchronisation

- common resources
  - files
  - printers, scanners, robots...
  - shared data, global variables
- multiple threads
  - access common resources
  - wait if resource is not ready yet

Where is the problem?

What is the solution?

Java's Solution: Monitors

Historical note: Anthony Hoare, 1970s.

A monitor ***is*** an object.

Monitor methods can be used by only one thread at a time.

If a thread calls a monitor method of a monitor that is currently engaged then **the call blocks**, the thread has to wait.

Which Monitor Methods?

- commonest forms: the access modifier **`synchronized`** turns a method into a monitor method of *its object*.
- alternative: the statement
  `synchronized` `(expr) {  code  }`
  turns the code into a (parameterless) monitor method of the object the expr evaluates to.

Example: Mutable Variable

```
public class Variable {
  private Object local;
  synchronized public Object getVal()
     { return local; }
  synchronized public void putVal(Object a)
     { local = a; }
}
```

Explanation

- the methods `getVal` and `putVal` are monitor methods of any object of class `Variable`
- we cannot run two `getVal`s, or two `putVal`s or one `getVal` and one `putVal` *on the same object* in parallel
- however, `putVal` of one object is not in conflict with a `putVal` of *a different object* - different objects, different monitors!

Conflict Resolution

What if two or more threads request a monitor?

One gets it...(the monitor's **lock**)

The other threads block and have to wait...

The wait is not necessarily fair, i.e. it is not always a proper queue.

Other Methods

What if... some methods are `synchronized` and others are not?

The ones which are not simply fail to be monitor methods.

Anyone can call them, any time.

Surrender!

There is a situation which is not adequately dealt with so far.

It could happen that the execution of the monitor method reveals that there is a problem.

Another thread would need to provide a resource and that has not happened yet. So, the thread has to surrender the monitor, block, and await the resource.

Wait/Notify

- any Java object has methods `wait`/`notify`
- these can only be called when in possession of the objects monitor lock
- the `wait` method blocks and releases *this* lock (and *only* this lock)
- the `notify` method alerts waiting threads; they attempt to regain the lock after which they continue

Example

```
class Buffer {
  private Object local=null;
  public synchronized void put(Object a)
    { local = a; notify(); }
  public synchronized Object get()
    { if (local==null) wait();
      Object result=local;
      local=null;
      return result;
    }
}
```

Explanation (i)

- this is a buffer carrying one object
- `put` overwrites whatever is in the buffer
- `get` tries to fetch a non-**null** element from the buffer; it blocks when the current entry is **null**, otherwise it fetches the object and resets the buffer content

Explanation (ii)

- notice that several **`get`**-threads may be waiting
- if a **`put`** happens then the notify call will
  - awake one of the waiting threads which subsequently will proceed
  - do nothing if nobody is waiting

Race Hazard

Sadly, the code is not 100% correct.

It would be if lock-queuing were following some particular fair strategy - but we cannot rely upon that.

There is a scenario in which the buffer does not behave as wanted.

Scenario

1. get request is blocked (buffer empty)
2. put call fills buffer, thread is woken up and is *runnable* (but not yet running)
3. a second get request queues for the lock
4. it is given the lock instead of the first thread; it clears the buffer and releases the lock
5. now the first get resumes and sadly retrieves **null**

Modification

```
class Buffer {
  private Object local=null;
  public synchronized void put(Object a)
    { local = a; notify(); }
  public synchronized Object get()
    { while (local==null) wait();
      Object result=local;
      local=null;
      return result;
    }
}
```

Proper Buffer

In a proper buffer, **`put`** should block as well, i.e. if the buffer is already filled.

Not too hard, is it?

Proper Buffer?

```
public synchronized void put(Object a)
  { while (local!=null) wait();
    local=a; notify(); }

public synchronized Object get()
  { while (local==null) wait();
    Object result=local;
    local=null; notify();
    return result;
  }
```

No, another race hazard!

1. get1 is blocked
2. get2 is blocked
3. put3 succeeds, wakes 1
4. put4 blocks
5. get1 succeeds, **wakes 2**
6. get2 is blocked
7. If no further requests come, the system is dead with a put and a get waiting simultaneously!

Solution

Use **`notifyall()`** !
(instead of **`notify`**)

Not ideal, is it?

It does not look nice to notify **both** consumers and producers if only **one** of the two groups is affected by the action.

Cannot we organise it in such a way that consumers alert producers and *vice versa*, but that they leave their own kind undisturbed?

(Failed) Attempt

```
class Buffer {
  Object inq=new Object();
  Object outq=new Object();
  public synchronized void put(Object a)
  { while (local!=null)
      synchronized(outq) {outq.wait();}
    local=a;
    synchronized(inq) {inq.notify();} }
  ...
}
```

Deadlock

It does not work, it **deadlocks**.

The problem is: the wait only surrenders the lock *it is waiting on*, so this time the thread will keep the lock of the buffer itself, preventing other threads from accessing the buffer.

If we drop the `synchronized` modifier from the method then the deadlock goes away, but so does the security.

Solution

```
public void put(Object a){
  synchronized(outq) {
    synchronized(this) {
      if (local!=null) wait();
      local=a;
      notify();
    }
  }
}
```

Solution (ii)

```
public Object get(){
  synchronized(inq) {
    synchronized(this) {
      if (local==null) wait();
      Object result=local;
      local=null;
      notify();
      return result;
    }
  }
}
```

How does this work?

- in order for `put` to succeed it needs to be in possession of both the locks for **`this`** and `outq`
- if the buffer is full it relinquishes the lock for **`this`** but keeps the `outq` lock
- thus further `put` requests are bounced off, they do not call wait, they just queue on outq
- `get` requests can succeed and notify

Notice

- no more **`while`**'s, back to **`if`**'s
- at most one thread is waiting (as a result of wait) at any one time
- the waiting `put` thread cannot be overtaken by another `put` thread (as it fails to relinquish `outq` before completion)
- being overtaken by a `get` thread is harmless

Conclusions

Monitor synchronisation is rather subtle.

It does not scale very well.

Things can go wrong - no system checks for **deadlocks** or **race hazards**.

Use threads with caution!