



University of
Southampton

COMP2212 Programming Language Concepts

Locks and Synchronisation

Dr Julian Rathke

Locks as fundamental

- It seems clear that, in order to communicate via shared memory we need to allow threads to coordinate their activity for reading/writing
- To coordinate activity (independently of scheduling) we need some sort of blocking mechanism
- Locks achieve this by asking threads to block when the lock is not available
- We have seen that we can code up a locking mechanism using code such as Peterson's algorithm (with memory barriers) or, nowadays, using hardware supported atomic operations.
- Once we have built such locking mechanisms we can provide them as primitives in a higher-level language and use fixed, default implementations rather than trying to roll our own.
- Most languages that support concurrency also support some sort of high-level locking mechanism and synchronisation primitives.

Explicit Locks in Java

- Since Java 5 there has been a very useful package in Java called `java.util.concurrent`
- This package provides many high-level primitives for concurrency control, including locks.
- `java.util.concurrent.locks.ReentrantLock` is one such class.
- This class provides a simple lock with extended capabilities: e.g.
 - `tryLock()` allows for a non-blocking attempt to acquire the lock
 - `lockInterruptibly()` allows for an attempt to acquire the lock but allows the block thread to be interrupted
 - `getQueuedThreads()` returns the Collection of threads currently waiting on the lock.
- Note that client code using these locks is responsibly for both acquiring **and** releasing the lock! This can be dangerous.

Deadlocks


- Consider the following Java code showing the use of a lock

```
import java.util.concurrent.locks.ReentrantLock;

class Account {

    private int balance = 0;
    private ReentrantLock lock = new ReentrantLock();

    void deposit(int n){
        lock.lock();
        balance = balance + n;
        lock.unlock();
    }
    void withdraw(int n){
        deposit(-n);
    }
    int getBalance(){
        lock.lock();
        return balance;
    }
}
```

 **oops - no unlock**

let's write a piece of client code that uses this class.

Deadlocks

Imagine two customer threads that share a home and a work account. One thread is trying to make a deposit and the other a withdrawal.

```
class CustomerA {  
    ...  
    int hb = homeAccount.getBalance();  
    workAccount.withdraw(hb+100);  
    ...  
}  
class CustomerB {  
    ...  
    int wb = workAccount.getBalance();  
    homeAccount.deposit(wb-100);  
    ...  
}
```

- CustomerA gets the lock on the homeAccount via getBalance
- CustomerB gets the lock on the workAccount via getBalance
- Neither of these are released.
- Neither CustomerA or CustomerB can proceed because they are each holding lock that the other needs.

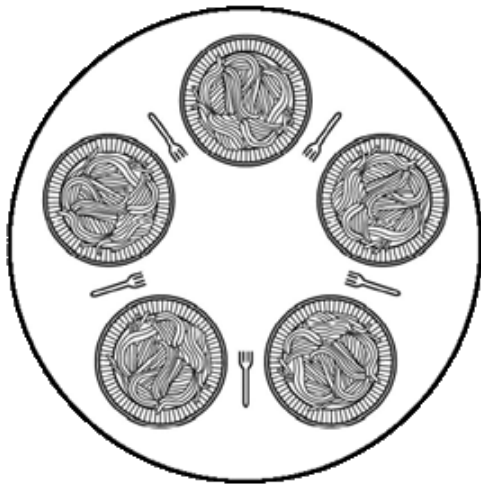
For a certain scheduling of threads the system will not make either transaction.

This situation is known as a **Deadlock** and it is surprisingly easy to encounter.

Deadlock refers to a situation in which two or more threads are both holding resources and waiting to acquire resources that others are holding, so that no threads can proceed.

Dining Philosophers

- Deadlocks can occur in any system where there are multiple threads and multiple locks being acquired and not released, or acquired released in the wrong order.
- The classic example to illustrate this is the Dining Philosophers Problem.



5 Philosophers sit around a table pondering the meaning of existence whilst trying to eat spaghetti. It takes two forks to eat spaghetti. Each philosopher is very fixed in their ways and they repeatedly grab the fork to their left, then the fork to their right, eat, think and then put down the left fork, followed by the right fork. Only a single philosopher can be holding a fork at any given time.

For a certain scheduling of threads, this system is can deadlock: each philosopher begins activity, each grabs the fork to their left and then tries to grab the fork to their right.

Oh dear, hungry philosophers.

Note that, if even just one philosopher went right fork first then the system would be fine!

Other locking concepts

- Two key concepts other than deadlock that one should be aware of with locking is that of
- **Lock Overhead** : it takes computation cycles to test, wait and acquire, and release locks. Using locks often to protect many small areas of code increases overhead.
- **Lock Contention** : refers to the situation in which one thread is trying to acquire a lock whilst another holds it. Using locks less frequently but protecting larger blocks of code increases the risk of contention.
- Choosing the **granularity** of locking is a trade-off between these two concepts.
- We refer to **Fine-grained concurrency** for systems with more locking of small blocks of code and
- **Coarse-grained concurrency** for systems with fewer locks over larger blocks of code.

Lock Composability Problem

- A major problem with the use of locks is their lack of clear composability.
- Consider the following use of the Account class above:

```
class Ledger {  
  
    public static void transfer(Account from, Account to, int n){  
        from.withdraw(n);  
        to.deposit(n);  
    }  
}
```

- This is not **thread safe**, i.e. if one thread calls this method and another checks the balances in the accounts then there may be unwanted behaviour
 - Balance checks falling between the withdraw and deposit of the transfer will be observed as a missing amount!
- This is even with lock protection on the balance in the Account class!
- To make this thread safe we would need to acquire locks on both accounts for the duration of the transaction.
 - This increases the risk of deadlock, overhead, and contention.
- In general, the pattern of lock acquires and releases is a tricky business.

Intrinsic Locking

- Relying on the client code to always release locks at appropriate times is considered dangerous and deadlock prone.
- An alternative approach, and primitive in Java is that of **intrinsic locking**
- This refers to identifying blocks of code that should be protected with a mutex lock.
- There is an implicit call to acquire a lock at code block entry and an implicit call to release the lock at code block exit.
- In Java this is provided by the `synchronized` keyword

```
synchronized (object) { // Critical Section // }
```

- This uses the object `object` as a mutex lock throughout the critical section and automatically acquires and releases the lock.
- Deadlocks are still possible (cf. Dining Philosophers) but not due to forgetting to release the locks.

The Producer / Consumer Problem

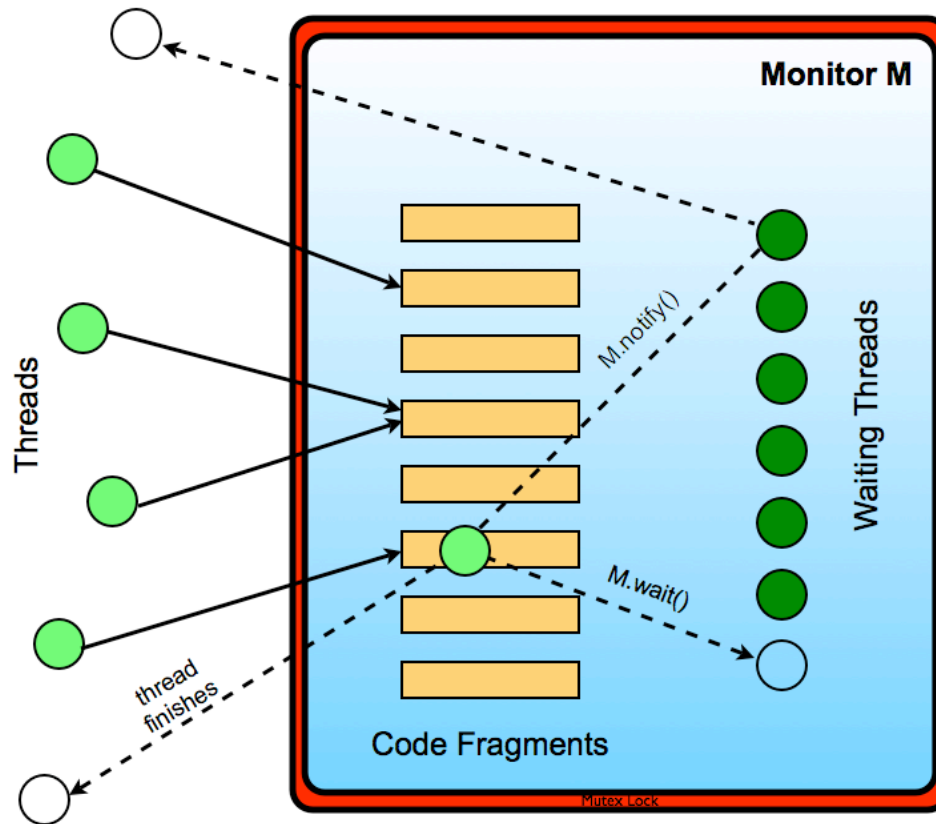
- Another classic concurrency problem is the Producer/Consumer Problem.
- One thread is trying to produce a sequence of messages in a shared buffer and the other thread is trying to consume these messages.
- We would like the sequence of messages to be produced and consumed in the correct order with no omissions.
- This problem lies right at the heart of shared variable communication.
- We can use locks to protect the read/writes in to the shared buffer but this is insufficient to guarantee correct behaviour.

```
class Buffer {  
    Object lock = new Object();  
    ArrayList<String> al = new ArrayList<String>();  
  
    void write(String msg){ synchronised (lock) { al.add(msg); } }  
  
    String read(){ synchronized (lock) { al.get(0); al.remove(0); } }  
}
```

What if the Consumer reads messages before the Producer has written them?
There is nothing to stop this happening!

Monitors

- Recall from Programming II that Monitors provide high-level communication primitives in Java in the form of `wait()` and `notify()` and `notifyAll()`



Producer/Consumer Monitor

- We can use monitors to solve the Producer/Consumer problem easily.
Let's do it for a fixed size circular buffer:

```
class Buffer {
    final int LIMIT = 10;
    int count = 0; int lo = 0, int hi = 0;
    Object lock = new Object();
    String[] buffer = new String[LIMIT];

    void waitHere() { try { wait(); } catch (InterruptedException e){ } }

    void write(String msg){ synchronised (lock) {
        while ( count == LIMIT ) waitHere(); // FULL BUFFER
        buffer[hi] = msg;
        hi = (hi + 1) % LIMIT;
        count = count + 1;
        if (count == 1) notify();
    }
}

String read(){ synchronised (lock) {
    while ( count == 0 ) waitHere(); // EMPTY BUFFER
    int val = buffer[lo];
    lo = (lo + 1) % LIMIT;
    count = count - 1;
    if (count == LIMIT -1) notify();
    return val;
}
}
```

The wait and notify methods

```
public final void wait() throws InterruptedException
```

- Calling `o.wait()` on an object `o` means
 - the calling thread must be in the monitor associated with `o`
 - the calling thread then blocks (or `InterruptedException` is thrown)
 - the calling thread joins a wait queue for the monitor `o`
 - the monitor lock on `o` is released

```
public final void notify()
```

- Calling `o.notify()` on an object `o` means
 - the calling thread must be in the monitor associated with `o`
 - one arbitrary thread from the wait queue for `o` is woken up
 - this woken thread does not hold the monitor lock and must re-acquire it
 - the calling thread retains the monitor lock on `o` and continues

java.util.concurrent

- Using the monitor primitives in Java is difficult - choosing the granularity of the synchronised blocks and when to call wait and notify requires understanding and experience.
- It is generally better to use built-in implementations of concurrency patterns wherever possible
- The library `java.util.concurrent` provides a number of useful correct and efficient implementations of common structures. These should be used in preference to the monitor commands.
- We'll have a look at a few of these to finish this lecture
- Firstly, the `BlockingQueue` and `BlockingDeque` interfaces (and the associated implementations of these) provide Queue and Double-ended Queue data structures
 - These are thread safe for concurrent access to the structure
 - They also allow threads to block on empty/full for reading and writing to the Queue (Deque).

java.util.concurrent.CountDownLatch

- The `CountDownLatch` class allows a number of threads to coordinate by creating a barrier which will block all threads arriving at it until a given number of threads have arrived.
- The latch is initialised with an integer value and then counts down each time a thread calls `await()` on the latch.
- Threads that have called `await()` will wait until the latch reaches a 0 value.
- Latches never increase their value but the counter can be decremented directly by calling `countDown()`

java.util.concurrent.CyclicBarrier

- If you wish to repeat the behaviour of a `CountDownLatch` then, because latches never increase their value, the only option would be to create a new latch object.
- However, the `CyclicBarrier` class provides a repeatable latch.
- The `CyclicBarrier` is constructed with an integer value similar to a latch and threads calling `await()` will wait until this many threads have already called `await()` on the barrier.
- The `CyclicBarrier` can also be optionally constructed with `Runnable` code to execute once the barrier has completed. This code is executed by the last thread to arrive at the barrier.
- Once all threads have arrived at the `CyclicBarrier` its value is reset to the initial integer value. This can also be reset manually using the `reset()` method.

java.util.concurrent.Phaser

- Phasers are a generalisation of `CyclicBarrier` that allow for dynamically changing the number of threads needed to arrive at each phase of the barrier.
- Threads register with a phaser and can then independently
 - Arrive at a phaser barrier without then waiting
 - Wait until the phaser advances to its next phase of barriering
- Phasers can also be arranged hierarchically in a tree-like fashion to provide finer control on the number of threads due to wait at any given phase.

Examples of usage of `CountDownLatches`, `CyclicBarriers` and `Phasers` are all available in the Java API documentation pages at

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/concurrent/package-summary.html>

Next Lecture

Message Passing Concurrency