COMP2212 Programming Language Concepts

Locks, monitors and atomicity
Locks

- a classic programming abstraction for concurrency
- may use hardware atomicity guarantees or OS functionality for implementation
- it is a low-level abstraction that is difficult to get right and does not always interact well with other programming features
Problems with locks

- Locks often lead to deadlocks!
- e.g. classic dining philosopher problem

There is one fork between each philosopher
A philosopher can think or eat
To eat, a philosopher must pick up two forks
   (this is some kind of strange Dutch-American spaghetti)
Assume that every philosopher tries to pick up their left fork first

Starvation (or deadlock)
Locks and deadlocks

- Deadlocks are extremely common in code with multiple locks
  - e.g. locks are picked up and released in the wrong order, locks are not released, ...
- Due to this, some applications with unpredictable/wild interleaving of actions (e.g. GUIs) are often written as a single thread!
(Intrinsic) Locks in Java

- synchronized blocks (the object lock serves as the lock)

```java
synchronized (lock) {
    // access or modify shared state guarded by lock
}
```

- synchronized methods are simply shorthand for a synchronised block that spans the entire method body, whose lock is the object on which the method is invoked

- at most one thread can hold a lock
Explicit Locks in Java

Introduced in Java 5.0

- **ReentrantLock**, an alternative to intrinsic locking with some advanced features
  - e.g. interrupt a thread that is waiting for a lock
  - attempt to acquire a lock without waiting for it indefinitely (trylock())
  - it is also dangerous: a lock is not automatically released as at the end of a synchronized block
Volatile variables in Java

❖ When a variable is declared as volatile
❖ the compiler and runtime know that this variable is shared between threads
❖ operations on volatile variables are not reordered by the compiler
❖ writes do not get lost in local caches, a read is guaranteed always to see the last write
Producer/Consumer problem

- Another classic concurrency problem
- Two threads, a producer and a consumer, communicating via a **bounded buffer**
  - if the producer thread has produced too much (the buffer is full), it must wait for the consumer to “consume” in order to free up a space
  - if the consumer thread has consumed all the data in the buffer, it must wait for the producer to produce a data and insert it into the buffer
Monitors

- Monitors combine the concept of lock with condition variables - queues of threads that are waiting on a lock
  - `wait(&lock)` - release the lock, go to sleep. Will automatically reacquire the lock when woken up
  - `signal()` - wakeup a single waiting thread (in Java `notify()`)  
  - `broadcast()` - wakeup all waiting threads (in Java `notifyAll()`)
Producer / Consumer problem with Monitors

This is pseudo code, there are no monitors in C
Java Monitors

• locking done by declaring certain methods synchronized
  • one condition variable per object
  • wait(), notify(), notifyAll() correspond to wait(&lock), signal() and broadcast()

• next slide: producer/consumer in Java
public class ProducerConsumer{
    static final int N = 100;
    static producer p = new producer();
    static consumer c = new consumer();
    static our_monitor mon = new our_monitor();

    public static void main(String args[]){
        p.start();
        c.start();
    }
}

static class producer extends Thread{
    public void run() {
        int item;
        while (true) {
            item = produce_item();
            mon.insert(item);
        }
        private int produce_item(){ /* ... */ }
    }
}

static class consumer extends Thread{
    public void run() {
        int item;
        while (true) {
            item = consume_item();
            mon.remove(item);
        }
        private void consume_item(){ /* ... */ }
    }
}

static class our_monitor{
    private int buffer[] = new int [N];
    private int count = 0, lo = 0, hi = 0;

    public synchronized void insert(int val) {
        while (count == N) go_to_sleep(); // buf full
        buffer(hi) = val;
        hi = (hi +1) % N;
        count = count + 1;
        if (count == 1) notify();
    }

    public synchronized int remove(){
        int val;
        while (count == 0) go_to_sleep(); // buf empty
        val = buffer[lo];
        lo = (lo+1)%N;
        count = count - 1;
        if (count == N-1) notify();
        return val;
    }

    private go_to_sleep(){
        try { wait(); }
        catch(InterruptedException exp){};
    }
}

Atomic data structures

- Atomic data structures are an example of **thread-safe** data structures
  - e.g. `AtomicInteger` class
  - some methods
    - `int get()`
    - `int set(int newValue)`
    - `int getAndAdd(int delta)`
  - The operations are atomic and visible — thus atomic data structures are like a “better” volatile
Composing atomic operations

- Using atomic data-structures is not a catch-all because atomicity might be at the wrong level of granularity.

- For example, a number of variables might have to be updated atomically, using a number of individually atomic operations does not guarantee global atomicity.

Fantastic book, highly recommended!