Chip Multiprocessors
COMP35112

Lecture 6 - Programming with Locks and Barriers

Graham Riley
Today’s Lecture

- In the COMP25111 lab, exercise 2, we used
  - locks (in the form of *synchronized* methods/blocks) to implement semaphores
  - semaphores to implement barriers
- So semaphores, locks and barriers are linked …

- Start with barriers because their use is so simple
- Then locks – and some issues connected with them
Barriers

- A simple idea: barrier is where a number of threads “meet up”
- When all threads reach it, they can all proceed
  - But threads before the last must wait until the last arrives
- Very natural when threads are used to implement data parallelism
  - Want the whole answer from this step before proceeding to the next step
- Would also use when data dependence limits loop parallelisation
- `java.util.concurrent.CyclicBarrier` – allows multiple use!
class Solver {
    final int N;
    final float[][] data;
    final CyclicBarrier barrier;

    // Worker here!

    public Solver(float[][] matrix) {
        data = matrix;
        N = matrix.length;
        barrier = new CyclicBarrier(N);
        for (int i = 0; i < N; i++)
            new Thread(new Worker(i)).start();
    }
class Worker implements Runnable {
    int myRow;
    Worker(int row) { myRow = row; }
    public void run() {
        while (!done()) {
            processRow(myRow);
            try { barrier.await();
                } catch (InterruptedException ex) { return; 
                } catch (BrokenBarrierException ex) { return; }
        }
    } }
Locks

- In Java, any object can be locked by:
  - Using it as the target in a `synchronized` block
  - Calling one of its class’s instance methods which is `synchronized`

- Only ONE thread can lock any particular object at a time

- Other threads requesting the lock (by either of the ways described above) have to wait until:
  - the `synchronized` block/method completes, or
  - the ONE thread executes `wait()` on the object
Why lock?

- To achieve “correctness” (to be defined!) …

- If two pieces of code take a lock on the same object, one should start and complete before the other starts

- This way the normal (sequential) meaning of the chunks of code is preserved

- In some ways best understood by considering code which fails to lock when updating shared variables
class BoundedBuffer {
private int [] buffer ;
private int inPtr, outPtr, count, numEls ;

public BoundedBuffer(int size) {
    buffer = new int[size] ;
    numEls = size ;
    inPtr = 0 ; outPtr = 0 ;   // not needed
    count = 0 ;               // = defaults!
}
public synchronized void deposit(int message) 
    throws InterruptedException {
    while (count == numEls) this.wait();
    buffer[inPtr] = message;
    inPtr = (inPtr + 1) % numEls;
    if (count++ == 0) this.notify();
    // notify() when nothing waiting is ignored!
}
public synchronized int extract() throws InterruptedException {
    while (count == 0) this.wait();
    int message = buffer[outPtr];
    outPtr = (outPtr + 1) % numEls;
    if (count-- == numEls) this.notify();
    return message;
}

} // end of class BoundedBuffer
Omit \textit{synchronized} in BB and …

- Could have two threads in \texttt{deposit()} both writing to the same element of \texttt{buffer} – one value will be lost
- Could then either increment \texttt{inPtr}
  - Once – whole call of \texttt{deposit()} lost
  - Twice – spurious (old) value apparently deposited
- Similarly for two calls of \texttt{extract()}
- Even problems between a call of \texttt{deposit()} and one of \texttt{extract()}, e.g. both change \texttt{count}
Sequential Consistency

- We say we have sequential consistency if (both):
  a) method calls should *appear to happen* in a one-at-a-time sequential order, and
  b) method calls should *appear* to take effect in program order (i.e. the order in which a thread performs its calls)

- This is behaviour the programmer will see:
  - i.e. interleaved method calls respecting per-thread orders

- This is a common (but not the only) interpretation of what we mean by “correct” in this context (another would be “linearizable”)

Dangers with Locks

- **Deadlock** – as soon as code requires more than one lock to be acquired you have this possibility

- **Mistaken use of conditions** (the “Lost WakeUp” Problem) – as in above BoundedBuffer code!
  - Thread a tries to extract from empty buffer, waits
  - Thread b tries to extract from empty buffer, waits
  - Thread c deposits in empty buffer – signals
  - Thread d deposits in buffer – non empty, so no signal
  - Thread a awoken by c extracts from buffer
  - Thread b is not awoken – even though value awaits!

- Fix (here) would be to use `notifyAll()` instead of `notify()`
Granularity

- How big a chunk of code which depends on obtaining a lock should you write?
- If too large (granularity is coarse-grained), available parallelism is limited
- If too small (granularity is fine-grained), you do a lot of work obtaining and releasing locks, and the program is harder to write
- It’s a trade-off: e.g. could lock a whole array to allow updates to some element(s), or you could lock only the individual element(s) being changed
Java and Locks

- Note that the two ways mentioned (in slide 6) of acquiring a lock in Java impose a nesting discipline just by the syntax!
- This is not always appropriate: e.g. multiple threads inserting items into a sorted linked-list using fine-grained locking
  - Want to lock the node currently under examination
  - Then want to lock the next node before unlocking the current one while holding the lock on this next node! (like climbing a rope hand-over-hand)
- So Java has a separate Lock interface and several implementing classes
java.util.concurrent.locks.Lock

Suggested idiom of use:

```java
Lock myLock = new ReentrantLock(); // e.g.

myLock.lock();
try {
    // access the resource protected by
    // myLock
} finally { myLock.unlock(); } 
```

This behaves just like the implicit monitor lock on Objects. Note: ‘reentrant’ means you don’t deadlock with yourself!
Hand-over-hand Locking

So here is a fragment of code to add to a sorted linked list of Node objects which include Locks and have lock() and unlock() methods acting on them.

```java
Node pred = head;
pred.lock();
try { Node curr = pred.next;
curr.lock();
try { while (curr.key < key) {
pred.unlock(); pred = curr;
curr = curr.next; curr.lock();
} /* ... /* curr.unlock() } }
```
Next Lecture

- Hardware support for locking: i.e. the mechanisms built into the hardware to assist in the implementation of locks