Comp 204: Computer Systems and Their Implementation

Lecture 8: The Producer-Consumer Problem

Today

- Synchronisation: The Producer-Consumer Problem
  - Definition
  - Java implementation
  - Issues

The Producer-Consumer Problem

- Another classic problem, in which a producer process and a consumer process communicate via a buffer
- Producer cycle:
  - produce item
  - deposit in buffer
- Consumer cycle
  - extract item from buffer
  - consume item
- May have many producers & consumers

First Attempt

```
class Buffer {
  private int v;

  public void insert(int x) {
    v = x;
  }

  public int remove() {
    return(v);
  }
}
```

Producer Class

```
class Producer extends Thread {
  private Buffer b;

  public Producer(Buffer buf) {
    b = buf;
  }

  public void run() {
    int m;
    ...
    b.insert(m);
    ...
  }
}
```

- Consumer is similar but calls **`n=b.remove()`**

Main Thread

```
public class ProdCon {

  public static void main(String args[]) {
    // Create buffer, producer & consumer
    Buffer b = new Buffer();
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);

    p.start(); c.start();
    ...
  }
}
```

Exercise

- What is wrong with this first attempt?

Answer

- There are a number of problems with our first attempt: they are all problems to do with synchronisation
- We have to ensure that:
  - producer cannot put items in buffer if it is full
  - consumer cannot extract items from buffer if it is empty
  - buffer is not accessed by two threads simultaneously

Second Attempt

```
class Buffer {
  private int v;
  private volatile boolean empty=true;

  public void insert(int x) {
    while (!empty)
      ; // null
    empty = false;
    v = x;
  }

  public int remove() {
    while (empty)
      ; // null
    empty = true;
    return(v);
  }
}
```

Explanation

- The ‘empty’ boolean is used to tell us if the buffer is empty or not
  - ‘volatile’ ensures the compiler will re-load the variable each time it is tested, rather than transferring it to a register
- A producer will wait while the buffer is full; a consumer will wait while it is empty
  - busy waiting or spinlock
- Busy waiting is inefficient
  - A thread or process does nothing useful while it has the CPU
  - Spinlocks are only useful in a multiprocessor situation when expected wait time is very short

Yielding

- It may be more efficient for a waiting thread or process to relinquish control of CPU

```
public void insert(int x) {
  while (!empty)
     Thread.yield();
  empty = false;
  v = x;
}
```

Exercise

- What is wrong with this second attempt?

Answer

- Two threads can still access the buffer simultaneously – there is nothing to stop this

- Note: The problem gets even worse when there are several consumers and producers

Third Attempt

```
class Buffer {
  private int v;
  private volatile boolean empty=true;

  public synchronized void insert(int x) {
    while (!empty)
      ; // null
    empty = false;
    v = x;
  }

  public synchronized int remove() {
    while (empty)
      ; // null
    empty = true;
    return(v);
  }
}
```

Explanation

- Every object has a lock associated with it
- When a thread calls a synchronized method, it gains exclusive control of the lock
- All other threads calling synchronized methods on that object must idle in an entry set
- When the thread with control exits the synchronized method, the lock is released
  - JVM can then select an arbitrary thread from entry set to be granted control of the lock

Exercise

- What is wrong with this third attempt?

Answer

- Deadlock can arise
- If the consumer tries to remove an item from an empty buffer, it will have to wait for the buffer to be filled by the producer
- But the buffer will not be filled as the consumer has the lock
- Similarly for the producer

Fourth Attempt

```
class Buffer {
  private int v;
  private volatile boolean empty=true;

  public synchronized void insert(int x) {
    while (!empty) {
      try {
        wait();
      }
      catch (InterruptedException e) {}
    }
    empty = false;
    v = x;
    notify();
  }

  // Similarly for remove()
}
```

Explanation

- The wait() call
  - releases the lock
  - moves the calling thread to the 'wait set'
- The notify() call
  - moves an arbitrary thread from the wait set back to the entry set
- Can use notifyAll() to move all waiting threads back to entry set

Entry and Wait Sets