COMP124

Computer Systems

22 | Producer-Consumer Problem | Semaphores

Dr Stuart Thomason

Semaphore Notation

- The semaphore mechanism provides two operations
  - wait() and signal()
  - Sometimes called P and V, test and inc, or up and down
  - We will implement our own boolean semaphore over the course of this lecture
  - Java uses different names for its semaphore methods but they do the same thing

- A thread must call wait() before it enters a critical region
  - This will block if the region (semaphore) is in use by another thread
  - Otherwise the thread obtains the lock and can enter the critical region
- The thread must call signal() just before it exits the critical region
  - This releases (unlocks) the semaphore
  - Any other blocked threads will be notified that the region is now safe to enter (only one thread will be successful if there are multiple threads waiting)

Producer-Consumer Problem

- The producer-consumer problem is a good way to illustrate how semaphores work
  - The producer and consumer are separate threads with different roles
  - Both the producer and consumer require access to a shared resource
  - There could be multiple producers and/or consumers accessing the same resource

- For example…
  - A secretary writes a letter (producer)
  - They put the letter into a tray (shared resource)
  - A manager takes the letter from the tray to sign it (consumer)

- In programming terms…
  - Threads are usually consumers or producers of some resource
  - The resource is a shared variable or object represented by a class (in Java)

Buffer – First Attempt

- We can model the problem and solution as two threads trying to access a buffer
  - Producer cannot insert if the buffer is full
  - Consumer cannot remove if the buffer is empty
  - Buffer cannot be accessed by multiple producers and consumers at the same time

- Consider the code for the buffer object

```
class Buffer {
    private int store;
    public void insert(int item) {
        store = item;
    }
    public int remove() {
        return store;
    }
}
```

Producer and Consumer

- Consider the code for the producer object

```
class Producer extends Thread {
    private Buffer buff;
    public Producer(Buffer b) {
        buff = b;
    }
    public void run() {
        int m;
        …
        buff.insert(m);
        …
    }
}
```

- The consumer object looks the same but it removes from the buffer instead

```
n = buff.remove();
```

Main Program

- The code for the main program creates instances of the buffer, producer and consumer

```
public class ProdCon {
    public static void main(String[] args) {
        Buffer b = new Buffer();
        Producer p = new Producer(b);
        Consumer c = new Consumer(b);
        p.start();
        c.start();
        …
    }
}
```

- None of this code has protection for the constraints mentioned earlier…
    - Producer doesn’t check to see if the buffer is full
    - Consumer doesn’t check to see if the buffer is empty
    - Buffer is not protected from being used by both threads at the same time

Buffer – Second Attempt

- Prevent threads trying to insert if the buffer is full (or remove if the buffer is empty)

```
class Buffer {
    private int store;
    private volatile boolean empty = true;
    public void insert(int item) {
        while(!empty);
        store = item;
        empty = false;
    }
    public int remove() {
        while(empty);
        empty = true;
        return store;
    }
}
```

- The volatile keyword prevents variables from being cached (always uses latest value)

Spinlock

- Before inserting anything, the producer waits for the buffer to be empty

```
while(!empty);
empty = false;
```

- Before removing anything, the consumer waits for the buffer to be full

```
while(empty);
empty = true;
```

- At first it might seem like these loops will never terminate
  - But the status variable is shared between multiple threads
  - Another thread could change its value and then the loop will terminate

- This concept is known as spinlock or busy waiting
  - It works well, but it is inefficient because it wastes CPU cycles
  - Should only be used when the expected wait time is very short

Yielding

- It could be more efficient for the waiting thread to give up control of the CPU

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

- The yield() method politely tells the JVM that the thread is willing to block (give up CPU)
  - Behaviour depends on JVM implementation
  - The JVM might ignore the request, so we still need to put it within a spinlock
  - The request is non-deterministic (might behave differently on different occasions)

Buffer – Third Attempt

- Prevent multiple threads from trying to insert and remove at the same time

```
class Buffer {
    private int store;
    private volatile boolean empty = true;
    public synchronized void insert(int item) {
        while(!empty) { Thread.yield(); }
        store = item;
        empty = false;
    }
    public synchronized int remove() {
        while(empty) { Thread.yield(); }
        empty = true;
        return store;
    }
}
```

- Note the American spelling of the synchronized keyword (with a zed)

Java Object Locks

- Every Java object has a lock associated with it
  - Locks are not shared across multiple instances of the same class
  - Each object instance has its own lock that works independently of other instances

- The synchronized keyword declares methods that contain critical regions
  - When a thread calls a synchronized method, it gets exclusive control of the lock
  - Other calls to the object's synchronized methods must wait (block) in an entry set
  - Note that non-synchronized methods can still be called without any issues

- When the thread with the lock exits the synchronized method, the lock is released
  - JVM will select an arbitrary thread from the entry set
  - That thread will be able to call the synchronized method it was waiting for

Wait and Notify

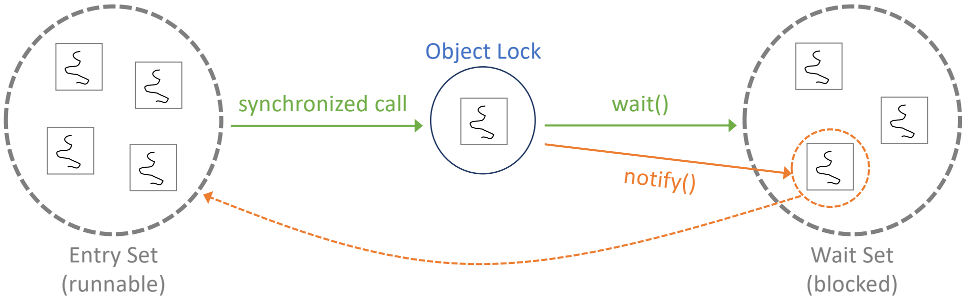
- Java provides methods to implement semaphores (but it uses notify instead of signal)

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

- These methods will always work regardless of the JVM or platform
- Note that exception handling is outside the scope of this module
  - It has to be included so the code will compile
  - But we are not interested in what it does (see COMP122 or online tutorials)

Entry and Wait Sets

- The wait() call suspends current thread and moves it to the wait set
- The notify() call moves an arbitrary thread from the wait set to the entry set (the choice of thread depends on the JVM implementation)



- Can also call notifyAll() to allow all threads in the wait set to compete for the chance to resume (winner is the one that calls a synchronized method first)

Java Semaphore Class

- The previous examples showed how we can implement our own semaphores
- Java provides a Semaphore class that we could use instead

```
import java.util.concurrent.*;
```

- Specify how many permits the semaphore allows (counting semaphore)

```
Semaphore sem = new Semaphore(1);
```

- When the acquire() method is called…
  - If the counter is 0, the thread will block (added to a wait queue)
  - If the counter is greater than 0, counter is decreased and the thread can continue
- When the release() method is called…
  - The counter is increased
  - The first blocked thread in the wait queue can acquire the permit
- Testing and setting the counter must be atomic operations to avoid race conditions

Buffer – Semaphore Version

- Update the code to use the Java semaphore mechanism

```
class Buffer {
    private int store;
    private Semaphore sem = new Semaphore(1);
    public synchronized void insert(int item) {
        try { sem.acquire(); } catch(InterruptedException e) {}
        store = item;
        sem.release();
    }
    public synchronized int remove() {
        try { sem.acquire(); } catch(InterruptedException e) {}
        int item = store;
        sem.release();
        return item;
    }
}
```