Programação Concorrente

Inverno 2018/2019

2ª Lista de Exercícios de preparação

1. Considere a classe **`UnsafeSemaphore`**, cuja implementação em *Java* se apresenta a seguir.

```
public class UnsafeSemaphore {
  private int maxPermits,
  permits;
  public UnsafeSemaphore(int initial, int maximum) {
    if (initial < 0 || initial > maximum) throw new
    IllegalArgumentException(); permits = initial; maxPermits = maximum;
  }
  public boolean tryAcquire(int
    acquires) { if (permits < acquires)
    return false; permits -= acquires;
    return true;
  }
  public void release(int releases) {
    if (permits + releases < permits || permits + releases > maxPermits)
      throw new IllegalArgumentException();
    permits += releases;
  }
}
```

   Esta implementação reflete a semântica de sincronização de um semáforo, contudo não é *thread-safe*. Implemente em *Java* ou em C#, sem utilizar *locks*, uma versão *thread-safe* deste sincronizador.

2. Considere a classe **`UnsafeCyclicBarrier`**, cuja implementação em *Java* se apresenta a seguir:

```
public class
  UnsafeCyclicBarrier {
  private final int partners;
  private int remaining, currentPhase;
  public UnsafeCyclicBarrier(int partners) {
    if (partners <= 0) throw new
    IllegalArgumentException(); this.partners =
    this.remaining = partners;
  }
  public void
    signalAndAwait() { int
    phase = currentPhase;
    if (remaining == 0) throw new IllegalStateException();
    if (--remaining == 0) {
      remaining = partners; currentPhase++;
    } else {
      while (phase == currentPhase) Thread.yield();
    }
  }
}
```

   Esta implementação reflete a semântica de sincronização de uma barreira cíclica (e.g., uma barreira que pode ser usada repetidamente para sincronizar o **mesmo grupo de *threads*)**, contudo não é *thread-safe*. Implemente em *Java* ou em C#, sem utilizar *locks*, uma versão *thread-safe* deste sincronizador.

3. Considere a classe `UnsafeSpinReentrantLock` , cuja implementação, em Java, é apresentada a seguir:

```
class UnsafeSpinReentrantLock {
      private Thread owner;
      private int count;
      public boolean tryLock() {
            if (owner == Thread.currentThread()) { count++; return true; }
            if (owner == null) { owner = Thread.currentThread(); return true; }
            return false;
      }
      public void lock() { while (!tryLock()) Thread.yield(); }
      public void unlock() {
            if (owner != Thread.currentThread()) throw new IllegalMonitorStateException();
            if (count == 0) owner = null; else count;
      }
}
```

Esta implementação reflete a semântica do sincronizador *reentrant lock* disponível em Java , contudo não é *threadsafe*. Usando técnicas de sincronização non blocking, implemente, em Java ou em C#, uma versão *threadsafe* deste sincronizador.

4. Considere a classe UnsafeSpinLazy<T> cuja implementação em C# é apresentada a seguir:

```
public class UnsafeSpinLazy<T> where T: class {
      private const int UNCREATED = 0, BEING_CREATED = 1, CREATED = 2;
      private int state = UNCREATED;
      private Func<T> factory;
      private T value;
      public UnsafeSpinLazy(Func<T> factory) { this.factory = factory; }
      public bool IsValueCreated { get { return state == CREATED; } }
      public T Value {
            get {
                SpinWait sw = new SpinWait();
                do {
                    if (state == CREATED) {
                        break;
                    } else if (state == UNCREATED) {
                        state = BEING_CREATED; value = factory(); state = CREATED; break;
                    }
                    sw.SpinOnce();
                } while (true);
                return value;
            }
      }
}
```

A implementação deste sincronizador, cuja semântica de sincronização é idêntica à do tipo `Lazy<T>` do . NET Framework, não é *threadsafe*. Sem utilizar locks , implemente uma versão *threadsafe* deste sincronizador.