Programación Concurrente

Práctica 4: Locks con granularidad fina y algoritmos lock-free

1. Analizar, para cada una de las implementaciones listas vistas en clase, que cambios deberían introducirse si las claves hash no fuesen únicas.

2. Considerar el siguiente monitor:

```
class Figura{
   int x = 0;
   int y = 0;
   int alto = 0;
   int ancho = 0;

   public synchronized void ajustarPosicion{
      x = algunX();
      y = algunY();
   }

   public synchronized void ajustarTamano{
      alto = algunAlto();
      ancho = algunAncho();
   }

  ...
}
```

   Asumir que `algunX` y `algunY` no utilizan `alto` y `ancho`, y que `algunAlto` y `algunAncho` no utilizan información sobre la posición.

   Dar una solución basada en locks para permitir que puedan ajustarse el tamaño y la posición de manera concurrente.

3. Considerar el siguiente monitor:

```
public class Recursos {
   Object[] recursos;
   int capacidad;

   public Recursos(int n){
      capacidad = n;
      recursos = new Object[capacidad];
   }

   public synchronized void assign(int pos, Object o){
      if (pos < capacidad)
         recursos[pos] = o;
   }

   public synchronized void swap(int i,int j){
      if (i < capacidad && j < capacidad){
         Object aux = recursos[i];
         recursos[i] = recursos[j];
         recursos[j] = aux;
      }
   }
}
```

   a) Dar una solución basada en locks que permita ejecutar concurrentemente operaciones `assign` y `swap` que operen sobre distintos recursos.

b) Indicar si su solución es libre de inanición.

4. Considerar la implementación de listas que utiliza sincronización optimista.

   a) Mostrar un escenario en el que un thread intenta indefinidamente eliminar a un nodo sin conseguirlo.

   b) Mostrar que la implementación no es correcta si se cambia el orden en que la operación `add` adquiere los locks sobre `pred` y `curr`.

5. Considere el siguiente monitor

```
public class Tabla {
   HashMap<Integer,Object> tabla = new HashMap<Integer,Object>();
   private final Lock lock = new ReentrantLock();

   private void actualizarEntrada (int i){
      lock.lock();
      try{
         Object v1 = tabla.get(i);
         Object v2 = ComputacionalmenteCostoso(v1);
         tabla.remove(i);
         tabla.put(i,v2);
      }
      finally{
         lock.unlock();
      }
   }
...
}
```

   a) Modificar la implementación de la operación `actualizarEntrada` utilizando la técnica de sincronización optimista para evitar que la estructura se encuentre bloqueada mientras se realiza el cómputo `ComputacionalmenteCostoso`.

   b) Indicar si la solución propuesta es libre de inanición. Justificar.

6. Considere la siguiente implementación de una cola no acotada.

```
class Stack {
   synchronized boolean isEmpty() { ... }
   synchronized Object pop() { ... }
   synchronized void push(Object o) { ... }
   ...
   }

class Queue {
   Stack in  = new Stack();
   Stack out  = new Stack();

   void enq(Object o){ in.push(o); }

   Object deq() {
      if(out.isEmpty()) {
         while(!in.isEmpty()) {
              out.push(in.pop());
         }
      }
      return out.pop();
   }
}
```

   a) Mostrar que la implementación no es correcta si las operaciones `enq` and `deq` pueden ser invocadas concurrentemente.

   b) Dar una solución correcta que utilice dos pilas y que permita la ejecución concurrente de `enq` and `deq` (al menos cuando la pila `out` no está vacía).

7. Considerar la implementación de una cola acotada que utiliza dos locks y la variable atómica `size`. Notar que el uso de `size` requiere la sincronización de los métodos `enq` y `deq`, que puede transformase en un cuello de botella que secuencializa a las operaciones. Modificar la solución considerando dos contadores, uno de incrementos y uno de decrementos de modo tal de minimizar la sincronización a los casos en los que es indispensable contar con el tamaño efectivo de la cola.

8. Implementar una pila acotada.

9. Considere la implementación de la pila lock-free.

   a) Mostrar que el problema ABA puede darse si no se cuenta con un garbage collector.

   b) Modificar la solución para evitar el problema ABA cuando no se cuenta con un garbage collector.

10. Implementar un contador que provea las operaciones inc, get and reset de manera lock-free. Indicar sin las operaciones son wait-free.

11. Dar una implementación libre de locks para las operaciones de la clase `Figura` en el Ejercicio 2.