Practica 5: Spinlocks
Cómputo Concurrente 2024-2

2024

Tiempo estimado de realizacion: 16 hrs
Objetivo: Comparación teórica y práctica de distintas implementaciones de Spinlocks.

Introducción

Los D.D, gracias a todo el apoyo de inteligencia, han podido completar diversos problemas de forma eficiente, pero ahora llega uno nuevo...

Un Candado es un objeto que garantiza la exclusión mutua, sin embargo, en la práctica ¿qué hacemos si no pudimos obtener el candado? Hay dos opciones:

1. Esperamos activamente
2. Le pedimos al S.O que agende otro hilo

Un Spinlock es un objeto que permite que los hilos esperen activamente hasta lograr entrar a la sección crítica.

Hasta ahora hemos visto implementaciones de Candados que tienen una relevancia teórica, sin embargo, en la práctica necesitamos eficiencia y lidiar con los hilos que no tienen éxito: lidiar con la contención.

Contención: ocurre cuando múltiples hilos intentan acceder al candado al mismo tiempo, una contención alta implica que son muchos hilos al mismo tiempo y una contención baja implica que solo son algunos.

Todos los candados utilizan la instrucción *var.testAndSet(true)* (*getAndSet()*), esta instrucción reemplaza el valor de la variable var con $true$ y devuelve el anterior. $BackoffLock()$, además, utiliza una ventana de tiempo para aliviar la contención.

$MCSLock()$ y $CLHLock()$ incorporan una cola para, además de determinar el hilo que puede acceder a la seccion critica, formar a los demás y así distribuir aún más la contención.

Especificaciones

El equipo de desarrollo te ha brindado los siguientes candados, deberás implementar cada uno de ellos.

TAS

```
1  public class TASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (state.getAndSet(true)) {}
5    }
6    public void unlock() {
7      state.set(false);
8    }
9  }
```

TTAS

```
1  public class TTASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (true) {
5        while (state.get()) {};
6        if (!state.getAndSet(true))
7          return;
8      }
9    }
10   public void unlock() {
11     state.set(false);
12   }
13 }
```

BackOffLock

```
1  public class Backoff {
2    final int minDelay, maxDelay;
3    int limit;
4    final Random random;
5    public Backoff(int min, int max) {
6      minDelay = min;
7      maxDelay = max;
8      limit = minDelay;
9      random = new Random();
10   }
11   public void backoff() throws InterruptedException {
12     int delay = random.nextInt(limit);
13     limit = Math.min(maxDelay, 2 * limit);
14     Thread.sleep(delay);
15   }
16 }
```

```
1  public class BackoffLock implements Lock {
2    private AtomicBoolean state = new AtomicBoolean(false);
3    private static final int MIN_DELAY = ...;
4    private static final int MAX_DELAY = ...;
5    public void lock() {
6      Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
7      while (true) {
8        while (state.get()) {};
9        if (!state.getAndSet(true)) {
10         return;
11       } else {
12         backoff.backoff();
13       }
14     }
15   }
16   public void unlock() {
17     state.set(false);
18   }
19   ...
20 }
```

CLH

```
1  public class CLHLock implements Lock {
2    AtomicReference<Qnode> tail;
3    ThreadLocal<QNode> myPred;
4    ThreadLocal<QNode> myNode;
5    public CLHLock() {
6    tail = new AtomicReference<QNode>(null);
7      myNode = new ThreadLocal<QNode>() {
8        protected QNode initialValue() {
9          return new QNode();
10       }
11     };
12     myPred = new ThreadLocal<QNode>() {
13       protected QNode initialValue() {
14         return null;
15       }
16     };
17   }
18   ...
19 }
```

```
20     public void lock() {
21       QNode qnode = myNode.get();
22       qnode.locked = true;
23       QNode pred = tail.getAndSet(qnode);
24       myPred.set(pred);
25       while (pred.locked) {}
26     }
27     public void unlock() {
28       QNode qnode = myNode.get();
29       qnode.locked = false;
30       myNode.set(myPred.get());
31     }
32   }
```

Qnode tiene por unico elemento un booleano volatil, llamado looked.

MCS

```
1  public class MCSLock implements Lock {
2    AtomicReference<QNode> tail;
3    ThreadLocal<QNode> myNode;
4    public MCSLock() {
5      queue = new AtomicReference<QNode>(null);
6      myNode = new ThreadLocal<QNode>() {
7        protected QNode initialValue() {
8          return new QNode();
9        }
10     };
11   }
12   ...
13   class QNode {
14     boolean locked = false;
15     QNode  next = null;
16   }
17 }
```

```
18   public void lock() {
19     QNode qnode = myNode.get();
20     QNode pred = tail.getAndSet(qnode);
21     if (pred != null) {
22       qnode.locked = true;
23       pred.next = qnode;
24       // wait until predecessor gives up the lock
25       while (qnode.locked) {}
26     }
27   }
28   public void unlock() {
29     QNode qnode = myNode.get();
30     if (qnode.next == null) {
31       if (tail.compareAndSet(qnode, null))
32         return;
33       // wait until predecessor fills in its next field
34       while (qnode.next == null) {}
35     }
36     qnode.next.locked = false;
37     qnode.next = null;
38   }
```

ALock

```
1  public class ALock implements Lock {
2    ThreadLocal<Integer> mySlotIndex = new ThreadLocal<Integer> (){
3      protected Integer initialValue() {
4        return 0;
5      }
6    };
7    AtomicInteger tail;
8    volatile boolean[] flag;
9    int size;
10   public ALock(int capacity) {
11     size = capacity;
12     tail = new AtomicInteger(0);
13     flag = new boolean[capacity];
14     flag[0] = true;
15   }
16   public void lock() {
17     int slot = tail.getAndIncrement() % size;
18     mySlotIndex.set(slot);
19     while (! flag[slot]) {};
20   }
21   public void unlock() {
22     int slot = mySlotIndex.get();
23     flag[slot] = false;
24     flag[(slot + 1) % size] = true;
25   }
26 }
```

Tablas y Gráficas

En esta parte, cada integrante debe hacerlo, es decir, con su computadora hacer todas las pruebas descritas.

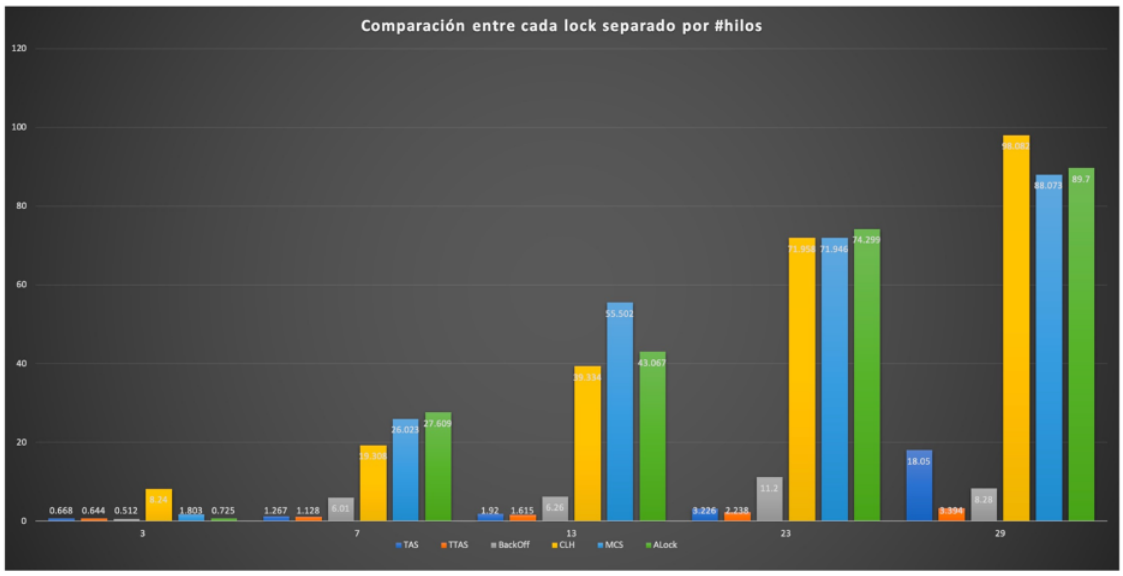
Primero escribe las especificaciones de tu computadora o donde ejecutaste las pruebas, las especificaciones requeridas son las siguientes (Si utilizas una VM, pon las especificaciones de esta):

- CPU (Frecuencia, Nucleo, Hilos)
- RAM (Cantidad, Modulos, Frecuencia)
- MB
- Cualquier otra que consideren necesaria

Una vez implementados, realiza pruebas de tiempo de ejecución usando distintos número de Hilos (2,3,7,15,21,30,50), de ahí realiza una tabla como la siguiente:

Hilos	TAS	TTAS	BackOff	CLH
3	6.68	6.44	5.12	8.24
7	12.67	11.28	6.01	193.08

Posteriormente realiza una gráfica como la siguiente:



Una vez hecho esto, explica el porqué de los tiempos de ejecución, compara el resultado con el de tus compañeros, y si la diferencia entre computadoras es muy grande explica el porqué.

Consideraciones

Para cada test, ejecutalo individualmente, no los ejecutes todos en serie, esto puede afectar en tus resultados.

Cuestionario

1. ¿Para que sirve el método Yield? (yield())
2. ¿Qué es un atributo átomico? (En la biblioteca atomic de Java)
3. Ventajas de usar atributos atomicos
4. Desventajas de usar atributos atómicos
5. ¿Que locks cumplen con la propiedad de Justicia?
6. ¿Qué locks cumplen con la propiedad libre de Hambruna (starvation-free)?
7. ¿Cuál es la implementación más eficiente? ¿Porqué crees que es así?
8. Por último, describe un problema en el que se pueda utilizar un candado visto en esta práctica
9. Escribe lo aprendido en esta practica asi como diferencias respecto a las anteriores.

Extra

- Toma las temperaturas de tu Computadora y comparalas entre cada algoritmo empleado, puedes usar la cantidad de Hilos Maxima para esto.
- Implementa TAS o TTAS usando solo variables volatiles, analiza el por que falla y explicalo.

Por ultimo...