Universidad

Nacional Autónoma de México

Facultad de Ciencias

Tarea 5
Operaciones Primitivas y Universalidad del Consenso

Integrantes:
Yonathan Berith Jaramillo Ramírez. 419004640
Diego Arturo Velázquez Trejo. 317227257
Marco Antonio Garcia Arce, 421014615

Profesor: Glide Valeria Rodriguez Jimenez
Ayudantes: Hermilio Cortez Gonzalez
Luis Angel Leyva Castillo
Rogelio ALcantar Arenas

14 Octubre, 2023

Computo concurrente

Intrucciones: La tarea se entrega y se resuelve en equipos de máximo 3 integrantes (1 a 3 personas). La tarea se entrega virtualmente, en el formato que más les convenga

siempre y cuando sea legible.

No habrán reposiciones de tareas. Las tareas se pueden entregar con máximo 2 días de retraso para ser evaluadas sobre 8.

1. **Ejercicio 54 de [HS08].** *Hint: Es similar al ejercicio 57 que resolvimos en clase. No es necesario demostrarlo, solo justifica tu respuesta.*

   > **Exercise 54.** Suppose we augment the FIFO Queue class with a `peek()` method that returns but does not remove the first element in the queue. Show that the augmented queue has infinite consensus number.

   El código proporcionado en el libro:

Consensus

```
public interface Consensus<T> {
    T decide(T value);
}
```

Consensus Protocol

```
public abstract class ConsensusProtocol<T> implements Consens
    protected T[] proposed = (T[]) new Object[N];
    // announce my input value to the other threads
    void propose(T value) {
        proposed[ThreadID.get()] = value;
    }
    // figure out which thread was first
    abstract public T decide(T value);
}
```

Queue Consensus

```
public class QueueConsensus<T> extends ConsensusProtocol<T> {
    private static final int WIN = 0; // first thread
    private static final int LOSE = 1; // second thread
    Queue queue;

    // initialize queue with two items
```

```
    public QueueConsensus() {
        queue = new Queue();
        queue.enq(WIN);
        queue.enq(LOSE);
    }

    // figure out which thread was first
    public T decide(T value) {
        propose(value);
        int status = queue.deq();
        int i = ThreadID.get();
        if (status == WIN)
            return proposed[i];
        else
            return proposed[i-1];
    }
}
```

Respuesta

Supongamos que agregamos a la clase `FIFO Queue` un método `peek()` que devuelve, pero no elimina, el primer elemento de la cola. El objetivo es demostrar que la cola mejorada tiene un número de consenso infinito.

Implementación Original de QueueConsensus

La clase `QueueConsensus` en su forma actual opera de la siguiente manera:

- El método `propose()` establece el valor propuesto para el hilo actual.
- El método `decide()` utiliza la cola para determinar qué hilo llega primero (el que desencola el valor `WIN`) y devuelve el valor propuesto correspondiente.

Mejora en la Implementación

Para incluir el método `peek()`, supongamos que la clase `FIFO Queue` originalmente se ve algo así:

```
public class Queue {
    private Node front;
    private Node rear;
```

```
    public void enq(int value) {
        // Implementaci n de encolar
    }

    public int deq() {
        // Implementaci n de desencolar
    }
}
```

Y ahora la clase `FIFO Queue` se modificaría para añadir el método `peek()` como sigue:

```
public class Queue {
    private Node front;
    private Node rear;

    public void enq(int value) {
        // Implementaci n de encolar
    }

    public int deq() {
        // Implementaci n de desencolar
    }

    public int peek() {
        // Devuelve pero no elimina el primer elemento
        return front.element;
    }
}
```

Nuevo Protocolo de Consenso

Suponiendo que hemos añadido `peek()`, podemos modificar `QueueConsensus` de la siguiente manera:

```
public T decide(T value) {
    propose(value);
    int i = ThreadID.get();
    queue.enq(i);
    int winner = queue.peek();
```

```
    return proposed[winner];
}
```

Justificación

Este nuevo método permite a cualquier número de hilos alcanzar un consenso:

- Cada hilo anuncia su valor propuesto.
- Luego, cada hilo se coloca en la cola.
- El método `peek()` permite que cada hilo observe el primer elemento en la cola (el ganador) sin alterarlo.
- Todos los hilos devuelven el valor propuesto por el ganador, logrando así el consenso.

Para entender completamente por qué la adición del método `peek()` a la clase `FIFO Queue` resulta en un número de consenso infinito, es vital examinar cómo interactúan los hilos con la cola y cómo eso afecta la decisión de consenso.

(a) Comportamiento de `peek()` El método `peek()` es crucial porque permite a todos los hilos acceder al primer elemento de la cola sin desencolarlo. En otras palabras, una vez que un hilo ha sido encolado primero, cualquier otro hilo que utilice `peek()` obtendrá el índice de este hilo, sin cambiar el estado de la cola.

(b) Protocolo de Anuncio Cada hilo anuncia su valor propuesto escribiéndolo en una matriz `announce[]` compartida. Esta operación se realiza en el método `propose()`, que se ejecuta antes de que el hilo encole su índice en la cola. Esta secuencia garantiza que cuando un hilo consulta el índice del hilo ganador con `peek()`, el valor propuesto correspondiente ya ha sido almacenado.

(c) Indeterminismo Limitado La clave para lograr un número de consenso infinito es que una vez que un hilo ha sido encolado primero, su posición se vuelve inmutable para todos los hilos que llegan después. Como cada hilo utiliza `peek()` para acceder al valor del primer hilo en la cola, la decisión es coherente y unánime entre todos los hilos.

Debido a la naturaleza de `peek()`, que permite la observación sin mutación, y debido a la secuencia en la que los hilos anuncian y encolan sus propuestas, la cola con `peek()` añadido permite un consenso entre un número infinito de hilos. Este consenso es tanto unívoco como uniforme, cumpliendo con los requisitos para tener un número de consenso infinito.

2. **Ejercicio 53 de [HS08].** Primero describe el protocolo de consenso para una Stack (similar al de la Queue) y argumenta (a) porqué lo resuelve para dos hilos y (b) porqué no lo resuelve para tres hilos.

   Hint: Es similar a probar que una Queue tiene un numero de consenso de exactamente dos (lo vimos en clase y está en el libro).

   Exercise 53. The Stack class provides two methods: `push(x)` pushes a value onto the top of the stack, and `pop()` removes and returns the most recently pushed value. Prove that the Stack class has consensus number *exactly two*.

Respuesta

Por contradicción. Asumimos que tenemos un protocolo de consenso para A, B y C. Po el lema 5.1.3, debe de haber un estado critico s. Sin pérdida de generalidad, asumimos que el próximo movimiento de A lleva al protocolo a un estado 0-valente, y el próximo movimiento de B lleva a un estado 1-valente. También sabemos que estas llamadas deben ser no conmutativas; esto implica que deben ser llamadas al mismo objeto. A continuación, sabemos que estas llamadas no pueden hacerse a registros, ya que los registros tienen un número de consenso de 1. Por lo tanto, estas llamadas deben hacerse al mismo objeto de pila.
Procedemos a revisar los 3 casos de ejecución.
A y B realizan pop(). Sea s' el estado del protocolo si A primero realiza pop() y despues B, y sea s" si es primero B y despues A. s' es 0-valente y s" es 1-valente, pero para C son exactamente lo mismo ya que los mismos objetos fueron quitados de la pila, por lo tanto C decide ambos valores.
A realiza push(x) y B realiza pop(). Si A realiza push(x) y despues B realiza pop(), que seria x, y despues A realiza pop(), con s' siendo el estado del protocolo de esta ejecución. s' no tendria diferencia para C con respecto a s" que seria el estado de protocolo si B realiza pop(), A realiza push(x) y despues pop(). C no sabe distinguirlos ya que el resultado es el mismo, por lo cual otra contradicción existe.
A y B realizan push(). Sea s' el estado del protocolo donde A realiza push(x), B realiza push(y), A realiza pop() dando y, B realiza pop() dando x. Sea s" el estado donde B realiza push(y), A realiza push(x), A realiza pop() dando x, B realiza pop(), dando y. C de nuevo no puede distinguir entre s' y s" ya que la pila tiene el mismo resultado, por lo cual esta esta contradicción.

3. De acuerdo a las clases y al libro [HS08], explica brevemente qué implica la Construcción Universal y porqué es relevante.

Construcción Universal

La Construcción Universal es un concepto en la computación concurrente que se refiere a la habilidad de implementar cualquier objeto concurrente de manera *wait-free* (sin esperas) utilizando una cantidad suficiente de ciertos objetos y registros de lectura-escritura.

Relevancia

- **Poder Computacional**: Una clase es universal en un sistema de n hilos si y solo si tiene un número de consenso mayor o igual a n. Este concepto es especialmente relevante para determinar la potencia computacional de una arquitectura de máquina o un lenguaje de programación.
- **Implementación de Objetos Concurrentes**: La construcción universal permite crear una implementación *wait-free* de cualquier objeto concurrente, lo cual es fundamental para la eficiencia y la robustez de los sistemas concurrentes.
- **Evitar Errores Conceptuales**: Entender la construcción universal y sus implicancias ayuda a evitar el error de intentar resolver problemas que son intrínsecamente irresolubles en este contexto.

4. De acuerdo a la Construcción Universal Wait-free (código a continuación). Si se ejecuta para $n = 5$ procesos (A con $id = 1$, B con $id = 2$, C con $id = 3$, D con $id = 4$ y F con $id = 5$) y en LOG aún no existen ningún nodo. Supón que los 4 hilos (de A a D) ejecutan `apply()` hasta la línea 9 y se quedan dormidos (se detienen). Describe el estado de LOG si después el hilo F ejecuta el método `apply()` sin detenerse. Justifica tu respuesta.

```
public class Universal {
    //The array announce was added to coordinate helping
    private Node[] announce;
    private Node[] head;
    private Node tail = new Node();
    tail.seq = 1;
    for (int j=0; j < n; j++) {
        head[j] = tail;
```

```
        announce[j] = tail;
    }
    public Response apply(Invoc invoc) {
        int i = ThreadID.get();
        announce[i] = new Node(invoc);
        head[i] = Node.max(head);
        while (announce[i].seq == 0) {
            Node before = head[i];
            Node help = announce[before.seq + 1 % n];
            if (help.seq == 0)
                prefer = help;
            else
                prefer = announce[i];
            after = before.decideNext.decide(prefer);
            before.next = after;
            after.seq = before.seq + 1;
            head[i] = after;
        }
        SeqObject MyObject = new SeqObject();
        current = tail.next;
        while (current != announce[i]) {
            MyObject.apply(current.invoc);
            current = current.next;
        }
        head[i] = announce[i];
        return MyObject.apply(current.invoc);
    }
}
```

Figure 6.6 The wait-free universal algorithm.

Observamos el algoritmo de construcción universal Wait-Free con los 5 procesos en los siguientes pasos:

1 Estado inicial

- `announce` es un array que mantiene el último nodo anunciado por cada hilo.

- `head` es un array que mantiene el último nodo confirmado por cada hilo.
- `tail` es el nodo inicial de la lista enlazada, con `seq` inicializado a 1.

2 Primero A, B, C y D llegan hasta la línea 9 y se detienen

Cuando A a D ejecutan el método `apply()` hasta la línea 9:

1. Cada uno crea un nodo y lo anuncia en el array `announce`.
2. Cada uno actualiza su `head` al máximo `head` existente.
3. Se quedan "dormidos".

Ahora, todos tienen `announce[i].seq = 0` y establecieron su respectivo `head[i]`. Como todos están dormidos, `head[i]` para todos sigue siendo `tail`.

3 Segundo F ejecuta `apply()` sin detenerse

Cuando F entra en acción, sucede lo siguiente:

1. F crea un nuevo nodo y lo anuncia.
2. F también actualiza su `head` al máximo `head` existente (que sigue siendo `tail` en este caso).
3. F entra en el `while` para determinar su `seq`.

En cada iteración del `while`:

- F establece `before = head[5]` (inicialmente `tail`).
- Como A a D están dormidos, no hay otros nodos con un `seq` establecido, por lo que F siempre tomará su propio nodo anunciado (`announce[5]`) como el nodo `prefer`.
- F actualiza `before.next` con `after` y establece `after.seq = before.seq + 1`.
- `head[5]` se actualiza a `after`.

Dado que F es el único hilo activo, terminará saliendo del `while` una vez que su `announce[5].seq` se haya establecido. Ahora bien, el log `LOG` tendría lo siguiente:

- Nodo de F con `seq = 2` (el `tail` original tenía `seq = 1`).

El estado de `LOG` será una lista enlazada donde `tail` (con `seq = 1`) apunta al nodo de F con `seq = 2`. Los demás nodos anunciados por A, B, C, y D no serán parte de `LOG` porque su `seq` sigue siendo 0.

References

[HS08] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.