Universidad Nacional Autónoma de México
Facultad de Ciencias

Computación Concurrente

Tarea 5 (Parte 1)

Bonilla Reyes Dafne - 319089660
García Ponce José Camilo - 319210536
Juárez Ubaldo Juan Aurelio - 421095568

Consenso y primitivas de sincronización

Ejercicio 1

Supón que tenemos una implementación de una cola FIFO modificada, con métodos *enq(x)* y *peek()* (Fig. 1), en la cual cada nodo de la cola encapsula el valor del elemento (Un entero o un booleano, etc) y un ID que inicialmente es -1.

El método *peek()* consiste en devolver el primer elemento de la Cola, sin embargo, a diferencia de un método *deq()*, no lo elimina de la Cola.

- *Muestra que esta cola modificada tiene un número de consenso infinito. Hint: Crea un protocolo de consenso para un número infinito de hilos utilizando esta cola modificada, como pseudocódigo. Considera el protocolo de consenso de una cola visto en clase, analiza qué sucede si en vez de utilizar deq() ahora utilizamos peek().*

```
   public int peek() {
1:   myID = obtiene su ID;
2:   if (head.compareAndSet(-1, myID)){
3:     return head.get(); }        ▷ Regresa el primer nodo de la cola sin eliminarlo
4:   else {
5:     return null; }
6: }
```

Figure 1: Método *peek*() de la Cola modificada

Para este ejercicio, proponemos el siguiente código:

```
class PConsensus extends ConsensusProtocol {
    private ColaPeek c;

    public PConsensus() {
        cola = new ColaPeek();
        cola.enq(id:-1, valor:null);
    }

    public Object decide(Object value) {
        propose(value);
```

```
        if (cola.peek() != null) {
            int i = ThreadID.get();
            return proposed[i];
        } else {
            Nodo nodo = cola.get();
            int i = nodo.getID();
            return proposed[i];
        }
    }
}
```

En general, para este código hacemos las siguientes suposiciones:

- `get()` de `ColaPeek()` regresa el nodo de la cabeza (`head.get()`).
- `peek()` regresa algún objeto de tipo `Nodo` o `null`.
- `proposed` es un arreglo y tiene escrituras y lecturas atómicas.

Ahora veamos que este algoritmo resuelve el consenso para 2 hilos A y B cualesquiera, para ello tendremos lo siguiente:

1. El hilo A es el primero en entrar y proponer su valor, seguido por el hilo B, que también propone su valor.
2. El hilo A realiza la operación `peek()` sobre la cola. Como es el primero en hacerlo, el valor en la cabeza de la cola es `id = -1` (ya que se hizo un `enq()` de un nodo con `id = -1` y un valor `null` en el constructor de la cola), por lo que la operación `compareAndSet()` regresa `true` y modifica el `id` que tiene la cabeza de la cola, poniendo el `id` del hilo A. Esto significa que el hilo A ha ganado la carrera para decidir. Entonces, el hilo A devuelve su valor propuesto como el valor de consenso.
3. El hilo B también intenta realizar un `peek()`, pero esta vez el valor en la cabeza de la cola ya no es `id = -1`, sino que contiene el `id` del hilo A, esto debido a que `compareAndSet()` es una operación atómica, por lo cual no puede haber inconsistencias en los valores. Por lo tanto, B no puede tomar la decisión directamente. En lugar de eso, B ejecuta un `get()` para obtener el nodo de la cabeza de la cola (aunque no lo elimina), y obtiene el `id` del hilo A.
4. Dado que B obtiene el `id` de A, este hilo devolverá el valor propuesto por A como el valor consensuado.

De esta manera, se asegura que ambos hilos, A y B, estén de acuerdo en un valor, resolviendo el consenso para dos hilos. Notamos que como `compareAndSet` es una operación atómica, solo el primer hilo que realice el `compareAndSet` será el que modifique el valor del `id` de la cabeza de la cola, por lo tanto, todos los demás hilos van a saber que ese hilo fue el primero y ganó. Por lo tanto si ambos hilos regresan el valor que propusieron, entonces significa el método `peek()` les regresa al distinto de `null` a ambos hilos, por lo cual tenemos que ambos hilos vieron que la cabeza tenia como `id` a -1, lo cual no puede pasar ya que para ver el `id` de la cabeza se usa `compareAndSet`. Y si ambos hilos regresan el valor que el otro hilo propuso, significa que el método `peek()` les regreso `null` a ambos hilos, por lo cual ambos hilos vieron que la cabeza tenia como `id` al `id` del otro hilo, pero esto no puede pasar ya que `compareAndSet` solo cambia una vez el `id` que esta en la cabeza.

A continuación, veamos la generalización para infinitos hilos cualesquiera, para ello, notemos que para n hilos, va a pasar algo muy similar:

El primer hilo que realice el `compareAndSet()` del `peek()` será el que ponga su `id` en la cabeza de la cola. Esto debido a que `compareAndSet()` es una operación atómica, lo cual nos garantiza que no existan inconsistencias al revisar el `id` de la cabeza de la cola.

Por lo tanto, el primer hilo que realice el `compareAndSet()` será el que gane y su valor será el decidido. Al poner su `id` en la cabeza, los demás hilos pueden saber qué valor eligió al revisar el arreglo de valores propuesto en la posición del `id` del primer hilo.

De esta forma, podemos resolver el consenso para una cantidad infinita de hilos, por ejemplo, siguiendo el ejemplo de 2 hilos, si llegara un hilo C entonces haría el `peek()` obteniendo `null` y, por lo tanto, tendría que revisar el valor de `proposed` en la posición del hilo A (que lo obtuvo usando el método `get()` para obtener la cabeza y luego ver que `id` tiene la cabeza) y regresando lo que eligió. De igual manera, si llega otro hilo D, y así sucesivamente para cualquier otro hilo que intente realizar `decide()`.

De esta manera, si a dos hilos distintos regresan su valor propuesto, entonces significa el método `peek()` les regresa al distinto de `null`, por lo cual tenemos que ambos hilos vieron que la cabeza tenia como `id` a -1, lo cual no puede pasar, ya que `peek()` usa `compareAndSet()` para revisar el `id` que esta en la cabeza y `compareAndSet()` es una operación atómica.

La condición de validación se sigue de la observación de que el hilo que realiza `peek()` y obtiene algo diferente a `null` (cambiar el `id` de la cabeza al realizar el `compareAndSet()`), guardó su valor propuesto en el arreglo `proposed` antes de que ningún otro hilo haya cambiado el `id` de la cabeza (realizar el `compareAndSet()`).

Ejercicio 2

Una pila tiene dos métodos, el método *push(x)* añade a x al inicio de la pila y el método *pop() : x* elimina el primer elemento x al inicio de la pila (mantiene *LIFO*). Muestra que una pila tiene un número de consenso exacto igual a 2.

Hint: Es similar a probar que una Queue tiene un número de consenso de exactamente dos (lo vimos en clase y está en el libro). Debes mostrar dos cosas: Un protocolo para dos hilos y argumentar por qué no es posible que exista un protocolo para tres hilos.
Para este ejercicio tendremos que revisar los siguientes puntos:

- La pila tiene consenso de al menos 2.

  Para esto, usemos el siguiente protocolo de consenso para 2 hilos:

```
public class StackConsensus<T> extends ConsensusProtocol<T> {
    private static final int WIN = 0;  // first thread
    private static final int LOSE = 1; // second thread
    Stack stack;

    // initialize stack with two items
    public StackConsensus() {
        stack = new Stack();
        stack.push(LOSE);
        stack.push(WIN);
    }

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

Mostramos un protocolo de consenso para 2 hilos, usando una sola pila *LIFO* (supondremos que la pila es *wait-free*). Aquí la pila guarda enteros y se inicializa agregando el valor `LOSE` y después el valor `WIN`.

El método `decide` primero llama a `propose(value)` el cual guarda `value` en `proposed` que es un arreglo compartido de `T`s. Después, saca un elemento de la pila:

- Si el elemento es `WIN`, entonces el hilo que realizó la llamada a `decide` fue el primero (ya que sacó el último valor agregado a la pila) y decide su valor propuesto
- Si el elemento es `LOSE`, entonces otro hilo ya había pasado (ya que sacó el primer valor agregado a la pila), por lo que el hilo regresa el valor del otro hilo (el que pasó primero), con ayuda del arreglo `proposed` y sabiendo que solo son dos hilos.

Notemos que lo siguiente no puede pasar:

- Si cada hilo regresa su propio valor, entonces significaría que ambos hilos sacaron el elemento `WIN`, lo cual viola que la pila sea *LIFO*, ya que solo se agregó un valor `WIN` a la pila y fue el último en agregarse.
- Si cada hilo regresa el valor del otro hilo, entonces significa que sacaron el elemento `LOSE`, lo cual es otra violación a como funciona la pila, ya que solo se agregó un valor `LOSE` a la pila y fue el primero en agregarse.

Tambien observemos que este protocolo es *wait-free*, ya que no tenemos ciclos en `decide()` y la pila es *wait-free*.

La condición de validación se sigue de la observación de que el hilo que saca `WIN` guardó su valor propuesto en el arreglo `proposed` antes de que ningún elemento haya sido sacado de la pila.

- La pila tiene consenso de exactamente 2 (no sirve para 3 hilos).
  Supongamos, por contradicción que existe un protocolo de consenso para 3 hilos **A**, **B** y **C**, utilizando una pila **LIFO**. Por el lema 5.1.3, sabemos que tiene un estado critico, es decir, un estado donde s es bivalente, pero cualquier movimiento de un hilo causa que el sistema entre en un estado univalente. Sin perdida de generalidad, digamos que si **A** se mueve entonces el protocolo entra a un estado 0-valente y si **B** se mueve entonces el protocolo entra a un estado 0-valente. También sabemos que las operaciones no se pueden conmutar, entonces van a realizar llamadas de métodos del mismo objeto, ademas no pueden hacer escrituras o lecturas a registros (tiene numero de consenso igual a 1), por lo tanto van a hacer llamadas a métodos de una pila compartida. Vamos a realizar un análisis de casos:

  - **Caso 1**. Los hilos **A** y **B** llaman `pop()` (sacan elementos de la pila).
    Supongamos que **A** y **B** estan a punto de llamar `pop()`, supóngamos que si **A** ejecuta `pop()` primero y luego **B** ejecuta `pop()`, el sistema entra en un estado s_0, 0-valente; y que si **B** ejecuta `pop()` primero y luego **A** ejecuta `pop()`, el sistema entra en un estado s_1, 1-valente.
    Vemos que en este caso tanto s_0 como s_1 son indistinguibles para **C** (quien se ejecuta ininterrumpidamente desde s_0 y s_1), ya que en ambos casos, los mismos dos elementos fueron sacados de la pila (respetando el orden `LIFO`), sin embargo, **C** no puede decidir diferentes valores en dos estados indistinguibles, lo que nos lleva a una contradicción.
  - **Caso 2**. **A** llama a `push(a)` y **B** llama a `pop()`.
    - Digamos que s_1 es el estado 1-valente al que se llega con este orden de operaciones, **A** hace `push(a)`, luego **B** hace `pop()` (del elemento `a`) y **A** hace `pop()` (del elemento superior

de la pila, si existe alguno) (de esta manera puede saber si realmente fue el primero en moverse, ya que `pop()` es la única manera de ver el estado de la pila). Ahora sea s_0 el estado 0-valente al que se llega cuando primero **B** hace `pop()` (del elemento superior de la pila, si existe alguno), luego **A** hace `push(a)` y por ultimo **A** hace `pop()` (del elemento `a`) (de esta manera puede saber si realmente fue el primero en moverse, ya que `pop()` es la única manera de ver el estado de la pila). Podemos notar claramente que **C** no puede decidir diferentes los dos estados s_0 y s_1, ya que el estado de la pila es el mismo. Notemos que no nos importa que pasa cuando se hace un `pop()` a una pila vacía, ya que no afecta la visibilidad de **C**.

- **Caso 3**. **A** llama a `push(a)` y **B** llama a `push(b)`.

  - **Ejecucion 1**. **A** realiza `push(a)` primero y luego **B** realiza `push(b)`. Luego **A** corre hasta que haga `pop()` del elemento `b` (de esta manera puede saber si realmente fue el primero en hacer `push()`, ya que `pop()` es la única manera de ver el estado de la pila), **B** realiza algo similar solo corriendo hasta hacer `pop()` del elemento `a`. De esta manera los dos hilos terminaran de correr cuando la pila este vacía (cada uno saca un elemento). Esto nos lleva al estado s_0 donde la pila esta vacía.
  - **Ejecucion 2**. **B** realiza `push(b)` primero y luego **A** realiza `push(a)`. Luego **B** corre hasta que haga `pop()` del elemento `a` (de esta manera puede saber si realmente fue el primero en hacer `push()`, **A** realiza algo similar solo corriendo hasta hacer `pop()` del elemento `b`. De esta manera los dos hilos terminaran de correr cuando la pila este vacía (cada uno saca un elemento). Esto nos lleva al estado s_1 donde la pila esta vacía.

  Podemos observar que en ambas ejecuciones el estado que **C** ve (s_0 y s_1) son indistinguibles para el, ya que al realizar dos operaciones `pop()` los mismos dos elementos se sacan de la pila, por lo cual, para **C** esto es una contradicción ya que, dos estados indistinguibles llevan a decisiones diferentes. Ademas las dos ejecuciones de **A** son similares ya que se detiene hasta que hace `pop()`, de manera similar para **B**.

De esta manera ya cubrimos todos los casos, y podemos concluir que pilas no resuelven el consenso para 3 hilos.

Referencias

[1] Herlihy, M. (2020). The Art of Multiprocessor Programming. En Elsevier eBooks. https://doi.org/10.1016/c2011-0-06993-4