Universidad Nacional Autónoma de México
Facultad de Ciencias

Computación Concurrente

Tarea 4

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

Fundamentos de memoria compartida: Registros y Snapshots

Ejercicio 1

De acuerdo a la clase y al siguiente video `https://youtu.be/iAqyufwYcHc?si=rRdjISdUndNSakb9` (también puedes consultar el libro principal en el capítulo 4) contesta de forma breve:

(a) ¿En qué consiste la transformación de un registro seguro a un registro regular?

La transformación de un registro seguro a un registro regular implica la creación de un registro regular a partir de uno seguro. Un registro seguro garantiza que se puede leer el valor correcto si no hay nadie escribiendo en él, pero no garantiza que se obtenga el valor correcto si hay lecturas concurrentes durante una escritura. El autor menciona haber considerado primero la implementación de registros regulares usando registros seguros, lo que sugiere que fue posible construir registros regulares a partir de registros seguros, aunque más tarde se encontró con la dificultad de crear registros atómicos a partir de regulares. Además, para transformarlo se agrega un registro extra donde se guarda el valor viejo para poder devolverlo si es necesario.

(b) ¿Cuál es la diferencia entre un registro atómico y uno regular?

La principal diferencia entre un registro atómico y uno regular es la forma en que gestionan las lecturas y escrituras concurrentes. Un registro atómico garantiza que cualquier operación de lectura o escritura ocurre de manera instantánea, es decir, las operaciones parecen ejecutarse en un solo paso, asegurando que las lecturas siempre reflejan el valor más reciente, incluso si hay concurrencia. Por otro lado, un registro regular solo garantiza que las lecturas darán un valor coherente si no ocurre una escritura concurrente; en caso contrario, la coherencia no está garantizada.

Por otro lado, otra diferencia es que cuando hay concurrente entre `read` y `write`, el atómico regresa el último valor actualizado o el que se va a actualizar, mientras que él regula cualquier valor de las llamadas traslapadas.

(c) ¿Qué relación hay entre los registros atómicos y el algoritmo de Bakery (para Lamport)?

El algoritmo de Bakery de Lamport está relacionado con los registros atómicos, ya que este algoritmo aborda el problema de coordinar el acceso a la sección crítica entre varios procesos. Lamport estudió clases de registros que son más débiles que los atómicos, como los registros seguros y regulares, para comprender cómo construir registros atómicos y mejorar la coordinación de procesos en concurrencia.

Lamport pensaba que su algoritmo necesitaba registros atómicos, pero al final se mostró que no eran necesarios, sino que incluso hacían al algoritmo un poco menos eficiente.

Ejercicio 2

Dados los siguientes valores de las vistas V_d y V_a, ¿es posible que la ejecución sea una ejecución del Snapshot Wait-free? Si tu respuesta es sí, argumenta por qué es linealizable con respecto a un objeto de tipo Snapshot atómico (muestra una linearización).

a) Si $V_d = [\perp, v_2, v_4, \perp]$ y $V_a = [v_1, v_2, v_4, \perp]$.

No es posible que esta sea una ejecución del Snapshot Wait-free.

Esto debido a que los valores de V_d no pueden estar asi, ya que el valor del hilo A en V_d es $\perp$ pero tenemos que D $scan() : V_d$ y A $update(v_1) : ok$ no son concurrentes por lo cual A $update(v_1) : ok$ termina antes de la llamada de D $scan() : V_d$ y esto nos indica que al realizar los collects del $scan()$ se tuvo que ver el valor v_1 para el hilo A, pero en V_d no aparece, por lo tanto no puede pasar que esta sea una ejecución, V_d siempre debe tener el valor v_1 ya que A $update(v1) : ok$ termina antes que la llamada del D $scan() : V_d$. Aunque los valores de V_a si son correctos.

b) Si $V_d = [v_1, v_2, v_4, \perp]$ y $V_a = [v_1, v_2, v_3, \perp]$.

No es posible que esta sea una ejecución del Snapshot Wait-free.

Esto debido a que los valores de V_a no pueden estar asi, ya que el valor del hilo C en V_A es v_3 pero tenemos que A $scan() : V_a$ y C $update(v_4) : ok$ no son concurrentes por lo cual C $update(v_4) : ok$ termina antes de la llamada de A $scan() : V_a$ y esto nos indica que al realizar los collects del $scan()$ se tuvo que ver el valor v_4 y no el del v_3 (el método C $update(v_3) : ok$ termino antes que la llamada de C $update(v_4) : ok$, por lo v_4 sería el valor más reciente actualizado) para el hilo C, pero en V_a no aparece v_4 y si v_3, por lo tanto, no puede pasar que esta sea una ejecución, V_a siempre debe tener el valor v_4 ya que C $update(v4) : ok$ termina antes que la llamada del A $scan() : V_a$ y el método C $update(v_3) : ok$ termino antes que de la llamada de C $update(v4) : ok$. Aunque los valores de V_d si son correctos.

c) Si $V_d = [v_1, v_2, v_3, \perp]$ y $V_a = [v_1, v_2, v_4, \perp]$.

Si es posible que esta sea una ejecución del Snapshot Wait-free.

Veamos la historia H para la linealización (vamos a tener que $H = H' = complete(H') = S$), digamos que el objeto Snapshot se llama O, entonces:

A $O.update(v_1)$
A $O : ok$
B $O.update(v_2)$
B $O : ok$
C $O.update(v_3)$
C $O : ok$
C $O.scan()$
C $O : V_d$
C $O.update(v_4)$
C $O : ok$
A $O.scan()$
A $O : V_a$

De esta manera, ambas vistas tienen sus valores correctos, ya que tienen los valores de la actualización más reciente para cada hilo dependiendo de cuando fue él $scan()$ (los valores actualizados antes de la llamada de $scan()$).

Lo importante es que $scan() : V_d$ hace sus dos collects antes de la llamada de $update(v_4)$, por lo tanto, V_d puede tener el valor v_3 y no v_4, ya que los dos métodos son concurrentes y sabemos que en los registros atómicos si una lectura es concurrente con una lectura, la lectura regresa el último valor actualizado o el valor de la nueva actualización.

Ejercicio 3

Plantea una ejecución del Snapshot Wait-free (código en 2) para $n = 4$ hilos en donde muestres un ejemplo de por qué esta implementación es *bounded wait-free*. Es decir, que cada *update()* o s*can()* termina en a lo más $O(n^2)$ lecturas o escrituras (pasos).

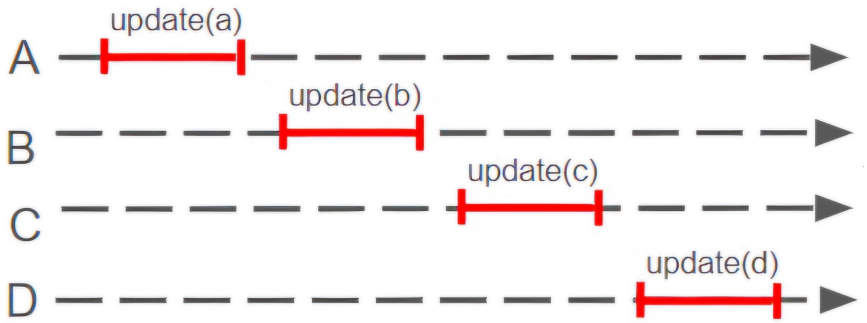
```
1    public void update(T value) {
2        int me = ThreadID.get();
3        T[] snap = scan();
4        StampedSnap<T> oldValue = a_table[me];
5        StampedSnap<T> newValue =
6            new StampedSnap<T>(oldValue.stamp+1, value, snap);
7        a_table[me] = newValue;
8    }
9
10   public T[] scan() {
11       StampedSnap<T>[] oldCopy;
12       StampedSnap<T>[] newCopy;
13       boolean[] moved = new boolean[a_table.length];
14       oldCopy = collect();
15       collect: while (true) {
16           newCopy = collect();
17           for (int j = 0; j < a_table.length; j++) {
18               if (oldCopy[j].stamp != newCopy[j].stamp) {
19                   if (moved[j]) {
20                       return oldCopy[j].snap;
21                   } else {
22                       moved[j] = true;
23                       oldCopy = newCopy;
24                       continue collect;
25                   }
26               }
27           }
28           T[] result = (T[]) new Object[a_table.length];
29           for (int j = 0; j < a_table.length; j++)
30               result[j] = newCopy[j].value;
31           return result;
32       }
33   }
```

Snapshot Wait-free

Para mostrar una ejecución para el código dado, primero notemos que tendremos dos casos, uno en donde los hilos no se traslapan y otro en el que esto sí sucede. Entonces, tendremos lo siguiente:

- **Caso 1:**

  Para este primer caso tendremos que los `update()` de los hilos no se traslapan, lo que conlleva que todos los hilos hagan solo `scan()`, ya que al hacer esto, nadie notará cambios y todos terminarán luego de dos `collect()`). Para ello, tendremos la siguiente ejecución:



  - *Explicación:*

    Para esta ejecución notemos que antes de cada `update` se hará un `scan()` y dado que no habrá ningún hilo concurrente en el `scan()`, entonces se realizará un `collect()` para obtener la vista vieja.

    Después, se realizará un segundo `collect()` para ahora obtener la vista nueva y entrará al `for`. En este punto, veremos que no hay diferencias entre la vista vieja y la nueva, por lo que entonces se terminará su `scan()` y después su `update()`.

  - *Complejidad:*

    Ahora, como ya tenemos la serie de pasos que sigue la ejecución, entonces podremos ahora podemos analisas su complejidad.

    Notemos que la composición del método `update()` es $O(1)$, a excepción de la llamada al método `scan()`, que describiremos a continuación.

    Veamos que `scan()` es $O(n)$, ya que en este se hacen dos `collect()` con complejidad $O(n)$, además de un ciclo `for` para revisar si hay cambios, lo cual toma $O(n)$, es decir, esto es $O(3n)$, que por análisis de algoritmos sabemos que termina siendo $O(n)$.

    Sin embargo, notemos que $O(n)$ está acotado por $O(n^2)$, por lo que finalmente podemos decir que el método `scan()` termina es $O(n^2)$ y, por consecuencia, `update()` también.

- **Caso 2:**

  Para el segundo caso tendremos que los hilos se traslapan o son concurrentes. Para ello, tendremos la siguiente ejecución:

- *Explicación:*

  Notemos que todos los hilos necesitan terminar un `scan()` para poder terminar su `update()`. De esta manera, veamos cómo sería la ejecución para que todos terminen. Primero, todos los hilos harán un `collect()`, es decir:

  - `B` realiza su primer `collect()`
  - `A` realiza su primer `collect()`
  - `C` realiza su primer `collect()`
  - `D` realiza su primer `collect()`

  Nota: El orden de los hilos al hacer su primer `collect()` no afecta demasiado.

  Después, los hilos harán su segundo `collect()` y su primer `for` de la siguiente manera:

  - `B` realiza su segundo `collect()` y su primer `for` para revisar si hay cambios entre lo obtenido. Ve que lo obtenido es igual, entonces termina su `scan()`, pasando a la línea 7 del `update()` y terminándolo.
    - **Complejidad:**
      `B` terminó su `scan()` y su `update()` en dos `collect()` y un `for`, lo cual tomó $O(3n)$.
  - `A` realiza su segundo `collect()` y su primer `for` para revisar si hay cambios entre lo obtenido. Ve que `B` se movió.
  - `C` realiza su segundo `collect()` y su primer `for` para revisar si hay cambios entre lo obtenido. Ve que `B` se movió.
  - `D` realiza su segundo `collect()` y su primer `for` para revisar si hay cambios entre lo obtenido. Ve que `B` se movió.

  Ahora, los hilos harán su tercer `collect()` y su segundo `for` de la siguiente manera:

  - `A` realiza su tercer `collect()` y su segundo `for` para revisar si hay cambios entre lo obtenido. Ve que lo obtenido es igual, entonces termina su `scan()`, pasando a la línea 7 del `update()` y terminándolo.
    - **Complejidad:**
      `A` terminó su `scan()` y `update()` en tres `collect()` y dos `for`, lo cual es $O(5n)$.
  - `C` realiza su tercer `collect()` y su segundo `for` para revisar si hay cambios entre lo obtenido. Ve que `A` se movió.
  - `D` realiza su tercer `collect()` y su segundo `for` para revisar si hay cambios entre lo obtenido. Ve que `A` se movió.
  - `C` realiza su cuarto `collect()` y su tercer `for` para revisar si hay cambios entre lo obtenido. Ve que lo obtenido es igual, entonces termina su `scan()`, pasando a la línea 7 del `update()` y terminándolo.
    - **Complejidad:**
      `C` terminó su `scan()` y `update()` en cuatro `collect()` y tres `for`, lo cual es $O(7n)$.

  Ahora, los hilos harán su cuarto `collect()` y su tercer `for` de la siguiente manera:

  - `D` realiza su cuarto `collect()` y su tercer `for` para revisar si hay cambios entre lo obtenido. Ve que `A` se movió.
  - `D` realiza su quinto `collect()` y su cuarto `for` para revisar si hay cambios entre lo obtenido. Ve que lo obtenido es igual, entonces termina su `scan()`, pasando a la línea 7 del `update()` y terminándolo.
    - **Complejidad:**
      `D` terminó su `scan()` y `update()` en cinco `collect()` y cuatro `for`, lo cual es $O(9n)$.

- *Complejidad:*

  Notemos que el último hilo terminó su `update()` y su `scan()` en $O(9n)$, pero dado que tenemos 4 hilos, entonces $m = 4$. Por lo tanto, $9 = 2 \cdot 4 + 1$, es decir:

  $$O(9n) = O((2 \cdot 4 + 1)n) = O((2n + 1)n) = O(2n^2 + n)$$

  Sin embargo, notemos que esto está acotado por $O(n^2)$, por lo que `update()` y `scan()` terminan son $O(n^2)$.

Ejercicio 4

Describe una ejecución del Snapshot Obstruction-Free en la cual el método scan() de un hilo A nunca termina (considera $n = 3$ hilos).

```
1  public class SimpleSnapshot<T> implements Snapshot<T> {
2    private StampedValue<T>[] a_table; // array of atomic MRSW registers
3    public SimpleSnapshot(int capacity, T init) {
4      a_table = (StampedValue<T>[]) new StampedValue[capacity];
5      for (int i = 0; i < capacity; i++) {
6        a_table[i] = new StampedValue<T>(init);
7      }
8    }
9    public void update(T value) {
10     int me = ThreadID.get();
11     StampedValue<T> oldValue = a_table[me];
12     StampedValue<T> newValue =
13         new StampedValue<T>((oldValue.stamp)+1, value);
14     a_table[me] = newValue;
15   }
16   private StampedValue<T>[] collect() {
17     StampedValue<T>[] copy = (StampedValue<T>[])
18         new StampedValue[a_table.length];
19     for (int j = 0; j < a_table.length; j++)
20       copy[j] = a_table[j];
21     return copy;
22   }
23   public T[] scan() {
24     StampedValue<T>[] oldCopy, newCopy;
25     oldCopy = collect();
26     collect: while (true) {
27       newCopy = collect();
28       if (! Arrays.equals(oldCopy, newCopy)) {
29         oldCopy = newCopy;
30         continue collect;
31       }
32       T[] result = (T[]) new Object[a_table.length];
33       for (int j = 0; j < a_table.length; j++)
34         result[j] = newCopy[j].value;
35       return result;
36     }
37   }
```

Figura 1: Código Snapshot Obstruction-Free

Veamos como puede suceder esto. Supongamos que tenemos tres hilos **A**,**B** y **C** y el método scan() de **A** nunca termina. Esto puede suceder si los hilos **B** y **C** están actualizando los registros continuamente,

lo que causa que **A** nunca pueda completar el método scan() ya que siempre está recogiendo nuevas instantáneas y nunca encuentra una idéntica a la anterior.

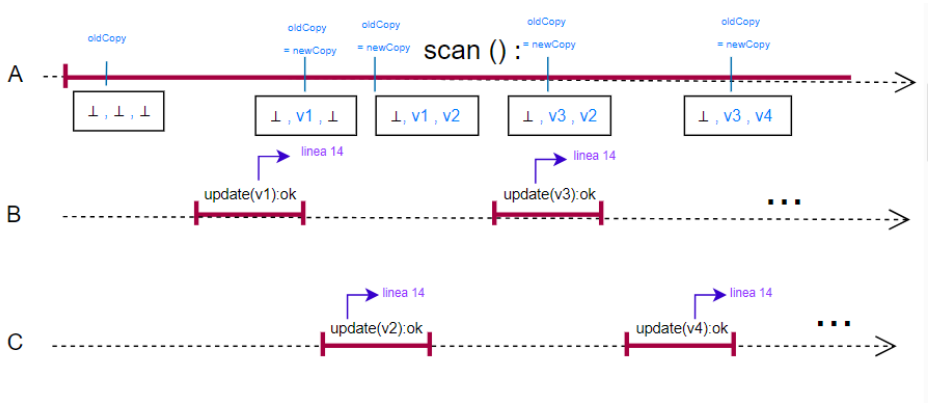


Figura 2: Ejecución donde **A** nunca termina scan()

En la ejecución anterior, denotamos ... tanto en el hilo B como en el hilo C, para indicar que ambos continúan realizando actualizaciones.

Como resultado, la condición en la línea 28,siempre se cumple, lo que significa que la nueva copia nunca será igual a la anterior, ya que las actualizaciones continúan ocurriendo. Estos puntos suspensivos indican que tanto B como C seguirán realizando actualizaciones, lo que impide que el hilo A termine nunca su ejecución.

Ejercicio 5

Si en la implementación del Snapshot Obstruction-free modificamos el método scan() de la siguiente manera:

```
   public T[] scan() {
1:   copy = collect();
2:   T[] result = T[] new Object(a_table.length);
3:   for (int j = 0; j< a_table.length; j++) {
4:     result[j] = copy[j].value;
5:     }
6:   return result;
```

Figure 4: Clase "VisibilityField"

- ¿Podría suceder la siguiente implementación? Supón que el arreglo está inicializado: $[\bot, \bot, \bot]$

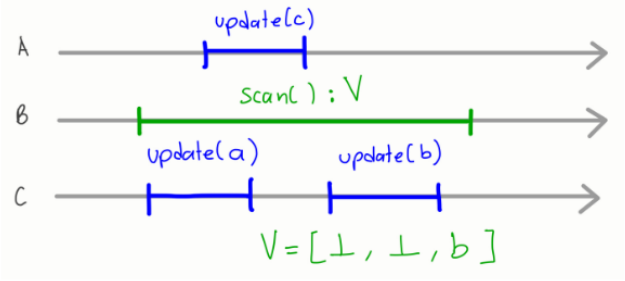


Figure 5

Veamos que no puede suceder. Sabemos que la vista final es $[\perp, \perp, b]$, notamos dos cosas: que el hilo **C** realiza `update(b)` después de que el hilo **A** termina su ejecución de `update(c)`, por lo que, si el hilo **B** pudo leer la actualización del hilo **C** al ejecutar `scan()` y collect(), también debería haber leído el `update(c)` del hilo **A**.

- ¿La implementación seguiría siendo linealizable? Argumenta por qué.

  Por las diapositivas de clase, sabemos que el método `update()` es linealizable, y el único cambio en el código es el método scan, así que veamos si la implementación sigue siendo linealizable.

  La linealizabilidad nos dice que cada operación debe parecer que tomo efecto instantáneamente en algún punto entre la invocación y la respuesta y preserva el orden real de las operaciones.

  Observamos que llamamos `collect()` solo una vez en nuestra nueva implementación de `scan()`. En este punto, las llamadas a `update()` se pueden ordenar de manera secuencial y ser consistentes con el orden de ejecución, por lo que aun con este cambio, sigue siendo linealizable.

  Veamos algunos ejemplos de como si se puede linealizar cuando un `scan()` se traslapa con un `update()` (en estos ejemplos el tache azul es la linea 14 del método `update()` y la linea morada es cuando se hace el `collect()`)

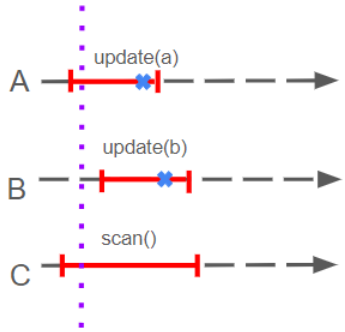


Figura 3: Aquí la vista que se regresa seria $[\perp, \perp, \perp]$ y es una vista correcta.

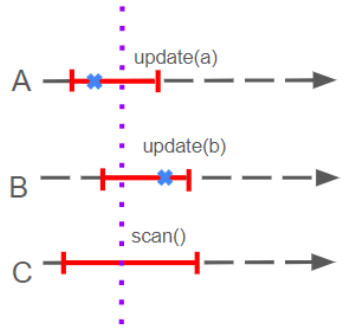


Figura 4: Aquí la vista que se regresa seria $[a, \perp, \perp]$ y es una vista correcta.

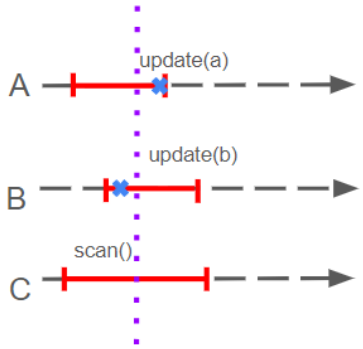


Figura 5: Aquí la vista que se regresa seria [⊥,b, ⊥] y es una vista correcta.

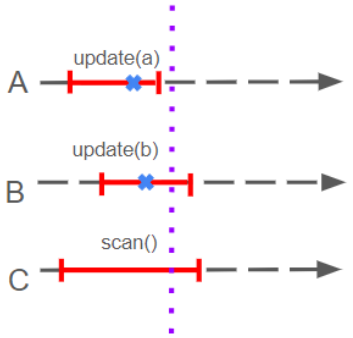


Figura 6: Aquí la vista que se regresa seria [a,b, ⊥] y es una vista correcta.

Ahora revisemos otros ejemplos de como si se puede linealizar cuando un `scan()` se traslapa con otro `scan()` y un `update()` (ya que si solo son dos `scan()` entonces las vistas siempre deberían ser iguales, que es igual al caso cuando solo es un `scan()`, no pasa nada interesante y es valido por la definición del método `scan()`) (en estos ejemplos el tache azul es la linea 14 del método `update()` y la linea morada es cuando se hace el `collect()` del `scan()` del hilo C y la linea verde para el `collect()` del `scan()` del hilo B)

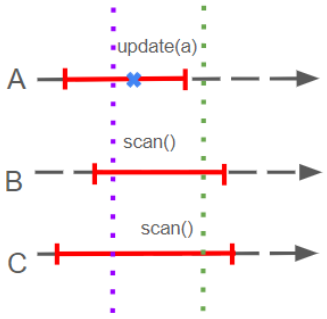


Figura 7: Aquí la vista que regresa el hilo C es $V = [\bot, \bot, \bot]$ y la vista que regresa el hilo B es $V' = [a, \bot, \bot]$, tenemos que las vistas son validas y además $V \leq V'$, lo cual es valido.

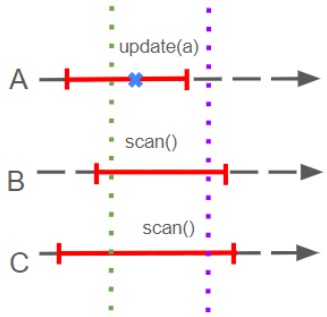


Figura 8: Aquí la vista que regresa el hilo C es $V = [a, \perp, \perp]$ y la vista que regresa el hilo B es $V' = [\perp, \perp, \perp]$, tenemos que las vistas son validas y además $V' \leq V$, lo cual es valido.

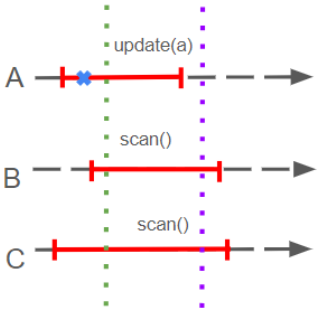


Figura 9: Aquí la vista que regresa el hilo C es $V = [a, \perp, \perp]$ y la vista que regresa el hilo B es $V' = [a, \perp, \perp]$, tenemos que las vistas son validas y además $V' = V$, lo cual es valido.

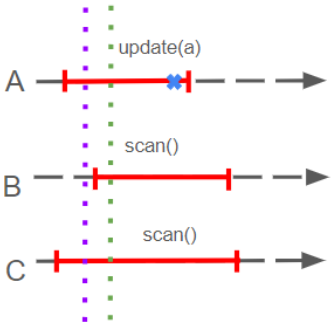


Figura 10: Aquí la vista que regresa el hilo C es $V = [\perp, \perp, \perp]$ y la vista que regresa el hilo B es $V' = [\perp, \perp, \perp]$, tenemos que las vistas son validas y además $V' = V$, lo cual es valido.

Entonces notamos que en todos los ejemplos si son correctos y como los métodos se traslapan podemos acomodarlos como sea conveniente para que las vistas tengan sentido.
Y por lo tanto si es linealizable, en particular la linea de linealizacion para `update()` es la 14 (`a_table[me] = newValue;`) y la linea para `scan()` es la 1 (`copy = collect();`).

Referencias

[1] Ayudantías y clases de teoría del 23 de septiembre al 1 de octubre del 2024.