Universidad
Nacional Autónoma de México

Facultad de Ciencias

Examen 1

Integrantes:
Yonathan Berith Jaramillo Ramírez. 419004640

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

30 Septiembre, 2023

Computo concurrente

Instrucciones: Contesta los ejercicios que elijas, la calificación máxima es 100, aunque también puedes obtener un extra.

Pueden discutir el examen con otras personas, pero el examen se escribe y se entrega individualmente (si se ven exámenes escritos idénticos se les dividirá la calificación, pueden compartir una idea, pero no puede ser que dos personas redacten de forma idéntica).

Se entrega virtualmente, en el formato que más les convenga siempre y cuando sea legible.

No hay reposición, se puede entregar con máximo un día de retraso para ser evaluado sobre 9.

Ejercicios:

1. **(20 puntos)** Ejercicio 4 (ítem 2). Describe una estrategia cuando no conocemos el estado inicial del switch.

   Exercise 4. You are one of P recently arrested prisoners. The warden, a deranged computer scientist, makes the following announcement:

   You may meet together today and plan a strategy, but after today you will be in isolated cells and have no communication with one another.

   I have set up a "switch room" which contains a light switch, which is either *on* or *off*. The switch is not connected to anything.

   Every now and then, I will select one prisoner at random to enter the "switch room." This prisoner may move the switch (from *on* to *off*, or vice-versa), or may leave the switch unchanged. Nobody else will ever enter this room.

   Each prisoner will visit the switch room arbitrarily often. More precisely, for any N, eventually each of you will visit the switch room at least N times.

   At any time, any of you may declare: "we have all visited the switch room at least once." If the claim is correct, I will set you free. If the claim is incorrect, I will feed all of you to the crocodiles. Choose wisely!

   Devise a winning strategy when you do not know whether the initial state of the switch is *on* or *off*.

   Hint: not all prisoners need to do the same thing.

   Resultado: La estrategia implica asignar roles específicos a los prisioneros.

   Dado que no conocemos el estado inicial del interruptor, consideraremos dos escenarios:

   El interruptor está inicialmente APAGADO. El interruptor está inicialmente ENCENDIDO.

   Estrategia:

   Designar un Contador: Entre todos los P prisioneros, elige a uno para que actúe como el "contador". Este prisionero tiene un papel único. Todos los demás pri-

sioneros tienen el mismo papel y son los "no contadores".

Rol del No Contador:

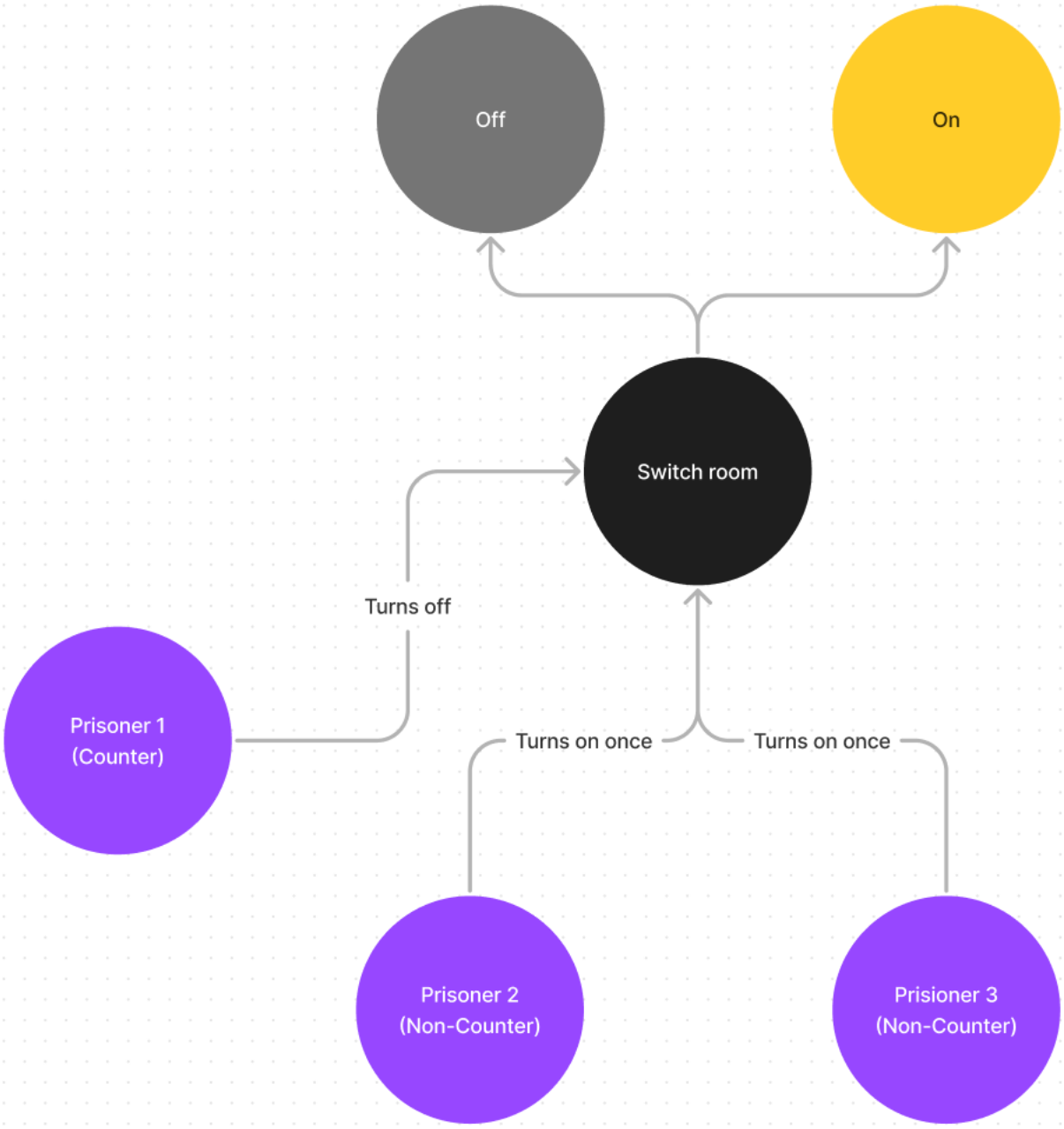
Cuando un prisionero no contador entra en la sala y ve el interruptor en el estado inicial (para esta explicación, supongamos que está APAGADO) y nunca antes ha cambiado el interruptor, lo cambian al otro estado (ENCENDIDO) para señalar que han estado en la sala al menos una vez. Sólo harán esto una vez. Si ya han cambiado el interruptor en una visita anterior, no hacen nada independientemente del estado del interruptor. Si el prisionero no contador entra en la sala y ve que el interruptor está ENCENDIDO, no hace nada.

Rol del Contador:

Si el contador entra en la sala y ve que el interruptor está ENCENDIDO, lo cambia a APAGADO y cuenta eso como un prisionero que ha visitado la sala. Si el contador entra en la sala y ve que el interruptor está APAGADO, no hace nada. Declaración para ser Liberado: Una vez que el contador ha contado P-1 señales (el menos uno es porque el propio contador no da una señal), puede declarar con seguridad que todos los prisioneros han estado en la sala del interruptor al menos una vez.

Esta estrategia es para cuando el interruptor esté inicialmente APAGADO. Si consideras que el interruptor está inicialmente ENCENDIDO, simplemente invierte los ENCENDIDOS y APAGADOS en la estrategia.

Ahora, ¿cómo tratamos la incertidumbre del estado inicial del interruptor? Implementamos ambas estrategias en paralelo. Designamos a dos prisioneros como contadores, uno para cada estado inicial asumido, y procedemos con ambas estrategias simultáneamente.



2. **(30 puntos)** Ejercicio 13 de [HS08].

   Exercise 13. Another way to generalize the two-thread Peterson lock is to arrange a number of 2-thread Peterson locks in a binary tree. Suppose n is a power of two. Each thread is assigned a leaf lock which it shares with one other thread. Each lock treats one thread as thread 0 and the other as thread 1.

   In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method for the

tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired, from the root back to its leaf. At any time, a thread can be delayed for a finite duration. (In other words, threads can take naps, or even vacations, but they do not drop dead.) For each property, either sketch a proof that it holds, or describe a (possibly infinite) execution where it is violated.

(a) Mutual exclusion.

(b) Freedom from deadlock.

(c) Freedom from starvation.

Is there an upper bound on the number of times the tree-lock can be acquired and released between the time a thread starts acquiring the tree-lock and when it succeeds?

(a) **Exclusión Mutua**:

- *Prueba*: Si dos o más hilos llegan a la raíz del árbol binario, entonces han adquirido con éxito todos los cerrojos de Peterson desde sus respectivas hojas hasta la raíz. El diseño de un cerrojo de Peterson básico garantiza la exclusión mutua entre dos hilos. Por inducción, si a dos hilos se les asignan diferentes cerrojos en el nivel más bajo, siempre estarán adquiriendo diferentes cerrojos en su camino hacia la raíz. Si dos hilos comparten el mismo cerrojo en cualquier nivel, entonces solo uno de ellos puede progresar al siguiente nivel mientras el otro espera. Por lo tanto, es imposible que dos hilos adquieran simultáneamente el cerrojo de árbol, asegurando la exclusión mutua.

(b) **Libertad de Interbloqueo**:

- *Prueba*: La estructura del árbol asegura que si un hilo está bloqueado en algún cerrojo de Peterson, es porque otro hilo está actualmente accediendo o está en la cola para acceder a ese cerrojo. El otro hilo eventualmente liberará el cerrojo debido a la propiedad de retraso finito, y nuestro hilo en espera obtendrá su turno. Dado que hay un camino definido desde la hoja hasta la raíz y ya que cada hilo libera los cerrojos una vez que ha terminado, no hay espera circular. Por lo tanto, el sistema está libre de interbloqueo.

(c) **Libertad de Inanición**:

- *Posible Violación*: Considera dos hilos, A y B, compitiendo por el mismo cerrojo en algún nivel del árbol. Si A adquiere el cerrojo y luego lo libera

inmediatamente, solo para volver a solicitarlo antes de que B tenga una oportunidad, hay una posibilidad de que B nunca obtenga el cerrojo, lo que lleva a la inanición. Aunque el cerrojo de Peterson en cada nivel asegura que si un hilo espera demasiado tiempo, el otro cederá, no hay tal garantía al considerar todo el camino desde la hoja hasta la raíz. Por lo tanto, aunque es poco probable, es teóricamente posible que un hilo pase hambre.

¿Existe un límite superior en la cantidad de veces que se puede adquirir y liberar el cerrojo de árbol entre el momento en que un hilo comienza a adquirir el cerrojo de árbol y cuando tiene éxito?

- *Discusión*: La profundidad del árbol binario es $\log_2 n$, dado que n es el número total de hilos y es una potencia de dos. Cada hilo necesitaría adquirir $\log_2 n$ cerrojos para obtener el cerrojo de árbol. En el peor de los casos, para cada cerrojo que un hilo quiere adquirir, podría tener que esperar a que el otro hilo (que comparte el cerrojo) primero lo adquiera y luego lo libere. Esto da un límite superior de $2 \times \log_2 n$ adquisiciones y liberaciones para el viaje de un solo hilo desde la hoja hasta la raíz.

  Sin embargo, la pregunta de cuántas veces se puede adquirir y liberar el cerrojo por *todos* los hilos en el sistema antes de que un hilo específico tenga éxito es más difícil de determinar. Depende de la programación y el comportamiento de todos los hilos en el sistema. Pero, considerando que cada uno de los n hilos podría estar compitiendo y podría tener que esperar a su pareja para cada cerrojo desde la hoja hasta la raíz, un límite superior podría ser $n \times 2 \times \log_2 n$.

3. **(18 puntos)** Ejercicio 37 de [HS08].

   Exercise 37. Give an example of a quiscently-consistent register execution that is not regular.

   Definiciones de conceptos:

   Consistencia Quiescente: Un sistema es consistentemente quiescente si, después de un período sin operaciones (quiescencia), el sistema parece ser secuencialmente consistente para cualquier observador. Esto significa que incluso si las operaciones no aparecen de una manera secuencialmente consistente durante su ejecución, lo serán si no hay operaciones en curso.

   Registros Regulares (o Seguros): Una lectura en un registro regular que no se superpone con una escritura siempre devolverá el valor más reciente escrito. Si

una lectura se superpone con una escritura, entonces la lectura podría devolver el valor actual que se está escribiendo o cualquier valor anterior. Los registros regulares son más fuertes que los registros atómicos pero más débiles que los registros seguros.

Ejemplo de una ejecución de registro consistentemente quiescente que no es regular:

Considera dos procesos, A y B, y un registro compartido R inicializado a 0.

A escribe 1 en R. B lee concurrentemente de R y ve el valor 0. A completa su operación de escritura. B escribe 2 en R. A lee concurrentemente de R y ve el valor 2. B completa su operación de escritura.

Período quiescente (sin operaciones en curso). Cualquier lectura posterior por cualquier proceso verá el valor 2, haciendo que parezca secuencialmente consistente. Esta ejecución es consistentemente quiescente porque después del período quiescente, todas las lecturas posteriores ven el valor final escrito, haciendo que toda la historia de operaciones parezca secuencialmente consistente.

Sin embargo, esta ejecución no es regular. La consistencia regular requiere que si una lectura ve el resultado de una escritura, entonces todas las lecturas posteriores también deben ver esa escritura o alguna escritura posterior. En el escenario anterior, la lectura de B vio la escritura inicial de 0 de A, pero la lectura posterior de A vio la escritura concurrente de 2 de B, lo cual viola la condición de consistencia regular.

4. **(15 puntos)** Da un ejemplo de una ejecución de una implementación de una una Cola secuencialmente consistente que no es linealizable, con al menos dos hilos y al menos 4 llamadas a métodos (enq() o deq()). Argumenta porqué.

   Para resolver esta pregunta, primero debemos entender las diferencias entre "consistencia secuencial" y "linealizabilidad":

   (a) **Consistencia Secuencial**: Un objeto concurrente es secuencialmente consistente si los resultados de cualquier ejecución son los mismos que si las operaciones de todos los procesos fueran ejecutadas en algún orden secuencial y las operaciones de cada proceso individual aparecen en ese orden.

   (b) **Linealizabilidad**: Un objeto concurrente es linealizable si las operaciones parecen, a todos los hilos, que se están ejecutando instantáneamente en algún punto entre cuando se inicia y cuando se completa. Es una forma más fuerte

de consistencia secuencial ya que determina puntos específicos en el tiempo (no solo un orden) en el que se puede considerar que cada operación ocurre.

Ejemplo de una Cola secuencialmente consistente que no es linealizable:

Imaginemos una cola compartida inicialmente vacía y dos hilos, Hilo A y Hilo B.

(a) Hilo A invoca `enq(1)`.

(b) Hilo B invoca `enq(2)`.

(c) Hilo A invoca `deq()` y recibe `2`.

(d) Hilo B invoca `deq()` y recibe `1`.

De acuerdo con la consistencia secuencial, esta ejecución es válida ya que podemos ordenar las operaciones de manera secuencial de la siguiente forma y obtener el mismo resultado:

(a) `enq(2)`

(b) `deq()` $\rightarrow$ `2`

(c) `enq(1)`

(d) `deq()` $\rightarrow$ `1`

Sin embargo, esta ejecución no es linealizable. No hay un punto entre el inicio y la finalización de las operaciones de encolar (`enq()`) en el que podamos decir que una operación ocurre antes que la otra, y todavía produce el resultado observado. Específicamente, si `enq(1)` de Hilo A es linealizado antes de `enq(2)` de Hilo B, entonces Hilo A no debería haber podido desencolar `2` antes que `1`.

Por lo tanto, hemos dado un ejemplo de una ejecución de una cola secuencialmente consistente que no es linealizable.

5. **(15 puntos)** Observa el siguiente bloque de código, supongamos que solo 2 hilos se ejecutan y nunca fallan, ¿Cuál sería el resultado teórico del programa? Elige un inciso. Justifica.

```java
public class Counter implements Runnable {
    public static final int ROUNDS = 5;
    private int counter = 0;
    Lock peterson = new PetersonLock();

    @Override
    public void run() {
        for(int i=0; i< Counter.ROUNDS; i++) {
            peterson.lock();
            try{
                counter++;
            } catch (Exception e){
                peterson.unlock();
            }

        }
    }
```

(a) Nunca termina.

(b) Termina en algunas ejecuciones pero hay una condici ón de carrera.

(c) Siempre termina pero hay una condición de carrera.

(d) Termina en algunas ejecuciones y siempre counter = 10

(e) Siempre termina con counter = 10

El código utiliza el algoritmo de bloqueo de Peterson para asegurar que, cuando múltiples hilos intenten entrar en una sección crítica, solo uno pueda hacerlo a la vez. En este caso, la sección crítica es el incremento del contador `counter`.

Dado que:

- Hay 2 hilos.
- Cada hilo incrementa el contador 5 veces (dado por `Counter.ROUNDS`).
- Se utiliza el bloqueo de Peterson para asegurar la exclusión mutua.

El comportamiento esperado sería que cada hilo incremente el contador de manera segura, sin interferencias, y que al final de la ejecución de ambos hilos, el contador tenga el valor de 10 (5 incrementos por 2 hilos).

Vamos a analizar las opciones:

(a) **Nunca termina**: No hay evidencia en el código proporcionado de que los hilos puedan quedar atrapados en un bucle infinito. El bloqueo de Peterson garantiza que los hilos eventualmente entrarán en su sección crítica.

(b) **Termina en algunas ejecuciones pero hay una condición de carrera**: El bloqueo de Peterson está diseñado para prevenir condiciones de carrera, por lo que esta opción no es correcta.

(c) **Siempre termina pero hay una condición de carrera**: Como se mencionó, el bloqueo de Peterson previene condiciones de carrera, así que esta opción tampoco es correcta.

(d) **Termina en algunas ejecuciones y siempre counter = 10**: El código siempre debería terminar, no solo en algunas ejecuciones. Además, dado que se utiliza el bloqueo para garantizar la exclusión mutua, el contador siempre debería llegar a 10.

(e) **Siempre termina con counter = 10**: Esta es la opción correcta. Dado que se utilizan dos hilos y cada hilo incrementa el contador 5 veces, y se asegura la exclusión mutua utilizando el bloqueo de Peterson, el valor final del contador siempre debe ser 10.

Por lo tanto, la respuesta correcta es:

(e) Siempre termina con counter = 10

6. **(15 puntos)** Considera el siguiente bloque de código que representa un contador:

```
public class Counter implements Runnable {
    public static final int ROUNDS = 5;
    private int counter = 0;

    @Override
    public void run() {
        for(int i=0; i< Counter.ROUNDS; i++) {
            counter++;
        }
    }
}
```

Da el intervalo de valores que se puede obtener con:

- 2 hilos

- 5 hilos
- n hilos

El código proporciona un contador concurrente que incrementa el valor de `counter` un total de `ROUNDS` veces por hilo. Dado que no hay mecanismos de sincronización, se pueden presentar condiciones de carrera.

- **2 hilos**: En el mejor escenario, los hilos se ejecutan de manera secuencial y el contador alcanza el valor de 10. En el peor escenario, debido a condiciones de carrera, el valor mínimo podría ser 5. Por lo tanto, el rango de valores posibles con 2 hilos es [5, 10].
- **5 hilos**: Siguiendo la misma lógica, en el mejor escenario, el contador alcanza el valor de 25. En el peor escenario, el valor mínimo es 5. Por lo tanto, el rango con 5 hilos es [5, 25].
- **n hilos**: Para un número generalizado n de hilos, en el mejor escenario, el contador alcanza $5n$. El peor escenario sigue siendo 5. Por lo tanto, el rango con n hilos es [5, $5n$].

7. **(20 puntos)** Zeratun vino de visita, y nos pidio que le preguntaramos a los alumnos de concur- rente sobre un problema que tiene uno de sus amigos, "El Inge", el cual es el siguiente:

   Nuestro amigo el Inge acaba de paralelizar un código para su chamba, con la finalidad de que se ejecute más rapido. Por el momento, solo cuenta con una computadora con 2 poderosos nucleos, la cual produjo un SpeedUp S2. El Inge quiere saber cuántos núcleos adicionales tendría que comprar para alacanzar el mejor desempeño posible.

   - Ayuda a su amigo el Inge y utiliza la Ley de Amdahl para derivarle una fórmula Sn (SpeedUp con n procesadores) en términos de n y S2
   - Nos volvemos a encontrar al Inge y te dice triste que uno de sus compañeros logro un ascenso en la empresa, pero cree que es falso lo que hizo, pues te cuenta lo siguiente: "Logre optimizar el programa del Inge un 10x haciendo únicamente el 35% de su código paralelo."

     El Inge siente que miente pero no sabe como demostrar la mentira por lo que nos pidio ayuda, ¿Lo que dice su compañero de trabajo es verdad?

   Ayuden al Inge y a Zeratun :D

   Respuesta:

La Ley de Amdahl establece que:

$$S_n = \frac{1}{(1-p) + \frac{p}{n}}$$

donde p es la proporción del código que es paralelizable y n es el número de procesadores.

Dado que ya se tiene un SpeedUp S_2 con 2 procesadores, podemos derivar p de la siguiente manera:

$$S_2 = \frac{1}{(1-p) + \frac{p}{2}}$$

Una vez derivado p utilizando S_2, puedes usar la misma fórmula de Amdahl para calcular S_n con cualquier valor de n.

Nos volvemos a encontrar al Inge y te dice triste que uno de sus compañeros logró un ascenso en la empresa, pero cree que es falso lo que hizo, pues te cuenta lo siguiente: "Logré optimizar el programa del Inge un 10x haciendo únicamente el 35% de su código paralelo."

El Inge siente que miente pero no sabe cómo demostrar la mentira por lo que nos pidió ayuda, ¿Lo que dice su compañero de trabajo es verdad?

Respuesta:

Utilizando la Ley de Amdahl con $S_n = 10$ y $p = 0.35$:

$$10 = \frac{1}{(1-0.35) + \frac{0.35}{n}}$$

Para $n \to \infty$ (el máximo speedup posible), la fracción $\frac{0.35}{n}$ tiende a cero, y aún así, no se alcanza un SpeedUp de 10x. Por lo tanto, lo que dice el compañero de trabajo del Inge es falso.

8. **(35 puntos)** Analiza la siguiente ejecución (figura 1) con respecto a la implementación del Snapshot Wait-free (código en 2). La inicialización del Snapshot es $SS = [\perp, \perp, \perp, \perp]$ en donde $SS[0]$ = registro de A, $SS[1]$ = registro de B, $SS[2]$ = registro de C y $SS[3]$ = registro de D.

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

   (a) Si $V_d = [\perp, v2, v4, \perp]$ y $V_a = [v1, v2, v4, \perp]$.

(b) Si $V_d = [v1, v2, v4, \perp]$ y $V_a = [v1, v2, v3, \perp]$.

(c) Si $V_d = [v1, v2, v3, \perp]$ y $V_a = [v1, v2, v4, \perp]$.

(a) Para $V_d = [\perp, v2, v4, \perp]$ y $V_a = [v1, v2, v4, \perp]$: Ambas vistas son posibles de acuerdo con la ejecución y el código de Snapshot Wait-free. La operación es linealizable con respecto a un objeto de tipo Snapshot atómico ya que las vistas representan estados válidos del sistema en puntos específicos del tiempo.

(b) Para $V_d = [v1, v2, v4, \perp]$ y $V_a = [v1, v2, v3, \perp]$: Dado que no hay ninguna restricción que indique que un proceso debe ver todas las actualizaciones anteriores antes de ver una actualización posterior, esta ejecución también es válida. La operación es linealizable ya que las vistas son consistentes con el estado del sistema en diferentes momentos.

(c) Para $V_d = [v1, v2, v3, \perp]$ y $V_a = [v1, v2, v4, \perp]$: 'A' no debería poder ver la segunda actualización de 'C' sin ver también la de 'D'. Dado que esto contradice el comportamiento esperado de un Snapshot atómico, podemos decir que esta ejecución no es linealizable con respecto a un objeto de tipo Snapshot atómico.

References

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