Universidad Nacional Autónoma de México
Facultad de Ciencias

Computación Concurrente

Tarea 6

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

Listas y 2da parte del Consenso

Ejercicio 1

De acuerdo a las clases y al libro principal, explica brevemente (no más de 5 líneas) que implica la Construcción Universal y porqué es relevante.

La Construcción Universal nos sirve para poder construir objetos *wait-free* o *lock-free* para n hilos, mediante el uso del objeto de consenso más poderoso. Su importancia es que nos permite poder construir estructuras de datos o objetos que se requieran compartir entre varios hilos y que sus resultados sean consistentes y tolerantes a fallos en entornos distribuidos.

Ejercicio 2

De acuerdo a la Construcción Universal Wait-free (código a continuacion). Si se ejecuta para $n = 5$ procesos (A con $id = 0$, B con $id = 1$, C con $id = 2$, D con $id = 3$ y F con $id = 4$) y en la lista de nodos aún no existe ningún nodo, solo está inicializada con el nodo *tail* y $seq = 1$. 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 la lista de nodos si después el hilo F ejecuta el método *apply*() sin deternerse. Justifica tu respuesta.

```
1  public class Universal {
2    private Node[] announce; // array added to coordinate helping
3    private Node[] head;
4    private Node tail = new node(); tail.seq = 1;
5    for (int j=0; j < n; j++){head[j] = tail; announce[j] = tail};
6    public Response apply(Invoc invoc) {
7      int i = ThreadID.get();
8      announce[i] = new Node(invoc);
9      head[i] = Node.max(head);
10     while (announce[i].seq == 0) {
11       Node before = head[i];
12       Node help =  announce[(before.seq + 1 % n)];
13       if (help.seq == 0)
14         prefer = help;
15       else
16         prefer = announce[i];
17       after = before.decideNext.decide(prefer);
18       before.next = after;
19       after.seq = before.seq + 1;
20       head[i] = after;
21     }
22     SeqObject MyObject = new SeqObject();
23     current = tail.next;
24     while (current != announce[i]){
25       MyObject.apply(current.invoc);
26       current = current.next;
27     }
28    head[i] = announce[i];
29    return MyObject.apply(current.invoc);
30   }
31 }
```

Figure 6.6 The wait-free universal algorithm.

Estado inicial:

- La lista de nodos está inicializada solo con el nodo `tail`, cuyo `seq = 1`.
- Los procesos A ($id = 0$), B ($id = 1$), C ($id = 2$), y D ($id = 3$) han ejecutado hasta la línea 9 del método `apply()` y están detenidos.

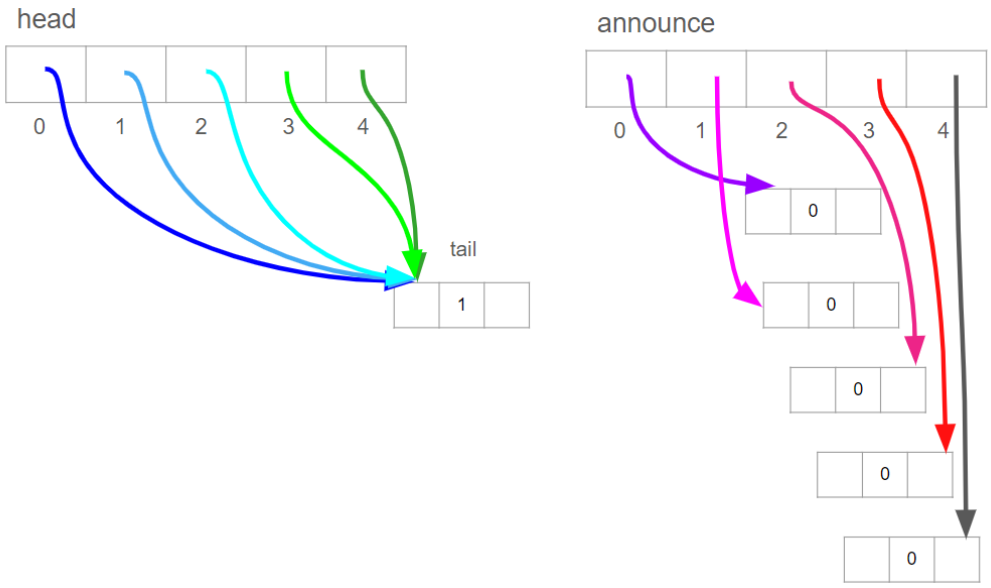
Ejecución del hilo F ($id = 4$):

1. Antes de entrar al `while` de la línea 10, el hilo F evalúa el nodo `before` (último nodo en la lista), que tiene $seq = 1$.
2. En la primera iteración del `while`, F ayuda al hilo con $id = 2$ (hilo C), ya que $before.seq = 1$ y $(1 + 1)\,\%5 = 2$. Esto añade el nodo de C a la lista.
3. En la segunda iteración, F ayuda al hilo con $id = 3$ (hilo D), ya que $before.seq = 2$ y $(2+1)\,\%5 = 3$. Esto añade el nodo de D a la lista.
4. En la tercera iteración, F ayuda a su propio nodo ($id = 4$), ya que $before.seq = 3$ y $(3+1)\,\%5 = 4$. Esto añade el nodo de F a la lista.

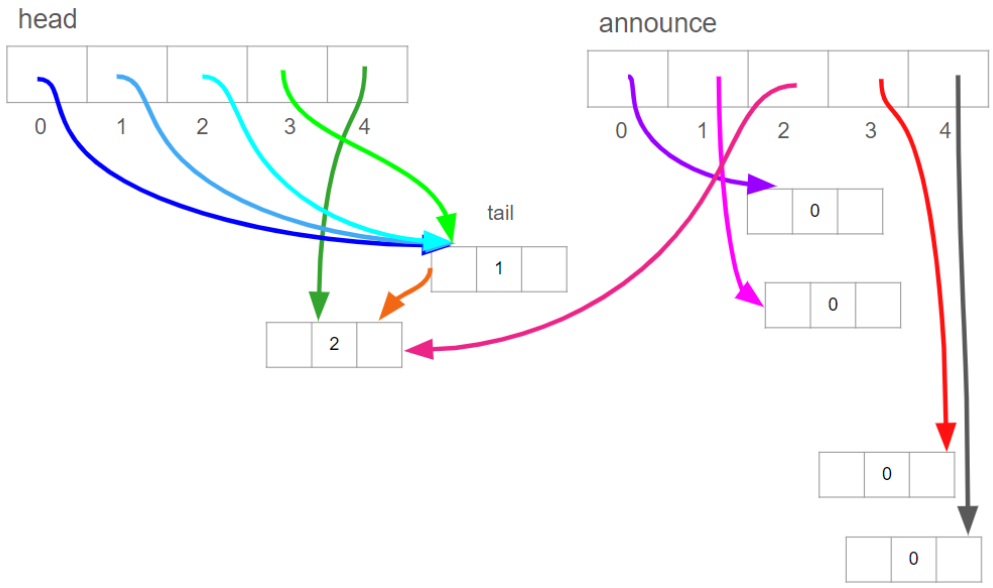
Estado final:

- Cuando el hilo F termina su ejecución de `apply()`, el `seq` de su nodo ya no es 0, por lo que no realiza más iteraciones del `while`.
- En resumen, el hilo F ayuda a los hilos C y D antes de añadir su propio nodo a la lista.

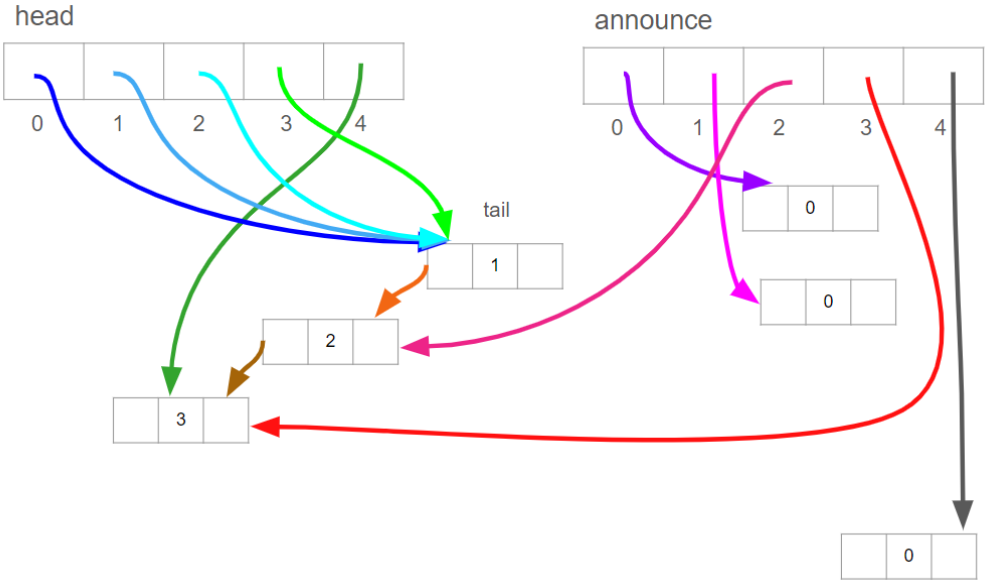
Antes del while de la linea 10 del hilo F tendremos esto



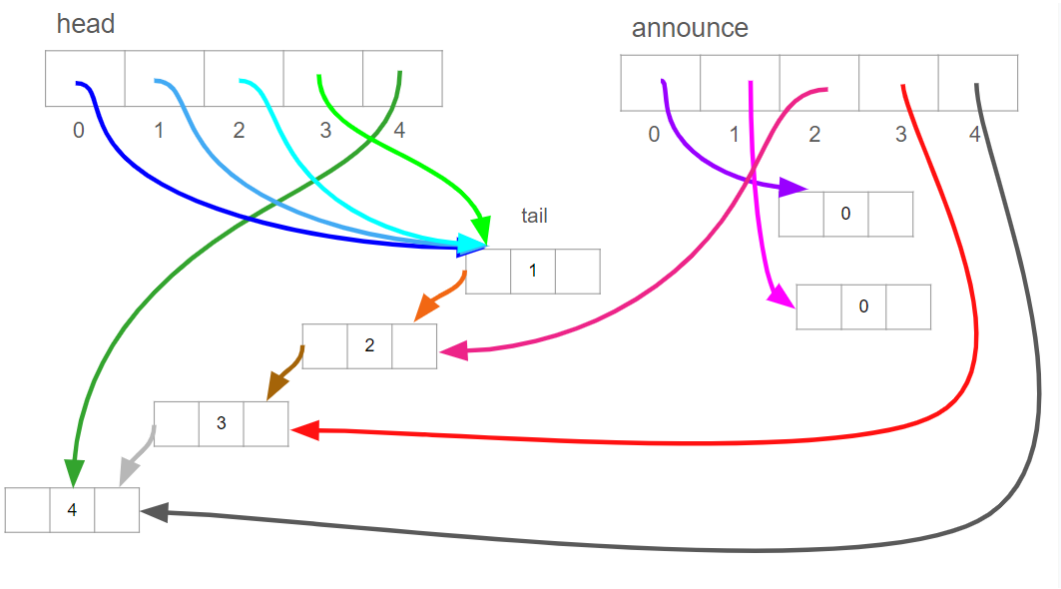
Luego de la primera iteración del while del hilo F tendremos esto, debido a que ayudo al hilo con indice 2 (el hilo C), ya que $before.seq = 1$ y $1 + 1\,\%5 = 2$



Después de la segunda iteración del while del hilo F tendremos esto, debido a que ayudo al hilo con indice 3 (el hilo D), ya que $before.seq = 2$ y $2 + 1 \,\%5 = 3$ y este seria el estado final de la lista cuando el hilo F termine su ejecución de apply, ya que no va a entrar una cuarta vez al while



Posteriormente de la tercera iteración del while del hilo F tendremos esto, debido a que ayudo al hilo con indice 4 (el hilo F, es decir el mismo hilo), ya que $before.seq = 3$ y $3 + 1 \,\%5 = 4$



Ejercicio 3

Para la implementación de la Lista de Grano Fino, código en las figuras 9.6 y 9.7 de [HS08]. ¿El algoritmo seguiría siendo linealizable y *starvation-free* si solo tomamos el candado de *curr* para añadir un elemento? Justifica tu respuesta.

Veamos que el algoritmo no sería linealizable y para ello, mostremos una ejecución que no podemos linealizar:

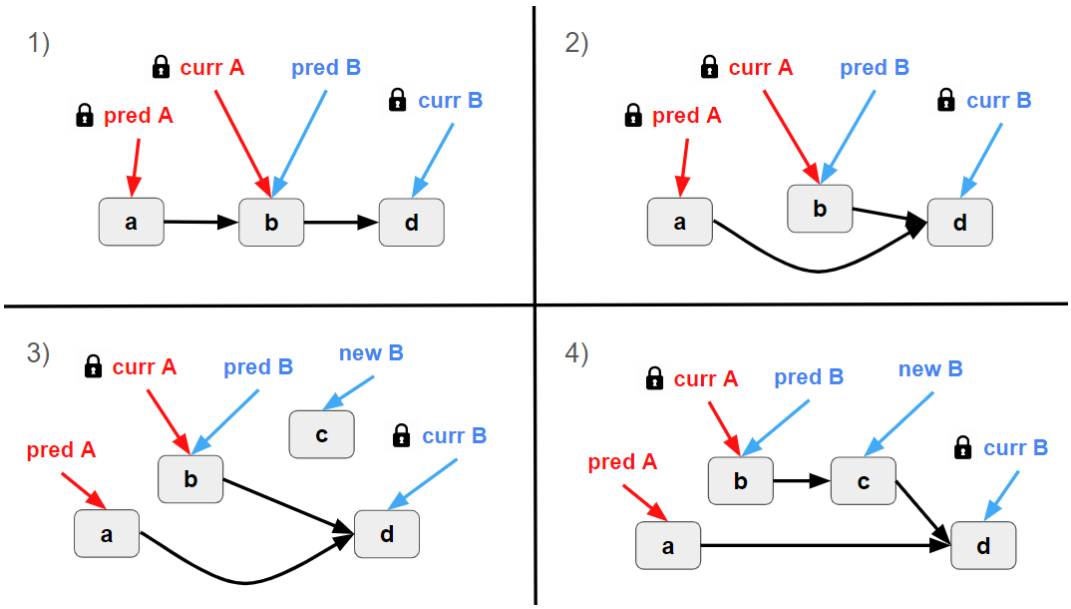
- Tendremos una lista con tres elementos, los cuales seran a, b y d, y tendremos los hilos A y B.
- Tendremos que el hilo A va a eliminar al elemento b y de manera concurrente el hilo B va a agregar al elemento c.

Entonces los hilos A y B van a empezar a recorrer los nodos de la lista. Notemos que el hilo A va a tomar los candado de `curr` y `pred` ya que va a eliminar un elemento, mientras que B solo va a tomar el candado de `curr`.

Digamos que llegamos al hilo A que tiene el candado de b como `curr` y el candado de a como `pred`, y para el hilo B tiene el candado de d como `curr` y como `pred` tiene a b, pero sin tomar su candado.

Entonces el hilo A realiza la eliminación, esto haciendo que ahora a apunte a d. Entonces inmediatamente el hilo B va a realizar su agregado, poniendo un nuevo nodo para c, el cual apunta a d y b apuntando a c. Después, todos los hilos liberan sus candados y terminan.

Notemos que el estado final de la lista no es una normal o que tenga sentido, ya que desde el nodo a (la cabeza) no podemos llegar al nodo c, lo cual no tiene sentido ya que si se debería poder. Por lo tanto, tenemos inconsistencias en los datos y esta ejecución no es puede linealizar, por lo que el algoritmo no sería linealizable.



Ahora notemos que el algoritmo seguiría siendo `starvation-free`, debido a que no tomar un candado no nos produce situaciones donde un hilo se queda atorado en algún método, entonces cada hilo que realice una operación eventualmente tendrá éxito, es decir va a terminar, tal vez termine con una lista toda rara, pero terminará.

Ejercicio 4

Para la implementación de la Lista Optimista, código en las figuras 9.11-9.14 de [HS08]. Si el método *contains* no toma ningún candado y solo regresa *true* si observa el *key* y $false$ sino. ¿El método *contains* seguiría siendo linealizable? Justifica tu respuesta, explica si es linealizable o da un contraejemplo si no lo es.

El método *contains* sí seguiría siendo linealizable:

- Si el método *contains* no es concurrente con ningún otro método (*add*, *remove* o *contains*), entonces es sencillo escoger un punto de linealización: cualquier punto durante la ejecución del método es válido.
- Si el método *contains* es concurrente con algún otro método, se presentan dos casos:

  1. **Caso 1:** El método *contains* regresa `true`.
     En este caso, sabemos que se cumple `curr.key == key`, lo que significa que durante la ejecución del método *contains* era posible recorrer la lista desde el nodo `head` hasta un nodo con el valor buscado (`key`). Por lo tanto, el punto de linealización puede ser el instante en el que este recorrido fue posible, es decir, cuando `curr.key == key` se cumplió.
  2. **Caso 2:** El método *contains* regresa `false`.
     En este caso, sabemos que no se cumple `curr.key == key`, lo que implica que no fue posible encontrar un nodo con el valor buscado desde `head`, ya sea porque dicho nodo no existe o porque fue eliminado antes de que *contains* pudiera observarlo. Por lo tanto, el punto de linealización puede ser el instante en el que esta condición (`curr.key == key` no se cumple) fue observada por el método *contains* o el momento en el que desde la cabeza no pudimos llegar al nodo con el elemento buscado o no existe ese nodo.

De esta manera, en ambos casos es posible identificar un punto de linealización del método *contains* que sea consistente con el resto de la ejecución. Por lo tanto, el método *contains* es linealizable, incluso si no toma ningún candado y no hace *validate*.

Ejercicio 5

Para la implementación de la Lista No bloqueante, código en las figuras 9.24-9.27 de [HS08]. Argumenta porqué no podría pasar que existan dos nodos con el mismo elemento "x", en donde uno de los nodos esté borrado de forma lógica. Describe todas las características en el algoritmo que previenen esto.

Para que una lista tuviera dos nodos con el elemento `x`, uno de ellos marcado como eliminado lógicamente, tendría que ocurrir lo siguiente: primero, la lista debe contener un nodo con el elemento `x`. Posteriormente, se tendría que realizar una operación *remove* seguida de una operación *add* del mismo elemento `x`. Veamos los casos que pueden presentarse en esta situación:

- **Caso 1: La línea 27 del método *remove* ocurre antes que la línea 4 del método *add*.**
  En este caso, el nodo con el elemento `x` se marca como eliminado lógicamente en la línea 27 de *remove* antes de que el método *add* ejecute la línea 4. En la línea 4 de *add*, se llama al método *find*, que, en las líneas 14 a 26, busca los nodos adyacentes al nodo con la clave `x`. Durante esta búsqueda, si encuentra un nodo marcado como eliminado lógicamente, lo elimina físicamente (líneas 16 a 21). Por lo tanto, el método *add* eliminará físicamente el nodo original con el elemento `x` antes de agregar uno nuevo. De esta manera, al finalizar ambas operaciones, la lista contendrá únicamente un nodo con el elemento `x`.
- **Caso 2: La línea 4 del método *add* ocurre antes que la línea 27 del método *remove*.**
  En este caso, el método *add* ejecuta *find* antes de que *remove* marque el nodo con el elemento `x` como eliminado lógicamente. Durante la ejecución de *find*, si el nodo actual (`curr`) tiene una clave igual a `x` (`curr.key == key`), el método *add* regresará `false` en las líneas 6 y 7, sin agregar un nuevo nodo. Posteriormente, el método *remove* marcará el nodo original con el elemento `x` como eliminado lógicamente. Por lo tanto, no se podrá tener más de un nodo con el elemento `x`.

En resumen, la situación planteada en la pregunta no puede ocurrir debido a las siguientes características del algoritmo:

- El método *add* no permite agregar nodos con elementos ya existentes. Esto se verifica en las líneas 6 y 7, donde se regresa `false` si el nodo actual (`curr`) ya contiene la clave buscada (`key`).
- Antes de agregar un nuevo elemento, el método *add* ejecuta *find*, el cual elimina físicamente cualquier nodo marcado como eliminado lógicamente (líneas 16 a 21).
- El uso de la operación `compareAndSet` asegura que las modificaciones sean consistentes, previniendo estados en los que existan múltiples nodos con la misma clave.

Por lo tanto, el diseño del algoritmo garantiza que no puedan existir dos nodos con el mismo elemento `x`, donde uno esté marcado como eliminado lógicamente.

Ejercicio 6

¿Porqué la lista No bloqueante, código en las figuras 9.24-9.27, es *lock-free*? Presenta un ejemplo de un hilo que no puede eliminar un nodo en un número finito de pasos. *Hint: Puedes suponer que una ejecución se repite infinitas veces en el tiempo.*

`Lock-free`, ya que algún hilo va a tener éxito, en el ejemplo donde un hilo no logra eliminar un nodo en un infinito número de pasos es causado debido a que en cada fallo de eliminar el nodo, otro hilo tuvo éxito o completo una acción/método (en una cantidad finita de pasos), la cual evito que se pudiera borrar el nodo. Veamos la ejecución donde un hilo no logra eliminar un nodo en un infinito número de pasos:

- Tendremos una lista con tres nodos, de esta manera nodo con elemento a seguido de nodo con elemento b y por último nodo con elemento c (ninguno de los nodos eliminados lógicamente), y tres hilos A, B y C.

El hilo A va a intentar eliminar el elemento b, entonces realizar el `find` de la línea 21 de `remove`, obteniendo que `pred` es el nodo con elemento a y `curr` es el nodo con elemento b, luego el hilo A ejecuta el código de `remove` hasta la línea 26 (antes de marcar a b como eliminado, lógicamente).
Entonces el hilo B va a eliminar al elemento b y tendrá éxito, por lo tanto marcara al nodo con elemento b como eliminado lógicamente.

Posteriormente el hilo A va a realizar la linea 27 y el `comparaAndSet` para intentar marcar el nodo falla, ya que el hilo B ya logró eliminarlo. Entonces el `remove` del hilo A se va a volver a intentar, por la línea 29.

Pero antes de que el hilo A pueda hacer la línea 21, el hilo C va a lograr agregar un nodo con el elemento b, volviendo al estado inicial de la lista (una lista con tres nodos, de esta manera nodo con elemento a seguido de nodo con elemento b y, por último un nodo con elemento c, ninguno de los nodos eliminados lógicamente). Por lo cual el hilo A va a volver a hacer el `find` encontrando que `pred` es el nodo con elemento a y `curr` es el nodo con elemento b, volverá a ejecutar el código hasta la línea 26. Luego el hilo B eliminará (marcará como eliminado lógicamente) al nodo con el elemento b y todo el ciclo se va a repetir.

De esta manera el hilo A va a seguir intentando eliminar al nodo con elemento b pero no tendrá éxito ya que el hilo B va a eliminarlo antes del `compareAndSet` del hilo A para que tenga que volver a hacer el `remove` (el `while`) y el hilo C va a agregar el elemento b para que el `find` si pueda encontrar un nodo con el elemento b, y de esta manera el hilo A nunca pueda terminar.

Entonces tenemos que es `lock-free` ya que usar referencias atómicas ayuda mucho a mantener consistencia y esto nos ayuda a que tengamos progreso, aunque solo sea de algunos hilos y no de todos.

Referencias

[1] Ayudantías y clases