Universidad Nacional Autónoma de México
Facultad de Ciencias

Computación Concurrente

Tarea 2

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

Problemas Clásicos

Ejercicio 1

Para cada uno de los siguientes enunciados identifica el tipo de propiedad e identifica:

- **La *cosa mala que nunca sucede* en caso de que sea una propiedad segura.**
- **La *cosa buena que sucederá* en caso de que sea una propiedad de viveza.**

(a) La inflación sube año con año

Propiedad de seguridad, la cosa mala que nunca sucede es "la inflación no sube cada año" o "la inflación no sube"

(b) Si Bob quiere entrar al patio a dejar comida, eventualmente entrará

Propiedad de viveza, la cosa buena que sucederá es "Bob entra al patio"

(c) Cada historia concurrente de esta implementación se puede transformar siempre a una implementación secuencial correcta

Propiedad de seguridad, la cosa mala que nunca es "una historia concurrente no se puede transformar a una implementación secuencial correcta"

(d) El candado se liberará en algún momento

Propiedad de viveza, la cosa buena que sucederá es "el candado se libera"

(e) El retraso de un proceso detendrá a otros eventualmente

Propiedad de viveza, la cosa buena que sucederá es "detener a otros"

(f) El retraso de un proceso nunca detiene a otros

Propiedad de seguridad, la cosa mala que nunca sucede es "detener a otros"

Ejercicio 2

Muestras un ejemplo de una ejecución de la implementación de `Filter` en la que un hilo A lo rebasa más de una vez cualquier hilo *B*, considera un número n de hilos mayor a 10. Explica por qué no se cumple la propiedad de Justicia. *Hint: Puedes mostrar una imagen de una ejecución o describirla.*

Imaginemos que tenemos 11 hilos, 12 niveles y todos inician con nivel 0:

```
Hilo A hace llamada de lock() entrando al nivel 1, victim[1] = A
Hilo B hace llamada de lock() entrando al nivel 1, victim[1] = B
Hilo A sube a nivel 2, victim[2] = A
Hilo C hace llamada de lock() entrando al nivel 1, victim[1] = C
Hilo A sube a nivel 3, victim[3] = A
Hilo B sube a nivel 2, victim[2] = B
Hilo B se duerme
Hilo C se duerme
Hilo A sube a nivel 4, victim[4] = A
Hilo A sube a nivel 5, victim[5] = A
...
Hilo A sube a nivel 11, victim[11] = A
Hilo A adquiere el candado y entra a la CS
Hilo A hace unlock(), entrando al nivel 0
Hilo A hace llamada de lock() entrando al nivel 1, victim[1] = A
Hilo D hace llamada de lock() entrando al nivel 1, victim[1] = D
Hilo A sube a nivel 2, victim[2] = A
Hilo C se despierta
Hilo C sube a nivel 2, victim[2] = C
Hilo A sube a nivel 3, victim[3] = A
Hilo A sube a nivel 4, victim[4] = A
...
Hilo A sube a nivel 11, victim[11] = A
Hilo A adquiere el candado y entra a la CS
Hilo A hace unlock(), entrando al nivel 0
```

Hilo solo puede subir al nivel $i+1$ si no es la víctima del nivel i o no hay nadie en niveles $\geq i$.

La propiedad de Justicia (Fairness) en los algoritmos de exclusión mutua es que en que los hilos que empiecen a ejecutar el candado antes que otros, toman primero el candado, es decir, que satisfacen FIFO. Notemos que el primero que pide el candado es el primero que lo obtiene y por eso si A rebasa a B significa que A lo pide primero y luego B, pero A lo tomó 2 veces, lo cual no es FIFO.

Ejercicio 3

Argumenta si la siguiente implementación (*VictimAround*) de un candado para dos hilos cumple con:

(a) **Exclusión mutua**

(b) **Deadlock-free**

(c) **Starvation-free**

Hint: Si no lo cumple, describe un ejemplo de una ejecución. Si lo cumple, encuentra la invariante y justifica por qué

```
class VictimAround implements Lock {
    Shared Variables:
    boolean busy = false;
    int me;

    void lock() {
        do {
            int me = ThreadID.get();
            do {
                victim = me;
            } while (busy == true);
            busy = true;
        } while(victim == me);
    }

    void unlock() {
        busy = false;
    }
}
```

(a) **Exclusión mutua:**

Recordemos que la exclusión mutua garantiza que solo un hilo puede estar en la CS a la vez. Para comprobar que esta propiedad se cumple, supongamos que ambos hilos A y B están en la CS al mismo tiempo y veamos si esto nos lleva a una contradicción.

Demostración. Por contradicción

Vamos a suponer que la exclusión mutua no se cumple, es decir, que dos hilos, digamos A y B, pueden estar en la sección crítica simultáneamente.

Primero veamos que para que el hilo A este en la sección critica tenemos que tener que `victim = B`, esto para que el `do-while` externo no se cumpla (al hacer que `victim == me` sea `false`) y el hilo A pueda terminar el método `lock()` y entrar a la sección critica.
Ahora, para que el hilo B este en la sección critica tenemos que tener que `victim = A`, esto para que el `do-while` externo no se cumpla (al hacer que `victim == me` sea `false`) y el hilo B pueda terminar el método `lock()` y entrar a la sección critica.
Notemos que esto genera una contradicción, ya que `victim` solo puede tener un valor a la vez, pero requerimos que `victim` sea A y B a la vez, para que ambos hilos entren a la sección critica. Por lo tanto podemos concluir que exclusión mutua se cumple. □

Dado que nuestra suposición llevó a una contradicción, concluimos que la implementación **cumple exclusión mutua**.

(b) **Deadlock-free:**

Esta propiedad no se cumple, veamos una ejecucion donde ambos hilos se quedaran congelados. Usaremos los hilos A y B

```
    Hilo A hace llamada de lock()
    Hilo A entra al do-while externo
    Hilo A hace int me = ThreadID.get();
    Hilo A entra al do-while interno
```

```
Hilo A hace victim = me; (victim = A)
Hilo A revisa si busy == true, obteniedo false y saliendo del do-while interno
Hilo B hace llamada de lock()
Hilo B entra al do-while externo
Hilo B hace int me = ThreadID.get();
Hilo A hace busy = true;
Hilo A revisa si victim == me, obtendiendo true y siguiendo en el do-while externo
Hilo A hace int me = ThreadID.get();
Hilo B entra al do-while interno
Hilo B hace victim = me; (victim = B)
Hilo B revisa si busy == true, obteniedo true y siguiendo del do-while interno
Hilo A entra al do-while interno
Hilo A hace victim = me; (victim = A)
Hilo A revisa si busy == true, obteniedo true y siguiendo del do-while interno
Hilo B hace victim = me; (victim = B)
Hilo B revisa si busy == true, obteniedo true y siguiendo del do-while interno
Hilo A hace victim = me; (victim = A)
Hilo A revisa si busy == true, obteniedo true y siguiendo del do-while interno
Hilo B hace victim = me; (victim = B)
Hilo B revisa si busy == true, obteniedo true y siguiendo del do-while interno
...
```

Como podemos observar los hilos A y B se quedaron atrapados en el `do-while` interno (el que tiene como condición `busy == true`) esto debido a que `busy = true` y la unica manera de cambia `busy` es con el método `unlock()`, por lo tanto los dos hilos se quedaron congelados intentando obtener el candado y al final ninguno lo pudo obtener.

Dado que encontramos que ambos hilos se congelaron en esta ejecución, concluimos que la implementación **no cumple deadlock-free** ya que los dos hilos intentaron adquirir el candado y ninguno lo logro adquirir.

(c) **Starvation-free:**

Considerando la información obtenida del inciso anterior, notemos que si una implementación no es `deadlock-free`, significa que existe al menos un escenario en el que uno o más hilos pueden quedar atrapados en un estado en el que no pueden avanzar (es decir, no pueden entrar a la CS ni salir de la espera).

Por otro lado, para que una implementación sea `starvation-free`, se requiere que cada hilo que trata de adquirir el candado eventualmente tiene exito.

Es decir, si una implementación no es `deadlock-free`, es posible que un hilo o un conjunto de hilos queden bloqueados indefinidamente. Si un hilo está atrapado en un estado de espera debido a un `deadlock`, no puede adquirir el candado. Esto implica que el hilo podría estar en una situación en la que nunca podrá avanzar, lo que claramente también viola la propiedad de starvation-free.

Por lo tanto, si una implementación no es `deadlock-free`, no puede ser `starvation-free`, ya que el `deadlock` impide que uno o más hilos avancen, lo que automáticamente lleva a que esos hilos experimenten inanición (`starvation`).

Concluimos que la implementación **no cumple starvation-free**.

Ejercicio 4

Nuevo problema de los Prisioneros. En una cárcel se encuentran n prisioneros y dos guardias. Los guardias jugarán un juego para liberarlos.

- **El guardia A lleva un prisionero a la vez a un cuarto E para alimentarlo. En este cuarto E existe un interruptor L_1. Este guardia pasa a cada prisionero al menos 1 vez cada $2n$ veces.**
- **El guardia B lleva un prisionero a la vez a un cuarto F para dejar que se bañe. En este cuarto F existe un interruptor L_2. Este guardia pasa a cada prisionero al menos 1 vez cada $3n$ veces.**
- **Los prisioneros se encuentran en un cuarto R en donde existe un interruptor L_3.**

Los guardias acuerdan liberarlos si uno de los prisioneros avisa "Ya todos estamos bañados y alimentados".

Los prisioneros no pueden hablar entre sí, solo se pueden poner de acuerdo antes de empezar el juego de los guardias. Suponemos que el estado inicial de todos los interruptores es `OFF`.

(a) **Diseña una solución para liberar a los prisioneros. Argumenta porque tu solución es correcta.**

Hint: No todos los prisioneros tienen que hacer lo mismo.

Para esta solución necesitamos que n sea 2 o mayor, si n es 1 se resuelve trivialmente.

Todos los prisioneros van a hacer lo mismo excepto dos, primero hablemos de estos dos prisioneros únicos. Para la solución habrá dos prisioneras que funcionaran como contadores (se puede hacer con un solo prisionero, pero tendría que llevar dos cuentas diferentes al mismo tiempo), llamemos al primer prisionero C, C tendrá la misión de contar cuantas prisiones ya fueron alimentados y al otro prisionero, que será D, tendrá la misión de contar cuantos prisioneros ya fueron bañados.

Ahora veamos que pasara cuando algún prisionero (diferente a los prisioneros C) entre al cuarto E, si al entrar al cuarto el interruptor L_1 está prendido entonces al momento de salir del cuarto, luego de comer, no interactuara con el interruptor, pero si al entrar al cuarto el interruptor L_1 está apagado entonces al momento de salir del cuarto, luego de comer, va a prender el interruptor L_1 si nunca lo ha prendido antes, en caso de que ya prendió el interruptor L_1 en algún viaje anterior al cuarto E entonces al salir no lo prende, en resumen, un prisionero (diferente a los prisioneros C y D) solo encenderá el interruptor L_1 si al entrar al cuarto E lo encuentra apagado y si nunca antes ha encendido el interruptor L_1. Luego, para los prisioneros (diferente a los prisioneros C y D) que entre al cuarto F, funcionara de la misma manera que para el cuarto E (lo que se explica arriba) pero para el cuarto F y el interruptor L_2, en resumen, un prisionero (diferente al prisionero D) solo encenderá el interruptor L_2 si al entrar al cuarto F lo encuentra apagado y si nunca antes ha encendido el interruptor L_2. Ahora veamos que pasa cuando el prisionero C entra al cuarto E, si al entrar al cuarto el interruptor L_1 está apagado entonces al momento de salir del cuarto, luego de comer, no interactuara con el interruptor L_1, pero si al entrar al cuarto el interruptor L_1 está prendido entonces al momento de salir del cuarto, luego de comer, va a apagar el interruptor L_1 y sumará 1 a su contador, en resumen el prisionero C solo apagará el interruptor L_1 si al entrar al cuarto E lo encuentra prendido y sumará 1 a su contador.

Por último veamos que pasa cuando el prisionero D entra al cuarto F, funcionara de la misma manera que cuando el prisionero C entra al cuarto E (lo que se explica arriba) pero para el cuarto F, el interruptor L_2 y prisionero D, en resumen el prisionero D solo apagara el interruptor L_2 si al entrar al cuarto F lo encuentra prendido y sumará 1 a su contador.

Ahora veamos como se liberaran, al inicio de todo los interruptores L_1, L_2 y L_3 están apagados y los contadores de los prisioneros C y D estarán en 0. Cuando el contador del prisionero C o D este en $n-1$, este prisionero va a prender el interruptor L_3, y cuando el contador del otro prisionero sea $n-1$ va a apagar el interruptor L_3 y dice "Ya todos estamos bañados y alimentados" (si el primero en llegar a $n-1$ fue el prisionero C entonces el prisionero D avisará de que terminaron, y si el primero en llegar a $n-1$ es el prisionero D entonces el prisionero C avisará). De esta manera, los prisioneros se podrán liberar.

Ahora veamos porque esto funciona. Notemos cuáles son las condiciones para que algún prisionero avise según nuestra solución, para esto necesitamos que se apague el interruptor L_3 y para que pueda suceder necesitamos que este prendido (su estado inicial es apagado), y para que este interruptor cambie de estado solo pasa cuando un contador llega a $n-1$, por lo tanto, como solo se incrementa en 1 el contador cuando los prisioneros elegidos apagan el interruptor del cuarto que les "asigno" (al prisionero C sería él E y para el prisionero D sería él F), pero ¿qué significa que apaguen $n-1$ veces el interruptor de su cuarto "asignado"?, lo que significa es los otros $n-1$ prisioneros ya pasaron al cuarto "asignado" (ya que los demás prisioneros solo pueden prender una vez el interruptor y solo el prisionero con contador puede apagarlo), entonces como el prisionero para apagar el interruptor tiene que pasar al cuarto significa que ya los n prisioneros pasaron al cuarto y fueron alimentos o bañados (dependiendo de que cuarto sea).

Por lo tanto, si el estado del interruptor L_3 cambio dos veces, significa que n prisioneros ya pasaron al cuarto E y F, por lo tanto, ya cumplen con lo necesario para ser libres. Funciona gracias a que dos prisioneras mantienen la cuenta de cuantos prisioneros y van aumentando estos contadores, aunque aumentar los contadores puede tomar muchos ciclos, para el cuarto E aumenta el contador en 1 cada $4n$ prisioneros que pasaron (ya que en el peor caso el prisionero tiene que pasar dos veces para poder encontrar la luz prendida) y para el cuarto F aumenta el contador en 1 cada $6n$ prisioneros que pasaron (mismo razonamiento que para el cuarto E), por lo que va a tomar una complejidad de $O(n^2)$ para que todos los prisioneros estén alimentados y bañados. Así que esta solución es válida, solo toma mucho tiempo.
Por último veamos una demostración de que esta solución sirve.

Demostración. Por inducción sobre n

- **Caso base:**

  Usaremos a $n = 2$ (si usamos $n = 1$, trivialmente se resuelve este problema), entonces el primer prisionero sea C y el otro D, por lo tanto, cuando el prisionero D pase al cuarto E va a prender el interruptor L_1, luego cuando el prisionero C pase al cuarto E apagara el interruptor y su contador estará en 1 por lo que ya termino de contar, como para que un prisionero pase al cuarto E es cada $2n$ visitas, entonces a lo más tomara 8 visitas al cuarto E para que el prisionero C cuente la cantidad necesaria (en el peor caso es el primero en pasar por lo que para volver a pasar tiene que esperar a lo más 4n-1, que sería 7 y por eso toma 8 visitas), ahora cuando el prisionero C pase al cuarto F va a prender el interruptor L_2, luego cuando el prisionero D pase al cuarto F apagará el interruptor y su contador estará en 1 por lo que terminara de contar, y esto tomará a lo más 12 visitas al cuarto F (mismo análisis que arriba solo que ahora son a lo más 6n-1 visitas, si el prisionero D pasa primero). De esta manera luego de a lo más 12 visitas al cuarto F y 8 visitas al cuarto E, los dos prisioneros ya tendrán 1 en sus contadores y, por lo tanto, el primero en llegar a uno prenderá el interruptor L_3 y cuando el otro prisionero llegue a 1 en su contador avisará "Ya todos estamos bañados y alimentados", logrando liberar a los prisioneros, por lo tanto, esta solución funciona si $n = 2$

- **Hipótesis de inducción:**

  Supongamos que esta solución cuando $n = k$

- **Paso inductivo:**

Veamos que la solución sirve cuando $n = k + 1$, por hipótesis de inducción sabemos que la solución funciono para $n = k$, por lo tanto, para que eso pasara significa que el contador del prisionero C está en $k - 1$ y el del prisionero D está en $k - 1$, ahora tenemos k prisioneros, entonces se repite la solución usada en la hipótesis y llegamos a que los prisioneros C y D tienen $k-1$ en sus contadores lo que significa que han contado a $k - 1$ prisioneros, por lo tanto, falta contar 1 prisionera a cada uno. Entonces notemos cuantas visitas a cada cuarto tomará esto (como se realizó en el caso base), para el prisionero C tienen que pasar a lo más $4k$ visitas al cuarto E para contar a la última persona, esto debido a que si es el primero en pasar entonces solo volverá a pasar (en el peor de los casos) luego de $4k - 1$ visitas (ya que al menos cada prisionero pasa cada $2n$), entonces luego de a lo más $4k$ visitas (luego que el contador este en $k - 1$) podrá tener su contador en k y terminar. De una manera muy similar pasa para el prisionero D solo que para él tomara a lo más $6k$ visitas (luego que el contador este en $k - 1$) para tener su contador en k y terminar. Por lo tanto, cuando el primero de los dos que llegue a k en su contador, prenderá el interruptor L_3 avisando que ya termino y cuando el otro llegue también a k en su contador, apagará el interruptor y avisará "Ya todos estamos bañados y alimentados", siendo liberados. Todo esto gracias a que cuando $n = k$ solo contaron $k-1$ prisioneros y como para pasar a contar se tiene que ir al cuarto E y F entonces ya pasaron a los k prisioneros, y si cuentan una vez más (apagando el interruptor del cuarto E o F dependiendo del prisionero C o D) entonces habrán contado k prisioneros y como tuvieron que pasar al cuarto para contar, significa que ya pasaron los $k + 1$ prisioneros a comer y bañarse, de esta manera esta solución sirve para $n = k + 1$.

Y por le principio de inducción podemos concluir que esta solución si sirve.

□

Ejercicio 5

Peterson es una implementación de un candado para dos hilos, en la implementación siguiente (DoublePeterson) se utilizan tres candados Peterson para realizar una implementación de un candado para 4 hilos. Recuerda que en el candado de Peterson se cumplen las propiedades de exclusión mutua, Deadlock-free y Starvation-free. Argumenta si cumple con:

- **Exclusión mutua**
- **Deadlock-free**
- **Starvation-free**
- **Justicia**

Hint: Si cumple con la propiedad, muestra la invariante que se mantiene (en el caso de la exclusión mutua y justicia), o muestra la cosa buena que pasará (en el caso de Deadlock-free y Starvation-free). No hace falta que demuestres Peterson. Si no cumple con alguna propiedad, muestra una ejecución que lo ejemplifique.

```
1 class DoublePeterson implements Lock {      /* Candado para n = 4 hilos */
2    Shared Variables:
3    Lock lock1, lock2, lock3;
4    DoblePeterson() {
5     lock1 = petersonLock();
6     lock2 = petersonLock();
7     lock3 = petersonLock();
```

```
8     }
9     void lock() {
10     int me = ThreadID.get();     /* cada hilo tiene un unico ID de 1 a 4 */
11     if (me == 1 or me == 2) {
12       lock1.lock()                    /* los hilos 1 y 2 toman el candado 1 */
13     if (me == 3 or me == 4) {
14       lock2.lock()                    /* los hilos 3 y 4 toman el candado 2 */
15     lock3.lock();                      /* todos pueden tomar el candado 3 */
16    }
17
18    void unlock() {
19     int me = ThreadID.get();
20     lock3.unlock();
21     if (me == 1 or me == 2) {
22       lock1.unlock()
23     if (me == 3 or me == 4) {
24       lock2.unlock()
25    }
26 }
```

- **Exclusión mutua:** Veamos que DoublePeterson cumple con exclusión mutua.

  Observamos que es un sistema en el que los hilos compiten por los candados en dos grupos separados. Los hilos 1 y 2 compiten por el candado 1 mientras que los hilos 3 y 4 compiten por el candado 2 mediante un Peterson Lock, lo que asegura que solo un hilo pueda adquirir su respectivo candado. Posteriormente, cada hilo ganador compite por el tercer candado, por lo que solo puede haber un hilo ganador que accede a la sección crítica, cumpliendo el principio de Exclusión mutua.

  Invariante: Dos grupos de hilos compiten por diferentes candados (lock 1 y 2) y solo el ganador de cada grupo puede competir por lock 3, lo que garantiza que solo un hilo accede a la sección crítica en un momento dado.

  Demostración por contradicción: Supongamos que en la ejecución, los hilos 1 y 2 logran acceder a la sección crítica simultáneamente, lo que violaría al principio de exclusión mutua, sabemos que usamos un Peterson Lock para determinar el ganador entre los hilos, por lo que esto querría decir que Peterson Lock no cumple con el principio de exclusión mutua, lo cual es una contradicción, ya que sabemos que un Peterson Lock cumple con las propiedades de exclusión mutua. Por lo tanto, nuestro DoublePeteson cumple con exclusión mutua.

- **Deadlock-free:** Veamos la cosa buena que pasará.

  Los hilos solo pueden avanzar y adquirir lock 3 después de haber adquirido con éxito lock 1 o lock 2, lo que rompe cualquier posible ciclo de espera y **garantiza que al menos un hilo pueda avanzar y acceder a la sección crítica** que es la cosa buena que pasará. Ya que esto asegura que el progreso del sistema no se vea obstaculizado por situaciones de deadlock.

- **Starvation-free:** Veamos la cosa buena que pasara.

  Se evita la situación en la cual un hilo queda bloqueado indefinidamente, sin poder acceder a la sección crítica protegida por **lock3**. Al garantizar que cada hilo tenga la oportunidad de adquirir sus candados respectivos antes de intentar acceder al candado 3, **se previene la posibilidad de que algún hilo sea negado indefinidamente el acceso**, lo cual promueve la equidad en el sistema y evita la inanición de recursos para cualquier hilo.

- **Justicia:** Veamos que DoublePeterson no cumple con la propiedad de justicia.

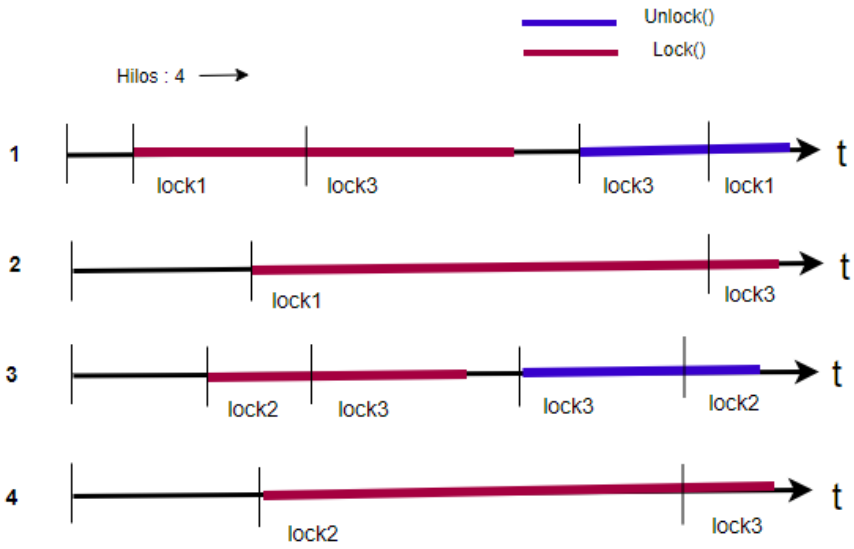

Figura 1: Violación de la propiedad de justicia

Veamos que no hay garantía que si un hilo llama primero a lock() antes sea siempre el primero en entrar a la sección crítica, veamos una ejecución:

Como podemos ver el orden al que llega sería $1 \to 3 \to 2 \to 4$, en ese orden aplican **lock** en su respectivo candado, pero veamos que cuando los hilos 1 y 3 hacen **lock3**, puede pasar que el hilo 3 entre antes a la sección crítica y haga **unlock** antes que el hilo 1, que llego antes que el hilo 3, por lo que no se cumple con el orden **FIFO**.

Referencias

Ayudantias y clase de la semana del 26 a 30 de agosto