Computación Concurrente - Tarea 2

Damián Rivera González
Alexis Hernandez Castro

September 18, 2019

1. 1.Considera el algoritmo de Peterson:

```
1. public void lock() {
2.     int i = ThreadID.get();
3.     int j = 1 - i;
4.     flag[i] = true;
5.     victim = i;
6.     while (flag[j] && victim == i) {};
7. }
```

a) Supón que intercambias las líneas 4 y 5. ¿El algoritmo sigue cumpliendo con las propiedades de exclusión y no deadlock? Demuestra o da una ejecución en donde no se cumpla.

- El algoritmo ya NO cumple con la exclusión mutua, ya que existe la siguiente ejecución como contraejemplo: (Suponemos que ambas banderas se inicializan como $false$. Damos por hecho que ambos hilos han ejecutado las primeras tres líneas de código)

$H1$	$H2$	Memoria
-	write(victim = B)	victim = B
write(victim = A)	-	victim = A
write(flag[A] = true)	-	victim = A, f[A]=T
read((flag[B]) = false and (victim == A) = true)	-	victim = A, f[A]=T
-	write(flag[B] = true)	victim = A, f[A]=T, f[B]=T
-	read((flag[A]) = true and (victim == B) = false)	victim = A, f[A]=T, f[B]=T
//CS	//CS	victim = A, f[A]=T, f[B]=T

- Sí cumple con No-Deadlock.
  Demostración.[Por contradicción] Supongamos que cumple con Deadlock, entonces existe una ejecución en la que se cumple que $Lock_A \nrightarrow Lock_B$ además que $Lock_B \nrightarrow Lock_A$. Entonces sabemos que:

$$1) read_A(flag[B] == true AND victim == A) \rightarrow Lock_A$$

$$2) read_B(flag[A] == true AND victim == B) \rightarrow Lock_B$$

  Supongamos sin perdida de generalidad que ocurre $Lock_A$. Entonces tuvo que ocurrir 1) así

$$read_B(flag[A] == false) OR read_B(victim! = B)$$

  en particular, tenemos que se cumple

$$read_B(victim! = B)$$

  ya que A leyó que victim == A.
  Por lo tanto al menos un hilo logra acceder a la sección crítica.

b) Supón ahora que la instrucción de la línea 6 se cambia por:

$$while(flag[j] \;\&\&\; victim == j)$$

Da un ejemplo en donde se cumpla la exclusión mutua y otro en donde no.

- Se cumple la exclusión mutua

$H1$	$H2$	Memoria
write(victim = A)	-	victim = A
write(flag[A] = true)	-	victim = A, f[A]=T
read((flag[B]) = false and (victim == B) = false)	-	victim = A, f[A]=T
//CS	-	victim =A, f[A] = T
write(flag[A] = false)	-	victim = A, f[A] = F
-	write(victim = B)	victim = B, f[A] = F
-	write(flag[B] = true)	victim = B, f[A]=F, f[B]=T
-	read((flag[A]) =false and (victim == A) = false)	victim = B, f[A]=F, f[B]=T
-	//CS	victim = B, f[A]=F, f[B]=T
-	write(flag[B] = false)	victim = B, f[A] =F, f[B] =F

- No se cumple la exlusión mutua

$H1$	$H2$	Memoria
write(victim = A)	-	victim = A
-	write(victim = B)	victim = B
write(flag[A] = true)	-	victim = B, f[A]=T
read((flag[B]) = false and (victim == B) = true)	-	victim = B, f[A]=T
-	write(flag[B] = true)	victim = B, f[A]=T, f[B]=T
-	read((flag[A]) = true and (victim == A) = false)	victim = B, f[A]=T, f[B]=T
//CS	//CS	victim = B, f[A]=T, f[B]=T

2. Considera el siguiente algoritmo para dos procesos:

Algoritmo para Alice:

1. Levanta su bandera
2. Espera hasta que la bandera de Bob este abajo
3. Libera la mascota
4. Baja la bandera cuando la mascota regresa

Algoritmo para Bob:

1. Levanta su bandera
2. Mientras la bandera de Alice este arriba:
   1) Baja su bandera.
   2) Espera hasta que la bandera de Alice este abajo
   3) Levanta su bandera
3. En cuanto la bandera de Alice está abajo y la suya arriba, libera su mascota
4. Baja su bandera cuando la mascota regresa.

Traducimos el algoritmo a código:
//Alice
flag[A] = true
while(flag[B]);
//CS
flag[A] = flase

```
//Bob
flag[B] = true
while(flag[A]){
flag[B] = false
while(flag[A]);
flag[B] = true;
}
while(flag[A]);
//CS
flag[B] = flase
```

a) Demuestra que resuelve el problema de la exclusión mutua.
Por contradicción. Suponemos que existe una ejecución en la cual ocurre que $CS_A \nrightarrow CS_B$ y que $CS_B \nrightarrow CS_A$ Tenemos entonces que:

$$1) write_A(flag[A] = true) \rightarrow read_A(flag[B] == flase) \rightarrow CS_A$$
$$2) write_B(flag[B) = true) \rightarrow read_B(flag[A] == false)$$
$$\rightarrow read_B(flag[A] == flase) \rightarrow CS_B)$$

Revisaremos dos casos, sobre el orden de la última escritura

Caso 1 Bob escribe al último
Entonces:

$$3) write_A(flag_A = true) \rightarrow write_B(flag[B] = true)$$

por transitividad de $\rightarrow$ con 3) y 2) tenemos

$$4) write_A(flag[A] = true) \rightarrow read_A(flag[A] == false)$$

por lo que es una contradicción

Caso 2 Alice escribe al último
Entonces tenemos:

$$5) write_B(flag[B] = true) \rightarrow write_A(flag[A] = true)$$

tenemos dos subcasos en la ejecución de B:

Caso 2.1 B entra en el primer while
Si entra en el while tenemos:

$$6) read_B(flag[A] == true) \rightarrow write_B(flag[B] = false)$$
$$\rightarrow read_B(flag[A] == false)$$
$$\rightarrow wrtie_B(flag[B] = true)$$

Entonces por 6) y 5) sabemos que entre los siguientes eventos no ocurre ningun otro evento

$$7) write_B(flag[B] = true) \rightarrow write_A(flag[A] = true)$$

por la transitividad de $\rightarrow$ entre 7) y 1) tenemos

$$write_B(flag[B] = true) \rightarrow read_A(flag[B] == flase)$$

lo cual es una contradicción

Caso 2.2 Bob no entra en el primer while
Entonces sabemos que no existe un escritura de flag[B] entre los dos eventos, siendo así, por transitividad de $\rightarrow$ entre 5) y 1) tenemos

$$8) write_B(flag[B] = true) \rightarrow read_A(flag[B] == false)$$

lo cual es una contradicción

Por lo tanto vemos que la suposición es incorrecta.
Por lo tanto si se resuelte el problema de la exlusión mutua.

b) Demuestra por qué no cumple con la propiedad libre de hambruna.
Basta con mostrar la siguiente ejecución:

Alice	*Bob*	Memoria
write(flag[A] = true)	write(flag[B] = true)	flag[A] = T, flag[B] = T
read(flag[B] == true)	read(flag[A] == true)	flag[A] = T, flag[B] = T
... esperando	write(flag[B] = false)	flag[A] = T, flag[B] = F
//CS	.. esperando	flag[A] = T, flag[B] = F
write(flag[A] = false)	-	flag[A] = F, flag[B] = F
write(flag[A] = true)	write(flag[B] = true)	flag[A] = T, flag[B] = T
read(flag[B] == true)	read(flag[A] == true)	flag[A] = T, flag[B] = T
... esperando	write(flag[B] = false)	flag[A] = T, flag[B] = F
//CS	.. esperando	flag[A] = T, flag[B] = F

Podemos repetir los pasos justo después de que el Alice escribe su bandera como falso, y Bob nunca entraría en la sección crítica.

c) El algoritmo cumple con la propiedad de no deadlock?
Sí, ya que al menos Alice pasará a la sección crítica

d) Cuál es la desventaja del algoritmo? Argumenta por qué es difícil de generalizar.
Las diferentes cerificaciónes mediante while's y también el cambio de valor de una variable en diferentes ocasiones. No se puede generalizar ya que Bob depende exclusivamente del otro hilo, así como Alice, por lo que al generalizar para más hilos crear la dependencia entre los hilos para que funcionen correctamente los whiles, se terminaría con demasiadas verificaciones de while para algún k hilo.

3. Supón que tienes la instrucción swap(A,B) que intercambia de manera atómica los valores de las variables A,B.

a) Da una propuesta de codificación para un objeto Look que resuelva el problema de la exclusión mutua para dos procesos.

```
flag[2] = {false, false}
flagTrue[2] = {true, true}
lock(){
    swap(flag[ID], flagTrue[ID]) // Hacemos verdadera la bandera
    victim = ID
    while(flag[1-ID] && victim == ID);
}

//Sección crítica

unlock(){
    swap(flag[ID], flagTrue[ID]) // Hacemos falsa la vandera
}
```

b) Demuestra que tu algoritmo cumple con la propiedad de exclusión y no deadlock. Dado que es prácticamente el mismo algoritmo que el de Peterson, ya sabemos que este cumple con ambas propiedades

4. Considera la siguiente modificación de la clase contador, que es ejecutada por dos procesos concurrentemente cuyos IDś son (0 y 1):

```
public class Counter implements Runnable {
    private int counter = 0;
    private static final int rounds = 10;
    private boolean wantToEnter[] =  { false, false };

    public void run() {
        long ID = Thread.currentThread().getId();
        for(int i = 0; i < Counter.rounds; i++) {
            wantToEnter[ID] = true;
            while(wantToEnter[1 - ID]) {
                wantToEnter[ID] = false;
                wantToEnter[ID] = true;
            }
            this.counter++;
            wantToEnter[ID] = false;
        }
    }
}
```

a) Muestra una ejecución en donde los hilos llegan a un estado de deadlock.
Damos por hecho que ambos hilos ejecutan todas las instrucciones hasta el punto en que entran al for, entonces tenemos. (wte = wantToEnter)

0	1	Memoria
write(wantToEnter[0] = true)	write(wantToEnter[1] = true)	wte[0] = T, wte[1] = T
read(wantToEnter[1]==true)	read(wantToEnter[0]==true	wte[0] = T, wte[1] = T
write(wantToEnter[0] = false)	write(wantToEnter[1] = false)	wte[0] = F, wte[1] = F
write(wantToEnter[0] = true)	write(wantToEnter[1] = true)	wte[0] = T, wte[1] = T
read(wantToEnter[1]==true)	read(wantToEnter[0]==true	wte[0] = T, wte[1] = T
write(wantToEnter[0] = false)	write(wantToEnter[1] = false)	wte[0] = F, wte[1] = F
write(wantToEnter[0] = true)	write(wantToEnter[1] = true)	wte[0] = T, wte[1] = T
.	.	.
.	.	.
.	.	.

Por lo que ambos hilos desean entrar a la sección crítica, sus variables se actualizan al mismo tiempo, entonces ninguno de los dos entraría nunca a la sección crítica.

b) Demuestra que el algoritmo no tiene una condición de carrera, es decir, toda ejecución que termina siempre imprime el mismo resultado.

Basta con mostrar que el algoritmo cumple con exclusión mutua.
Por contradicción. Supongamos que existe una ejecución en la que se cumple que $CS_0 \nrightarrow CS_1$ y $CS_1 \nrightarrow CS_0$

$$1) write_0(wantToEnter[0] = true) \rightarrow read_0(wantToEnter[1] == false) \rightarrow CS_0$$

$$2) write_1(wantToEnter[1] = true) \rightarrow read_1(wantToEnter[0] == false) \rightarrow CS_1$$

Suponemos sin perdida de generalidad que el hilo 0, entra en el while, entonces:

$$3) write_1(wantToEnter[1] = true) \rightarrow read_0(wantToEnter[1] == true)$$
$$4) write_0(wantToEnter[0] = true) \rightarrow read_0(wantToEnter[1] == true)$$
$$5) read_0(wantToEnter[1] = true) \rightarrow write_0(wantToEnter[0] == true)$$
$$6) write_0(wantToEnter[0] = false) \rightarrow write_0(wantToEnter[0] = true)$$

Entonces por 5) y 1) tenemos

$$6) read_0(wantToEnter[1] = true) \rightarrow read_0(wantToEnter[1] == false)$$

Por lo tanto es una contradicción.
Por lo que el algoritmo cumple con la exclusión mutua.
Por lo tanto cada hilo entra a la sección crítica independientemente, con lo cual $this.counter++$ es leida y modificada por cada hilo, con lo cual el resultado será siempre el mismo.

5. ¿Porqué requerimos definir una sección de entrada (doorway), porqué no podemos definir un algoritmo de exclusión mutua que cumpla con la propiedad FCFS basado en el orden en el que los hilos ejecutan la primer instrucción del método lock? Argumenta tu respuesta caso por caso según la naturaleza de la primera instrucción ejecutada por el método lock(): una lectura o una escritura, a registros separados o a un mismo registro.
Respuesta:

Cualquier algoritmo de exclusión mutua consiste en cómputo y espera. Al dividir el código en una sección de puerta seguida de una sección de espera, movemos todo el cálculo al frente (en lugar de intercalar el cálculo y la espera). Esto impone una estructura limpia al algoritmo y facilita el reconocimiento y la eliminación del bloqueo innecesario (es decir, el bloqueo durante la sección de la puerta).

El problema con la definición de FCFS en términos del primer paso dado por la puerta es que generalmente es imposible saber quién dio el primer paso. Considere los siguientes casos de dos hilos:

- El primer paso es una lectura. Si los dos hilos leen uno inmediatamente después del otro, es imposible saber cuál fue primero.
- El primer paso es escribir y dos hilos escriben en diferentes ubicaciones. Si los dos hilos escriben uno justo después del otro, es imposible saber cuál fue primero.
- El primer paso es una escritura, y ambos hilos escriben en la misma ubicación. En un escenario, B escribe, A escribe, por lo que FCFS requiere que B ingrese primero a la sección crítica. En otro escenario, A solo escribe, por lo que FCFS requiere que A ingrese primero a la sección crítica. El problema es que A no puede distinguir entre estas dos situaciones, ya que su escritura borró cualquier evidencia de que B pudo haber escrito primero.

6. Muestra que el algoritmo del filtro permite que algunos hilos superen a otros un número arbitrario de veces.

   Respuesta:
   Dados tres hilos A, B y C, C puede adelantar a A un infinito numero de veces.

   1. C adquiere la cerradura.
   2. Un bloqueo de llamadas (), convirtiéndose en la víctima en el nivel 1.
   3. B llama a lock (), convirtiéndose en la víctima en el nivel 1.
   4. C libera el bloqueo, vuelve a llamar al bloqueo () y se convierte en la víctima en el nivel 1.
   5. B adquiere y libera la cerradura.
   6. B llama a lock (), convirtiéndose en la víctima en el nivel 1.
   7. C adquiere la cerradura.
      Esto continúa hasta que A se despierte y siga adelante.

7. Supón que en el algoritmo del filtro se intercambia la operación $\geqslant$ por $>$ en la condición:

   $$while((Ek\ !=\ \text{me})\ (level[k] > i\ \&\&\ victim[i] == me))$$

   Demuestra o da un contraejemplo si cumple con las propiedades de exclusión, no-deadlock, libre hambruna, FCFS.

8. Una forma de generalizar el algoritmo de Peterson para n hilos (supón que n es potencia de 2) es utilizar varios candados de 2 hilos en un árbol binario. A cada hilo es asignado un candado en una hoja que comparte con otro hilo.Cada candado trata a cada hilo como hilo 0 o hilo 1. En el método de adquisición del árbol de candados (como el método lock() en Peterson), el hilo adquiere cada candado de Peterson de la hoja de ese hilo a la raíz (camino de la hoja de ese hilo a la raíz). El método que libera al candado (como el unlock() en Peterson) para el árbol de candados, desbloquea cada candado de Peterson que el hilo haya adquirido desde la raíz hasta su hoja. En cualquier momento un hilo puede retrasarse por un tiempo finito. Para cada propiedad, da un esbozo de prueba que la propiedad se cumple o describe una ejecución posiblemente infinita donde se viola:

   - Exclusión mutua
   - Libre de deadlok
   - Libre de hambruna

   ¿Hay una cota superior en el número de veces que el árbol de candados puede ser adquirido o liberado entre el tiempo en el que un hilo empieza el adquirir el candado y cuando lo logra?
   Respuesta:

   1. El algoritmo satisface la exclusión mutua.
      Si $n = 1$ (base) (n es una potencia de 2)
      $2^1 = 2 \rightarrow$ 2 hilos y 1 hoja
      Para una hoja, se utiliza el algoritmo Peterson, que tiene exclusión mutua.

Si $n = 2$ (n es una potencia de 2)

$2^2 = 4 \rightarrow 4$ hilos y 3 hojas

Puede dividir el árbol en dos subárboles iguales con 2 hilos y 1 hoja(mismo caso base).

Como cada hoja tiene dos hilos, usando el algoritmo Peterson, podemos inferir que la solución tiene exclusión mutua para $n = 2$.

Si $n = k$ (n es una potencia de 2)

$2^k \rightarrow 2^k$ hilos y $(2^k) - 1 hojas$

Puede dividir el árbol por $2^{(k-1)}$ subárboles que son idénticos al caso base. Con esto obtenemos exclusión mutua, para n = k, dados los casos anteriores.(divide y conquistaras)

2. Libre de deadlock.

   El algoritmo satisface sin bloqueo.

   Similar a la evidencia previa. Como el algoritmo Peterson satisface el punto muerto para el caso base.

   Y el caso donde $n = k$ puede reducirse al caso base, donde tenemos varios subárboles idénticos.

   El algoritmo de bloqueo basado en árbol también está libre de puntos muertos.

3. Libre del Hambruna.

   El algoritmo satisface libre de hambruna. Similar al punto anterior. Además, establece en la declaración que un hilo puede retrasarse por un período finito, sin embargo, un hilo no "muere".

9. Supón que n hilos ejecutan el mtodo visit() de la siguiente clase:

```
class Bouncer {
    public static final int DOWN = 0;
    public static final int RIGHT = 1;
    public static final int STOP = 2;
    private boolean goRight = false;
    private ThreadLocal<Integer> myIndex;
    private int last = -1;

    int visit() {
        int i = myIndex.get();
        last = i;
        if (goRight) return RIGHT;

        goRight = true;
        if (last == i) return STOP;
        else return DOWN;
    }
}
```

   a) A lo más un hilo obtiene el valor STOP

   b) A lo más n - 1 hilos obtienen el valor DOWN

   c) A lo más n - 1 obtienen el valor RIGHT

Respuesta:
Debemos mostrar qué, como máximo, un subproceso obtiene el valor STOP. El punto clave es que el último puede ser el ID de, como máximo, uno de los hilos que lee goRight + para que sea falso en la declaración if. Podemos concluir esto porque sabemos:

$$last.write(i) \rightarrow goRight.read(false) \rightarrow goRight.write(true) \rightarrow last.read(i) \rightarrow return(STOP)$$

Para que dos hilos A y B hayan devuelto STOP, habrían tenido que leer sus identificadores en último lugar. Sin pérdida de generalidad, digamos que B leyó por última vez después de que A leyó su valor allí.

$$A : last.read(A) \rightarrow B : last.read(B)$$

Se deduce que B escribió su valor en último después de que A había escrito y leído su valor allí (ya que A debe haber escrito su valor para haberlo leído):

$$A : last.write(A) \rightarrow A : last.read(A) \rightarrow B : last.write(B) \rightarrow B : last.read(B)$$

Pero de arriba sabemos qué:

$$A : last.write(A) \rightarrow A : goRight.write(true) \rightarrow A : last.read(A)$$

y

$$B : last.write(B) \rightarrow B : goRight.read(false) \rightarrow B : last.read(B)$$

Uniendo hambas cosas tenemos qué:

$$A : last.write(A) \rightarrow A : goRight.write(true) \rightarrow A : last.read(A) \rightarrow B : last.write(B) \rightarrow B : goRight.read(false) \rightarrow B : last.read(B)$$

Dado que es imposible que cualquier hilo escriba false en goRight, no hay forma que el valor de goRight podría haberse vuelto falso una vez que A lo escribió, contradiciendo B leyó de falso. Por lo tanto, más de un hilo no puede obtener el valor STOP. Aquí están los pasos en la prueba.

Probar como máximo los subprocesos $n - 1$ obtienen el valor DOWN.

Para que cualquier hilo obtenga el valor DOWN, vemos que deben haber leído GoRight para ser falso (para que no puedan regresar RIGHT). Asume n hilos pudimos hacer esto. Luego de haber vuelto DOWN, todos los hilos han tenido que haber leído por última vez para ser desiguales a su valor. Pero como todo n los subprocesos ya completaron la línea donde establecieron el último en su ID no quedan otros hilos para cambiar el último valor. Sabemos que algunos el subproceso debe haber sido el último en establecer el último en su ID, por lo que debe leer esto y devolver STOP, contradiciendo nuestra suposición de que n hilos eran capaz de regresar DOWN.

Probar que como máximo, los subprocesos n-1 obtienen el valor RIGHT.

En primer lugar, vemos las siguientes restricciónes para regresar a RIGHT:

$$goRight.read(true) \rightarrow return(RIGHT)$$

También notemos que goRight se inicializa en falso y que no hay escritura en goRight antes de que sea leída por el primer hilo:

$$goRight.write(false) \rightarrow goRight.read(false) \rightarrow goRight.write(true)$$

Entonces, el primer subproceso para verificar el valor de goRight debe encontrar que es falso, haciendo que se salte la línea para regresar a RIGHT. Por lo tanto, todos los n hilos no pueden obtener el valor RIGHT ya que el primero no puede.