UA

ProgramaciónConcurrente

DanielAsensiRochDNI : **48776120C**

27 de octubre de 2022

Índice

1. Apartado 1: C

1.1. Intento 1: Alternancia

Se puede apreciar en la traza obtenida por el código como los hilos 1 y 2 se **alternan**, esto es debido a al uso de la variable denominada en el código como *turno*. Esto puede derivar en ciertos problemas como la conocida como **condición de progreso**, en este caso para el hilo 2 pueda entrar en ejecución debe de entrar antes el hilo 1, si este hilo 1 por cualquier motivo no entrara a ejecución, esto llevaría a la **espera infinita**. Otro de los problema encontrado es que no se satisface la condición de entrada inmediata, ya que si turno tiene en valor 1, pero el hilo 2 se encuentra en el resto del código el hilo 1 no podrá entrar a ejecución.

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int I = 0;
int turno = 1;//variable que controla la alternancia
void *codigo_del_hilo(void *id)
{
    int i = *(int *)id;
    int j = (i == 1) ? 2 : 1;
    int k;
    for (k = 0; k < 100; k++)
    {
        // protocolo de entrada
        while (turno == j)
            ;
        // Secci n cr tica
        I = (I + 1) % 10;
        printf("En hilo %d, I=%d\n", i, I);
        // protocolo salida
        turno = j;
        // Resto
    }
    pthread_exit(id);
}

int main()
{
    int h;
    pthread_t hilos[2];
    int id[2] = {1, 2};
    int error;
    int *salida;
    for (h = 0; h < 2; h++)
    {
        error = pthread_create(&hilos[h], NULL, codigo_del_hilo, &id[h]);
        if (error)
        {
            fprintf(stderr, "Error: %d: %s\n", error, strerror(error));
            exit(-1);
        }
    }
    for (h = 0; h < 2; h++)
    {
        error = pthread_join(hilos[h], (void **)&salida);
        if (error)
            fprintf(stderr, "Error: %d: %s\n", error, strerror(error));
        else
            printf("Hilo %d terminado\n", *salida);
    }
}
```

1.2. Intento 2: Exclusión mutua

El problema de la alternancia mencionada anteriormente se corrige utilizando lo conocido como **exclusión mutua**, para ello usaremos un array de booleanos para comprobar si un hilo se encuentra *true* o no *false* en la **sección crítica**. A pesar de esto existe un problema fundamental en el algoritmo, este **no asegura la exclusión mutua** ya que es posible que ambos valores del array sean verdaderos, debido a que un hilo verifica el estado del otro antes de cambiar su propio estado.

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

int I = 0;
//Arrays de hilos para comprobar si esta en secci n cr tica
bool esta_dentro[] = {false,false};

void *codigo_del_hilo(void *id)
{
    int i = *(int *)id;
    int id1 = (i == 1) ? 0 : 1; // id del hilo
    int id2 = (i == 1) ? 1 : 0; // id del otro hilo
    int k;
    for (k = 0; k < 100; k++)
    {
        // protocolo de entrada
        while (esta_dentro[id2]);
        esta_dentro[id1] = true;
        // Secci n cr tica
        I = (I + 1) % 10;
        printf("En hilo %d, I=%d\n", i, I);
        // protocolo salida
        esta_dentro[id1] = false;
        // Resto
    }
    pthread_exit(id);
}

int main()
{
    int h;
    //Array de hilos
    pthread_t hilos[2];
    int id[2] = {1, 2};
    int error;
    int *salida;
    for (h = 0; h < 2; h++)
    {
        error = pthread_create(&hilos[h], NULL, codigo_del_hilo, &id[h]);
        if (error)
        {
            fprintf(stderr, "Error: %d: %s\n", error, strerror(error));
            exit(-1);
        }
    }
    for (h = 0; h < 2; h++)
    {
        error = pthread_join(hilos[h], (void **)&salida);
        if (error)
            fprintf(stderr, "Error: %d: %s\n", error, strerror(error));
        else
            printf("Hilo %d terminado\n", *salida);
    }
}
```

2. Apartado 2: Java

2.1. Intento 1: Exclusión Mutua

En el segundo intento, al igual que en el fExluMut.c en POSIX existe la problemática fundamental de no asegurar la exclusión mutua.

```
import java.lang.Math; // para random

public class Intento2 extends Thread {
  static int n = 1;
  static int C[] = { 0, 0 };
  int id1; // identificador del hilo
  int id2; // identificador del otro hilo

  public void run() {
    try {
      for (;;) {
        while (C[id2] == 1);
        C[id1] = 1;
        sleep((long) (100 * Math.random()));
        n = n + 1;
        System.out.println("En hilo " + id1 + ", n = " + n);
        C[id1] = 0;;
      }
    } catch (InterruptedException e) {
      return;
    }
  }

  Intento2(int id) {
    this.id1 = id;
    this.id2 = (id == 1) ? 0 : 1;
  }

  public static void main(String args[]) {
    Thread thr1 = new Intento2(0);
    Thread thr2 = new Intento2(1);

    thr1.start();
    thr2.start();
  }
}
```

2.2. Intento 2: Posible interbloqueo

En este se soluciona la problemática anterior realizando la comprobación del estado propio antes de verificar el del otro hilo, **impidiendo así que entren simultáneamente a la sección crítica**, pero por otro lado existe la posibilidad de que se genere un **interbloqueo** si ambos hilos se encuentran esperando que se cambie de estado.

```
import java.lang.Math; // para random

public class Intento3 extends Thread {
  static int n = 1;
  static volatile int C[] = { 0, 0 };
  int id1; // identificador del hilo
  int id2; // identificador del otro hilo

  public void run() {
    try {
      for (;;) {
        C[id1] = 1;
        while (C[id2] == 1);
        sleep((long) (100 * Math.random()));
        n = n + 1;
```

```
        System.out.println("En hilo " + id1 + ", n = " + n);
        C[id1] = 0;;
      }
    } catch (InterruptedException e) {
      return;
    }
  }

  Intento3(int id) {
    this.id1 = id;
    this.id2 = (id == 1) ? 0 : 1;
  }

  public static void main(String args[]) {
    Thread thr1 = new Intento3(0);
    Thread thr2 = new Intento3(1);

    thr1.start();
    thr2.start();
  }
}
```

3. Apartado 3: Python

3.1. Intento 3:Posible interbloqueo

En el tercer intento interBloqueo.py, al igual que en Java se genera un interbloqueo.

```
import java.lang.Math; // para random

public class Intento3 extends Thread {
  static int n = 1;
  static volatile int C[] = { 0, 0 };
  int id1; // identificador del hilo
  int id2; // identificador del otro hilo

  public void run() {
    try {
      for (;;) {
        C[id1] = 1;
        while (C[id2] == 1);
        sleep((long) (100 * Math.random()));
        n = n + 1;
        System.out.println("En hilo " + id1 + ", n = " + n);
        C[id1] = 0;;
      }
    } catch (InterruptedException e) {
      return;
    }
  }

  Intento3(int id) {
    this.id1 = id;
    this.id2 = (id == 1) ? 0 : 1;
  }

  public static void main(String args[]) {
    Thread thr1 = new Intento3(0);
    Thread thr2 = new Intento3(1);

    thr1.start();
    thr2.start();
  }
}
```

3.2. Intento 4: Espera infinita

En el cuarto intento, se puede romper el interbloqueo generado por la condición de carrera del código anterior cambiando temporalmente el estado de states[i] a falso, e inmediatamente volver a ponerlo en verdadero. De esta forma, **se abre una ventana temporal para que uno de los procesos pueda continuar**. Este algoritmo garantiza la exclusión mutua y no se produce una espera ilimitada, pero no podemos asegurar que la espera estará limitada a un número de pasos finito fenómeno conocido como **bloqueo activo (livelock)**.

```
import threading

states = [False, False]

THREADS = 2
MAX_COUNT = 100000

counter = 0

def entry_critical_section(i):
    global states
    states[i] = True
    while states[(i+1)%2]:
      states[i] = False
      states[i] = True

def critical_section(i):
    global counter
    counter += 1

def exit_critical_section(i):
    global states
    states[i] = False

def thread(i):
    for j in range(MAX_COUNT//THREADS):
        entry_critical_section(i)
        critical_section(i)
        exit_critical_section(i)

def main():
    threads = []
    for i in range(THREADS):
        # Create new threads
        t = threading.Thread(target=thread, args=(i,))
        threads.append(t)
        t.start() # start the thread

    # Wait for all threads to complete
    for t in threads:
        t.join()

    print("Counter value: {} Expected: {}\n".format(counter, MAX_COUNT))


if __name__ == "__main__":
    main()
```