UA

ProgramaciónConcurrente

DanielAsensiRochDNI : **48776120C**

22 de noviembre de 2022

Índice

1. Algoritmo Dekker POSIX

1.1. Explicación de resultados

Tras una gran cantidad de ejecuciones probadas, el algoritmo de de Dekker con hilos Posix funciona correctamente y devuelve los resultados esperados ya que asegura la exclusión mutua, cumpliendo siempre con la condición de progreso en la ejecución y satisfaciendo siempre la limitación de espera, el *problema* de la implementación es que esta es difícil de seguir si aumenta la cantidad de procesos y la implementación se vuelve más compleja.

Cabe recalcar que esto es solo aplicable a la implementación del algoritmo de Dekker que elijamos ya que existen5 posibles implementaciones del mismo, variando esta en la satisfacción de o **la exclusión mutua** o **la alternancia explicita**, también dependiendo del algoritmo será más eficiente o menos.

Adjunto dos implementaciones del algoritmo en las cuales podemos ver dependiendo de las misma si hay exclusión mutua o no.

1.2. Dekker con exclusión mutua y valor correcto

```c
/**
 * gcc -o dekker Dekker.c -lpthread
 * taskset -c 0 dekker
 */

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

int I = 0;
int turno = 0;
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++)
    {
        esta_dentro[id1] = true;
        // protocolo de entrada
        while (esta_dentro[id2]) {
            if (turno == id2) {
                esta_dentro[id1] = false;
                while (turno != id1);
                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;
        turno = id2;
        // 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);
    }
}
```

1.3. Dekker sin exclusión muta y valores incorrectos

```
// Dekker algorithm with POSIX threads
// Compile with: gcc -o Dekker Dekker.c -lpthread
// Run with: ./Dekker

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define N 1000000

int turn;
int flag[2];
int count = 0;

void *thread(void *arg)
{
    int i, j, k;
    int me = (int)arg;
    int other = 1 - me;

    for (i = 0; i < N; i++)
    {
        flag[me] = 1;
        turn = other;
        while (flag[other] == 1 && turn == other)
            ;
        // critical section
        count++;
        // end of critical section
        flag[me] = 0;
    }
    return NULL;
}

int main()
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread, (void *)0);
    pthread_create(&t2, NULL, thread, (void *)1);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("count = %d (should be %d) \n", count, 2 * N);
    return 0;
```

```
}

//Output

//count = 2000000 (should be 2000000)
```

2. Algoritmo de Peterson Python

2.1. Explicación de resultados

Al igual que el algoritmo anterior, se ajusta a las propiedades anteriores y devuelve el resultado esperado. La única diferencia con el algoritmo anterior es que Peterson obtuvo un algoritmo más simple que es más fácil de entender y ahorra algunos ciclos de procesador. Además, las variables y la idea básica son las mismas, solo ha cambiado el orden de las declaraciones.

2.2. Codigo implementado en python

```
#Petersonm algorithm

import threading
import time

def thread1():
    global turn
    global flag
    global counter
    while True:
        flag[0] = True
        turn = 1
        while flag[1] and turn == 1:
            pass
        #critical section
        counter += 1
        print("Thread 1: ", counter)
        flag[0] = False
        time.sleep(1)

def thread2():
    global turn
    global flag
    global counter
    while True:
        flag[1] = True
        turn = 0
        while flag[0] and turn == 0:
            pass
        #critical section
        counter += 1
        print("Thread 2: ", counter)
        flag[1] = False
        time.sleep(1)

if __name__ == "__main__":
    counter = 0
    flag = [False, False]
    turn = 0
    t1 = threading.Thread(target=thread1)
    t2 = threading.Thread(target=thread2)
    t1.start()
    t2.start()

##Output
##
##Thread 1:  1
##Thread 2:  2
```

```
##Thread 1:  3
##Thread 2:  4
##Thread 1:  5
##Thread 2:  6
##Thread 1:  7
##Thread 2:  8
```

3. Algoritmo de Hyman

3.1. Explicación de resultados

Tras varias ejecuciones el programa devuelve el resultado esperado, pero el incremento de la variable "n"solo se produce en un hilo debido al lenguage usado, Java. Sin embargo, si no ejecutamos el código con el comando taskset si que se ejecutan ambos hilos. Con takset también es posible que se ejecuten ambos hilos: taskset -c 0-1 java Hyman.java, en dicho comando se contemplan ambos núcleos.

Sin el tasket puede darse la siguiente situación:

```
      A primera vista este algoritmo parece correcto, sin
embargo, se puede dar la siguiente situación:

1.- turno = 0, P1 hace C1 = (quiereentrar) y
encuentra C0 = (restoproceso) superando la
sentencia 1.3 y se para.

2.- A continuación, P0 hace C0= (quiereentrar),
encuentra turno = 0 y entra en la sección crítica.

3.- P1 hace turno = 1 y entra también en la sección
crítica.
```

3.2. Código implementado en JAVA

```
/**
 * java Hyman.java
 * taskset -c 0 java Hyman.java
*/

import java.lang.Math; // para random

public class Hyman extends Thread {
  static int n = 1;
  int turno = 0;
  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 (turno != id1) {
            while (C[id2] == 1);
            turno = id1;
        }
        sleep((long) (100 * Math.random()));
        n = n + 1;
        System.out.println("En hilo " + id1 + ", n = " + n);
        C[id1] = 0;;
      }
    } catch (InterruptedException e) {
```

```
29        return;
30      }
31    }
32
33    Hyman(int id) {
34      this.id1 = id;
35      this.id2 = (id == 1) ? 0 : 1;
36    }
37
38    public static void main(String args[]) {
39      Thread thr1 = new Hyman(0);
40      Thread thr2 = new Hyman(1);
41
42      thr1.start();
43      thr2.start();
44    }
45  }
```

4. Algoritmo de Lamport

4.1. Explicación de resultados

A diferencia de los algoritmos anteriores, este algoritmo da solución para N procesos siendo de 4 hilos el codigo implementado. Tras varias ejecuciones el programa devuelve el resultado esperado cumpliendo los tres criterios fundamentales de requisitos para exclusión mutua: Asegura la exclusión mutua, el progreso de ejecución y no hay espera ilimitada.

4.2. Código implementado en Python

```
1  #Lamport algorithm 4 threads
2
3  import threading
4  import time
5
6  def thread1():
7      global turn
8      global flag
9      global counter
10     while True:
11         flag[0] = True
12         turn = 1
13         while flag[1] and turn == 1:
14             pass
15         #critical section
16         counter += 1
17         print("Thread 1: ", counter)
18         flag[0] = False
19         time.sleep(1)
20
21 def thread2():
22     global turn
23     global flag
24     global counter
25     while True:
26         flag[1] = True
27         turn = 0
28         while flag[0] and turn == 0:
29             pass
30         #critical section
31         counter += 1
32         print("Thread 2: ", counter)
33         flag[1] = False
34         time.sleep(1)
35
36 def thread3():
```

```
37      global turn
38      global flag
39      global counter
40      while True:
41          flag[2] = True
42          turn = 0
43          while flag[0] and turn == 0:
44              pass
45          #critical section
46          counter += 1
47          print("Thread 3: ", counter)
48          flag[2] = False
49          time.sleep(1)
50
51  def thread4():
52      global turn
53      global flag
54      global counter
55      while True:
56          flag[3] = True
57          turn = 0
58          while flag[0] and turn == 0:
59              pass
60          #critical section
61          counter += 1
62          print("Thread 4: ", counter)
63          flag[3] = False
64          time.sleep(1)
65
66  if __name__ == "__main__":
67      counter = 0
68      flag = [False, False, False, False]
69      turn = 0
70      t1 = threading.Thread(target=thread1)
71      t2 = threading.Thread(target=thread2)
72      t3 = threading.Thread(target=thread3)
73      t4 = threading.Thread(target=thread4)
74      t1.start()
75      t2.start()
76      t3.start()
77      t4.start()
78
79  ##Output
80  ##
81  ##Thread 1:  1
82  ##Thread 2:  2
83  ##Thread 3:  3
84  ##Thread 4:  4
85  ##Thread 1:  5
86  ##Thread 2:  6
87  ##Thread 3:  7
88  ##Thread 4:  8
89  ##Thread 1:  9
```