Programación Concurrente

Trabajo Práctico N°3

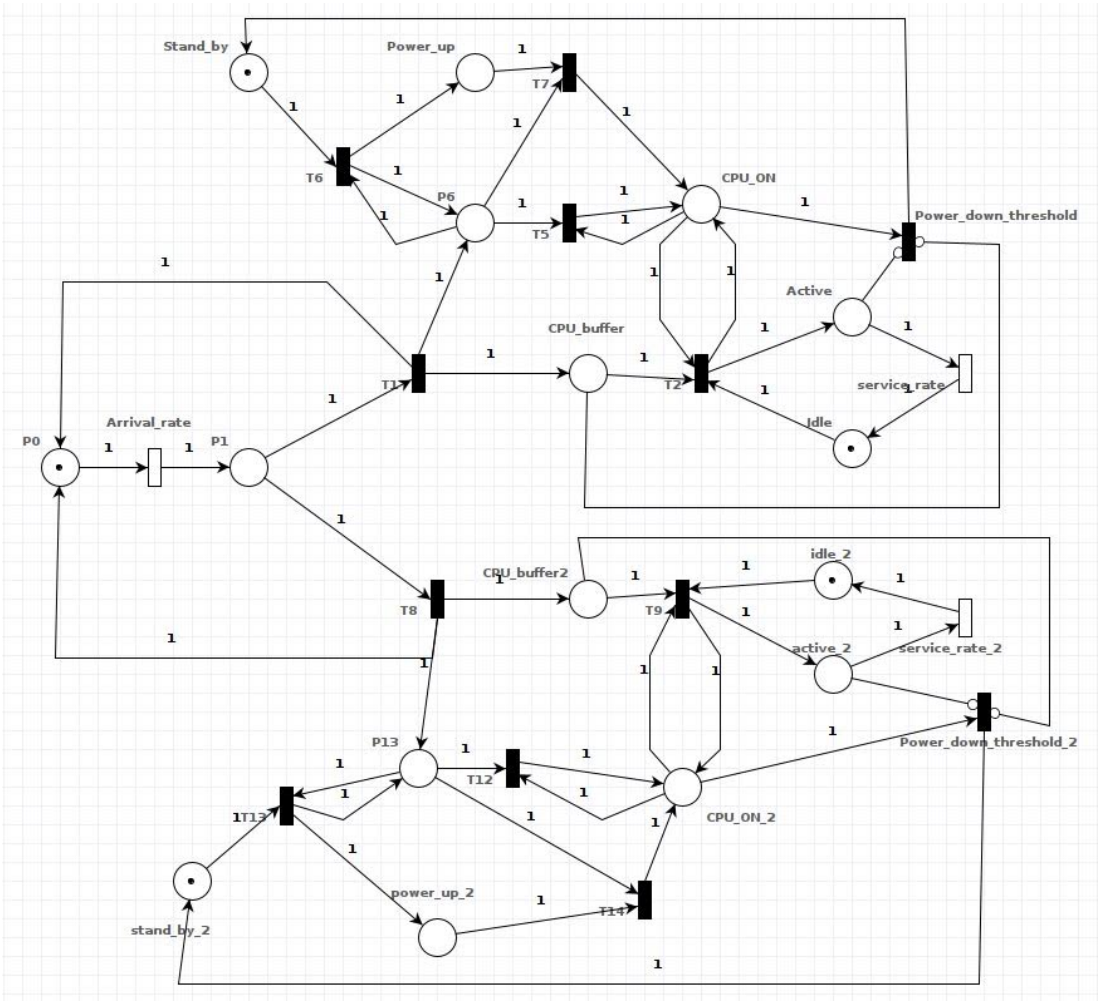
GRUPO: We are Teapots

INTEGRANTES:

Tracchia, Belén
Manterola, Lucas
Carrizo, Aixa M.
Fernández, Juan Ignacio
Ferrari, Ivo

ÍNDICE

1) La red de Petri modelada en Pipe es la siguiente:



Petri net state space analysis results

Bounded	false
Safe	false
Deadlock	false

Como se puede observar a partir del análisis de PIPE:

- La red no es acotada, puesto que los buffers pueden contener infinita cantidad de tokens.
- Dado que no es acotada, tampoco es segura.
- Tampoco hay deadlock, puesto que en ninguna marca se presenta la imposibilidad de disparar al menos una transición.

A continuación se muestran los invariantes de marcas y de transición, los cuales se analizarán más adelante.

Petri net invariant analysis results

T-Invariants

Arrival_rate	Power_down_threshold	Power_down_threshold_2	service_rate	service_rate_2	T1	T12	T13	T14	T2	T5	T6	T7	T8	T9
1	1	0	1	0	1	0	0	0	1	0	1	1	0	0
1	0	1	0	1	0	0	1	1	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0	0	0	0	0	1	1
1	0	0	1	0	1	0	0	0	1	1	0	0	0	0

The net is covered by positive T-Invariants, therefore it might be bounded and live.

P-Invariants

Active	active_2	CPU_buffer	CPU_buffer2	CPU_ON	CPU_ON_2	Idle	idle_2	P0	P1	P13	P6	Power_up	power_up_2	Stand_by	stand_by_2
0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0

The net is not covered by positive P-Invariants, therefore we do not know if it is bounded.

P-Invariant equations

M(active_2) + M(idle_2) = 1
M(CPU_ON) + M(Power_up) + M(Stand_by) = 1
M(CPU_ON_2) + M(power_up_2) + M(stand_by_2) = 1
M(Active) + M(Idle) = 1
M(P0) + M(P1) = 1

Analysis time: 0.001s

2) La política implementada está orientada a distribuir equitativamente entre ambos CPUs las tareas generadas, colocando la tarea entrante en el buffer que no recibió la tarea anterior, a menos que el tamaño del mismo sea superior al del otro buffer. De esta forma, se favorece la asignación de tareas al buffer con menor service_rate.

```
int bufferPolitic () {
    if(prevBuff == 11){
        if( buffer1.size() > buffer2.size() ){
            prevBuff = 12;
            return 12;
        }
        else
            return 11;
    }
    else if (prevBuff == 12){

        if ( buffer2.size() > buffer1.size() ){
            prevBuff = 11;
            return 11;
        }
        else
            return 12;

    }
    return -1;
}
```

3) Para la ejecución necesitaremos 5 hilos:
    - gd (GenData): Este hilo es el responsable de generar las tareas y distribuirlas entre los CPUS, utilizando la política.
    - cpu1(CPU): Actúa como “consumidor”. Quita tareas del buffer asociado a él y atiende la misma (mediante el disparo de service_rate).
    - cpu2(CPU): Ídem cpu1.
    - cpu1_poweronoff (CpuController): Se ocupa de encender y apagar el CPU asociado a él, según el estado del buffer asignado al CPU.
    - cpu2_poweronoff (CpuController): Ídem cpu1_poweronoff.

Sumados a éstos, existe un hilo encargado de, periódicamente, consultar el estado de cpu1, cpu2, cpu1_poweronoff, cpu2_poweronoff y los buffers asociados a ambos CPUs.
La cantidad de hilos a utilizar se deduce a partir de los T - invariantes. Estos últimos son 4 secuencias:

1. Generación de la tarea, asignación de la misma al CPU1, proceso de encendido, atención de la tarea y apagado del CPU1.
2. Generación de la tarea, asignación de la misma al CPU2, proceso de encendido, atención de la tarea y apagado del CPU2.
3. Generación de la tarea, asignación de la misma al CPU2, y atención de la tarea.
4. Generación de la tarea, asignación de la misma al CPU1, y atención de la tarea.

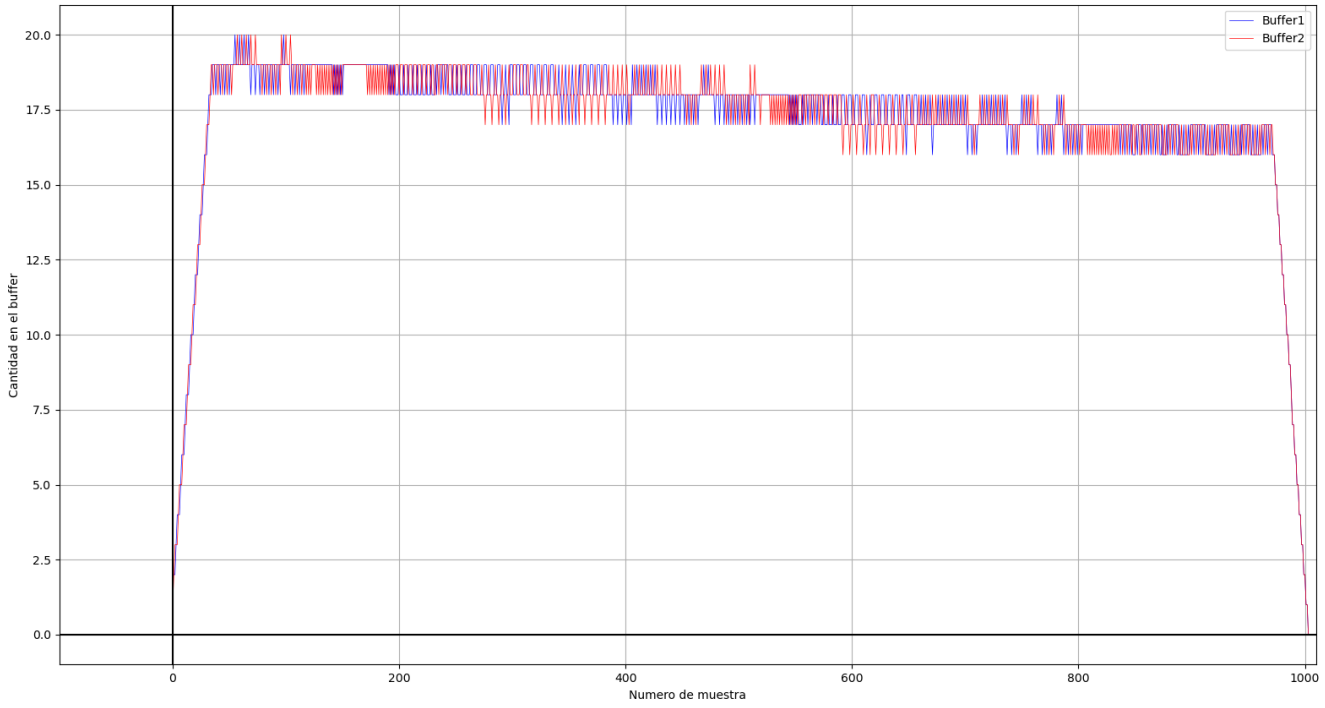
Es por esto que las operaciones de la red (las cuales son realizadas por los distintos hilos)se han dividido en los procesos que:

- Encienden, apagan y mantienen el estado de cada CPU.
- Atienden las tareas en cada CPU.
- Generan y asignan las tareas a cada buffer según lo establezca la política establecida.

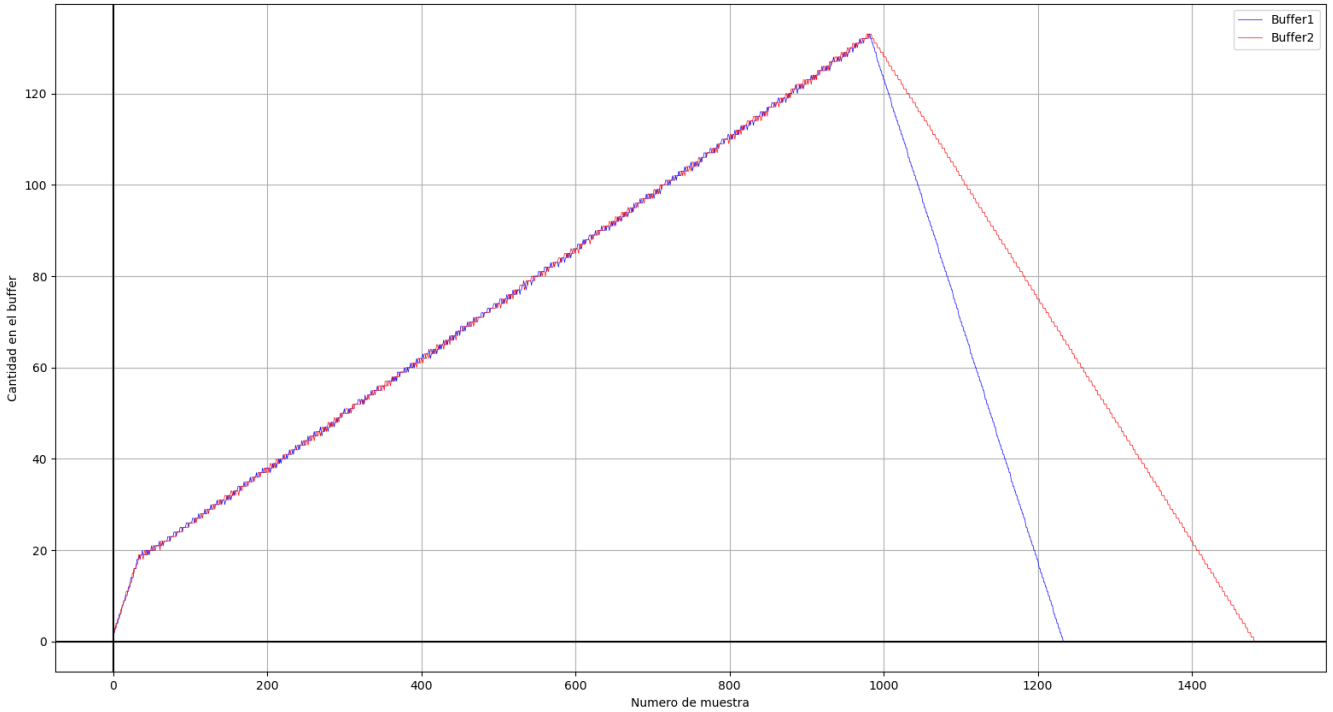
4) Se realizaron los logs con un valor de arrival_rate de 25 ms, asumiendo que el tiempo que tardan en llegar las tareas siempre será menor que el tiempo que demoran en ser atendidas (es decir, solo el retardo generado por el servicio de la tarea, sin considerar el tiempo que espera en el buffer). A partir de este supuesto, se produjeron los siguientes resultados, según los valores de service_rate de los CPUs:

	CPU1	CPU2
srv_rate (ms)	50	50
Tareas atendidas	500	500
% de tareas atendidas sobre el total	50%	50%

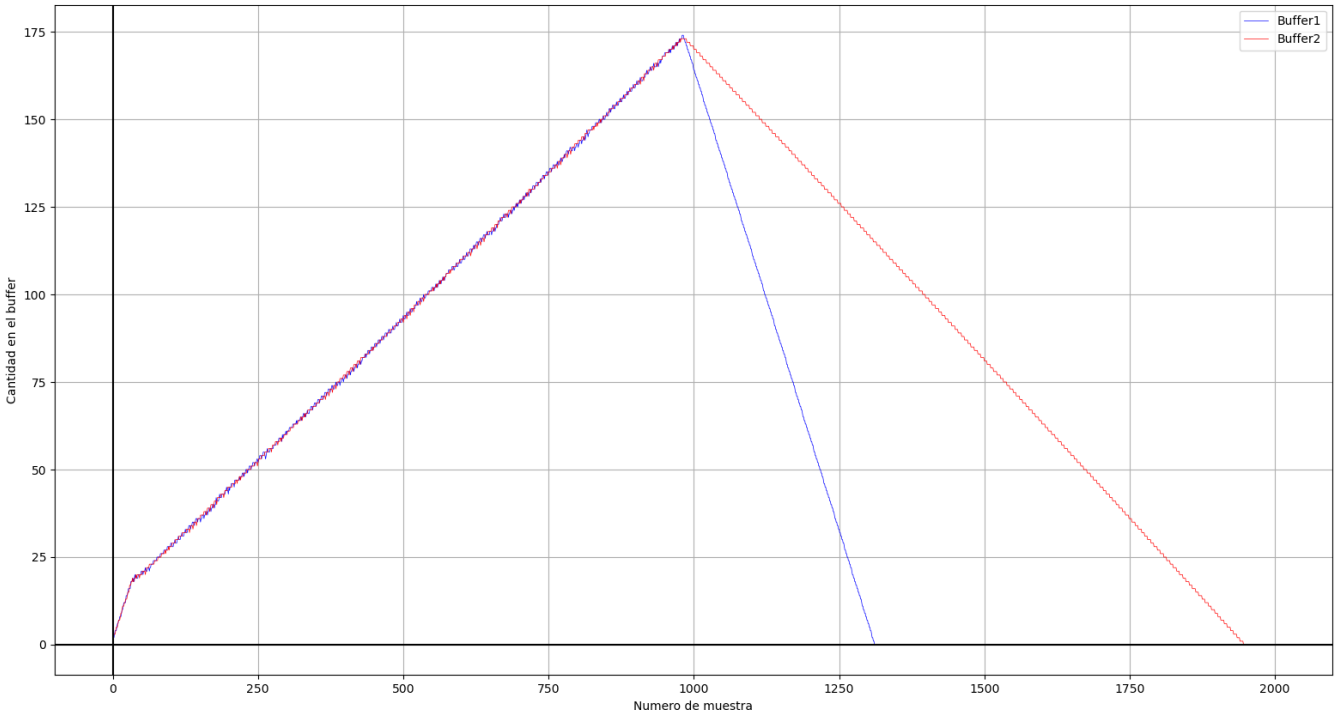
El gráfico a continuación ilustra la variación de los tamaños de las colas en los CPUs a lo largo de la ejecución:



	CPU1	CPU2
srv_rate (ms)	50	100
Tareas atendidas	621	379
% de tareas atendidas sobre el total	62,1%	37,9%



	CPU1	CPU2
srv_rate (ms)	50	150
Tareas atendidas	662	338
% de tareas atendidas sobre el total	66,2%	33,8%

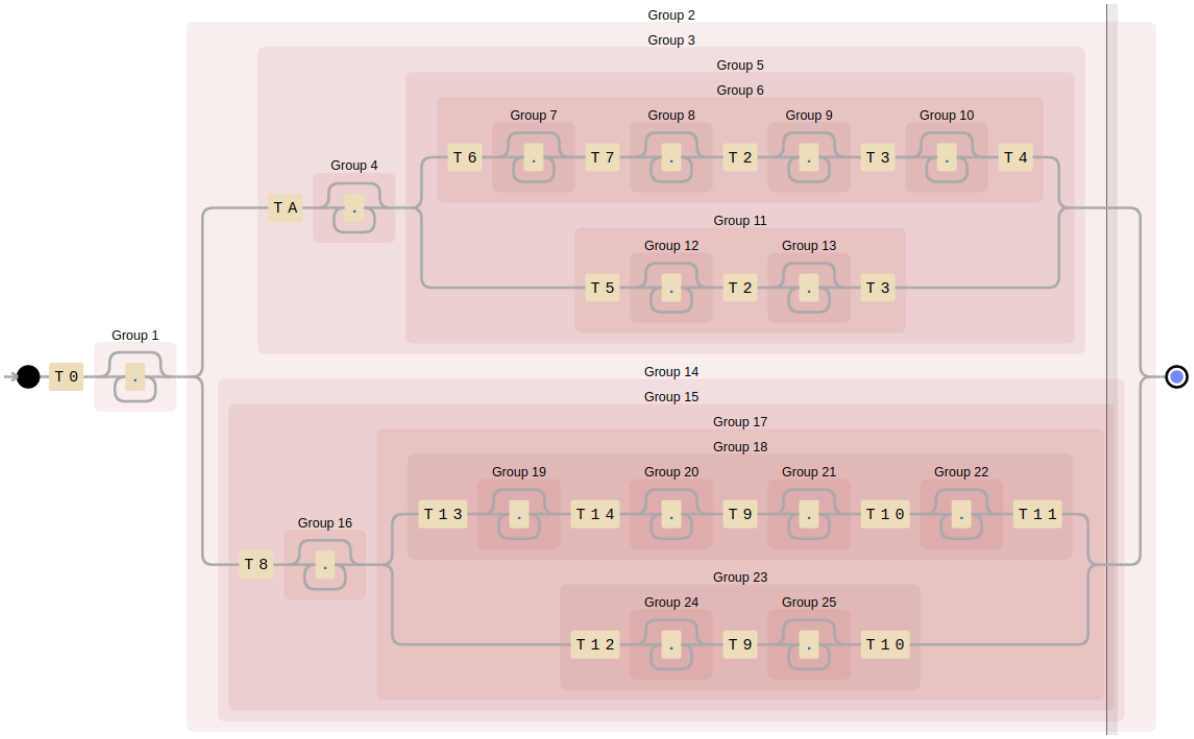


Como podemos ver, a medida que las tasas de service_rate varían una respecto de la otra, la distribución de tareas se hace desigual, favoreciendo la asignación de tareas al CPU con menor service_rate. Este comportamiento coincide con lo estipulado en la política planteada anteriormente.

5) Para verificar el cumplimiento de los invariantes de transición, se empleó un script en Python en el cual, mediante una expresión regular, se eliminan los invariantes encontrados, de forma que, si la salida al final es un string vacío, se puede afirmar el cumplimiento de los mismos durante toda la ejecución.

```
import fileinput
import re

def verifyInvariants(file ):
        i=0
        line = [file.readline() , 0]
        while(True):

                line = re.subn('T0(.*?)((TA(.*?)((T6(.*?)T7(.*?)T2(.*?)T3(.*?)T4)|(T5(.*?)T2(.*?)T3)))|((T8(.*?)((T13(.*?)T14(.*?)T9(.*?)T10(.*?)T11)|(T12(.*?)T9(.*?)T10)))))',
                '\g<1>\g<4>\g<7>\g<8>\g<9>\g<10>\g<12>\g<13>\g<16>\g<19>\g<20>\g<21>\g<22>\g<24>\g<25>', line[0].rstrip())
                #print(line[0])

                i = i+1
                if(line[1] == 0):
                        break

        line = re.subn('-' , '' , line[0].rstrip())
        if(line[0] == ""):
                print("El test ha finalizado exitosamente. Enhorabuena!")
        else:
                print("Error de T-Invariantes")
                print(line[0])

        #print(i)


file = open("prueba.txt" , "r")

print("Test de invariantes de transicion:")
verifyInvariants(file)
```

La expresión regular utilizada es la siguiente:
T0(.*?)((TA(.*?)((T6(.*?)T7(.*?)T2(.*?)T3(.*?)T4)|(T5(.*?)T2(.*?)T3)))|((T8(.*?)((T13(.*?)T14(.*?)T9(.*?)T10(.*?)T11)|(T12(.*?)T9(.*?)T10)))))



En la figura anterior se muestra una representación gráfica de la expresión regular y cómo se buscan las combinaciones asociadas a ésta (y que representan a los 4 invariantes de transición). Dichas secuencias se encuentran en el orden en el que deberían ocurrir, y son las siguientes:

- T0TAT6T7T2T3T4
- T0T8T13T14T9T10T11
- T0T8T12T9T10
- T0TAT5T2T3

Cabe aclarar que:

1. La expresión “(.*?)” hace referencia a cualquier conjunto de transiciones entre la transición anterior y posterior, permitiendo encontrar invariantes entremezclados.
2. Se utiliza TA en vez de T1. Esto se debe a que la expresión regular podría confundir la transición 11 (T11) con la transición 1 más un 1 (T1 + 1), dando un resultado erróneo.

La imagen a continuación muestra la aplicación del script a la secuencia de transiciones obtenida de una de las pruebas realizadas anteriormente.

```
D:\Aixa\Trabajos de la facu\Concurrente\tp3Returns\TP3Concurrente2021>python test.py
Test de invariantes de transicion:
El test ha finalizado exitosamente. Enhorabuena!
```

Si la secuencia de transiciones no cumple con los T-Invariantes establecidos, veremos una salida similar a la siguiente:

```
D:\Aixa\Trabajos de la facu\Concurrente\tp3Returns\TP3Concurrente2021>python test.py
Test de invariantes de transicion:
Error de T-Invariantes
T13T14T0T8T9T10
```

En la misma se muestra el mensaje que indica el fallo, junto a la secuencia de transiciones que no corresponde a ningún invariante. En este caso, la misma se obtuvo borrando T11 del final del archivo obtenido originalmente.
Cabe aclarar que el error detectado por el script no siempre es a causa de un T-invariante incorrecto, sino que puede darse el caso en el que las transiciones disparadas correspondan con uno de los T-invariantes, pero se encuentren en un orden diferente.

Por otro lado, para controlar los invariantes de plaza, luego de cada disparo que se realiza, antes de que el hilo abandone el monitor, el mismo ejecuta una rutina para verificar el cumplimiento de los mismos.

```
/*
INVARIANTES DE PLAZA
m1 + m7 = 1
m4 + m12 + m14 = 1
m5 + m13 + m15 = 1
m0 + m6 = 1
m8 + m9 = 1
*/

private boolean verifyMInvariants()throws Exception{
    int mark [] = pn.getMarkVector();

    if( ( ( mark[1] + mark[7] ) == 1) &&  ( ( mark[4] + mark[12] + mark[14] ) == 1) &&
            ( ( mark[5] + mark[13] + mark[15] ) == 1) && ( ( mark[0] + mark[6] ) == 1) &&( ( mark[8] + mark[9] ) == 1) )
        return true;
    else{
        throw new Exception ("Fallo en invariantes de plaza");
    }
}
```

En el caso de que resulte exitoso, la ejecución continuará normalmente. Si se encuentra algún error, se detendrá la ejecución y el mensaje obtenido será similar al siguiente: