Programación Concurrente

Tomas Sarquis

Julio 2020

El presente documento detalla la realización del **trabajo práctico número 3** de la asignatura **programación concurrente**, correspondiente al año **2019** por el alumno Tomas Sarquis, matrícula 39884977

1 Enunciado

Se debe implementar un simulador de un procesador con dos núcleos. A partir de la red de Petri de la *figura 1*, la cual representa a un procesador mono núcleo, se deberá extender la misma a una red que modele un procesador con dos núcleos. Además, se debe implementar una política que resuelva los conflictos que se generan con las transiciones que alimentan los buffers de los núcleos.

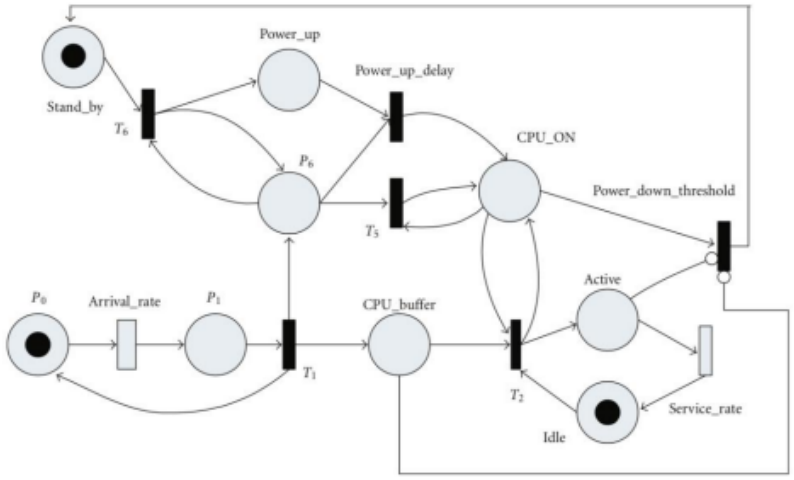

Figure 1: red de Petri mono núcleo

2 Implementación y resultados

2.1 Red de Petri

En la *figura 2* se observa cómo fue modelada la red de Petri de acuerdo al enunciado.
La red de Petri modelada cuenta con:

- 16 plazas
- 15 transiciones

2.2 Hilos

Teniendo en cuenta la red de la *figura 2*, lo siguiente es pensar cuántos hilos serán los encargados de ejecutarse el programa.

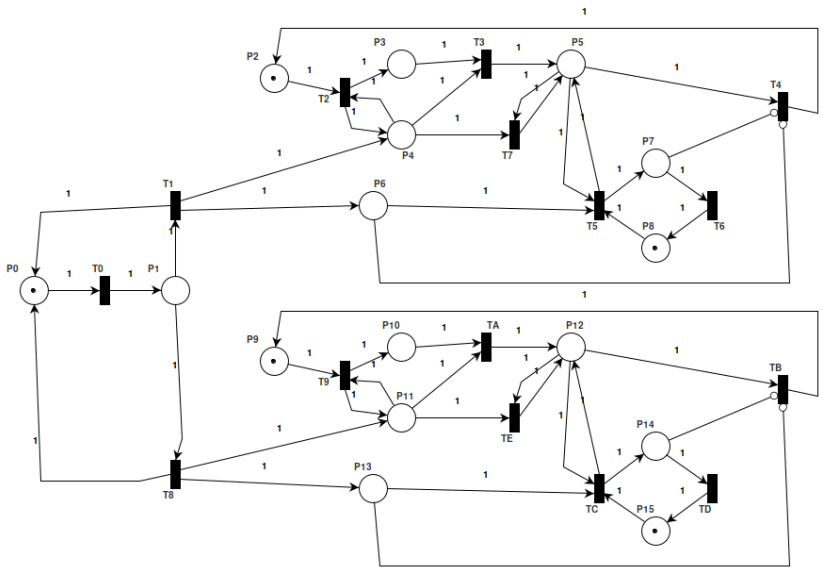

Figure 2: red de Petri doble núcleo

Luego de realizar un análisis, se llegó a la conclusión de que en la red deben trabajar **7 hilos** en forma concurrente. Estos hilos son llamados:

- ProcessGenerator
- CPUPower x 2
- CPUProcessing x 2
- CPUGarbageCollector x 2

ProcessGenerator: Encargado de generar los procesos comúnes a ambos CPUs. Dispara las transiciones 0, 1 y 8.

CPUPower: Maneja el encendido/apagado de cada CPU. Dispara las transiciones 2, 3 y 4 o 9, A y B.

CPUProcessing: Procesa las tareas de cada CPU. Dispara las transiciones 2, 3 y 4 o 9, A y B.

CPUGarbageCollector: Previene que la plaza número 4/11 no junte *tokens* "basura".

2.3 Clases en Java

El programa encargado de simular el sistema se desarrola en el lenguaje de programación *Java*. A continuación, se describen las clases, implementadas en dicho lenguaje, que modelan el programa:

- **Cola:** Colas (*queues*), las cuales son usadas para que los hilos puedan dormir en caso que lo necesiten
- **Colors:** Clase sin funciones que contiene códigos de colores (para loggear)
- **CPUGarbageCollector:** Modelo del hilo CPUGarbageCollector anteriormente descrito
- **CPU:** Modela un CPU. Clase encargada de "comunicar" los hilos Generator, Processing y GarbageCollector
- **CPUPower:** Modelo del hilo CPUPower anteriormente descrito
- **CPUProcessing:** Modelo del hilo CPUProcessing anteriormente descrito
- **InvarianteTest:** Implementación de un test de t-invariantes
- **Main:** Clase principal y ejecutable
- **Monitor:** Encargada de administrar los tiempos y el uso de la red de Petri
- **Politica:** A la hora de despertar un hilo, la politica es quien decide a cuál hacerlo
- **ProcessBuffer:** Modelo de un buffer en el cuál se depositan Process
- **ProcessGenerator:** Modelo del hilo ProcessGenerator anteriormente descrito
- **Process:** Clase que simula un proceso de CPU. Dichos Process van a parar a un ProcessBuffer
- **RedDePetri:** Implementación de la red de Petri

Para información más detallada acerca de las clases y sus funciones, se puede consultar la documentación[1] hecha por *Doxygen*[2].

[1] La documentación se encuentra en el archivo *html* "documentacion/html/index.html"
[2] www.doxygen.nl

2.4 Invariantes

Sabemos que una de las maneras de chequear que la red (y nuestro programa) funciona de forma correcta, es analizando las invariantes. Estas son, las **t-invariantes** y las **p-invariantes**.
Para el análisis, se usó la herramienta Pipe[3]. Los resultados fueron los siguientes:

P-Invariantes:

P0	P1	P10	P11	P12	P13	P14	P15	P2	P3	P4	P5	P6	P7	P8	P9
1	1	0	0	0	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	0	0
0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1

Figure 3: análisis de P-Invariantes en Pipe

La *figura 3* nos enseña qué plazas están relacionadas mediante una invariante de plazas. Esta relación indica que la suma de los *tokens* en dichas plazas, en cada relación, sea siempre unitaria. Estos es:

$M(P0) + M(P1) = 1$

$M(P14) + M(P15) = 1$

$M(P2) + M(P3) + M(P5) = 1$

$M(P10) + M(P12) + M(P9) = 1$

Para comprobar que lo anterior se cumpla siempre, la clase *RedDePetri* hace un chequeo cada vez que se dispara una transición. El método de la clase que hace dicho chequeo se llama *isPInvariantesCorrecto*.

T-Invariantes:

De la misma forma, podemos observar en la *figura 4* las invariantes de transición que existen en nuestro sistema.
Anteriormente se mencionó la clase *InvarianteTest*. Esta clase es la encargada de chequear, una vez finalizada la ejecución, que el orden de los disparos (de las transiciones) sea el correcto.
Para esto, la clase *Main* se encarga de grabar todas las transiciones (en orden) que han sido disparadas en un archivo[4] de texto. Luego, la clase

[3] http://pipe2.sourceforge.net/
[4] src/T-Invariantes.txt

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	TA	TB	TC	TD	TE
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	0	0	0	1	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	0	1	0	0	0	1	1	1

Figure 4: análisis de T-Invariantes en Pipe

InvarianteTest usa esa información para, mediante un algoritmo de remplazo, ir eliminando las coincidencias (o *matches*) entre el *log* de transiciones y las invariantes. Si el sistema funciona correctamente, deben quedar eliminadas todas las transiciones. De lo contrario, existen errores, que son escritos en otro archivo[5] de texto.

Resultados:

Al finalizar una ejecución, se loggea el estado de los análisis de invariantes. Un ejemplo se puede ver en la *figura 5*.

```
--> ANALISIS DE P-INVARIANTES: CORRECTO
--> ANALISIS DE T-INVARIANTES: INCORRECTO
```

Figure 5: ejemplo del análisis de invariantes de una ejecución

[5] src/T-InvariantesErr.txt