Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Nturales

Tomas Sarquis

Julio 2020

Programación Concurrente

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

Índice

1. Introduccion

1.1. Objetivo

El objetivo del siguiente trabajo práctico es que el estudiante sea capaz de diseñar, implementar y analizar un programa que realice una simulación mediante redes de Petri, conociendo sus propiedades y monitoreando que éstas se cumplan.

Algunos de los conocimientos aplicados en este trabajo son:

- Programación en *Java*
- Concurrencia y manejo de hilos en *Java*
- Redes de Petri: propiedades y ventajas
- *UML*[1]: diagramas para modelar

1.2. 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.

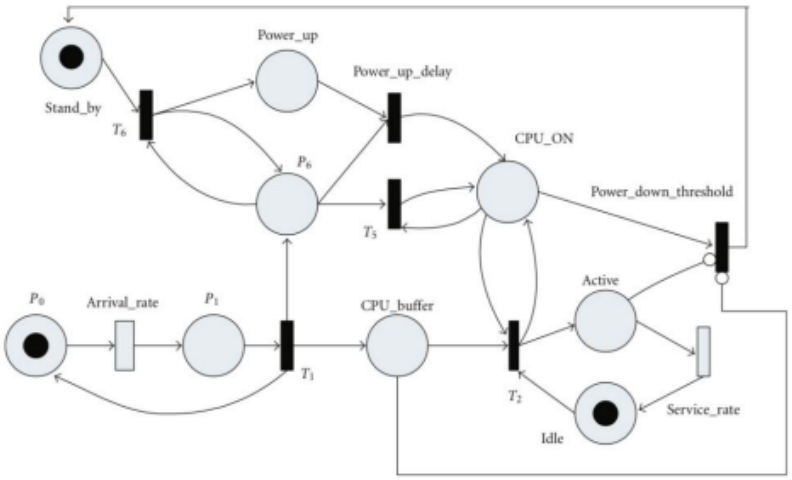

Figura 1: Red de Petri mono-núcleo

[1] *Unified Modeling Language*, `en.wikipedia.org/wiki/Unified_Modeling_Language`

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

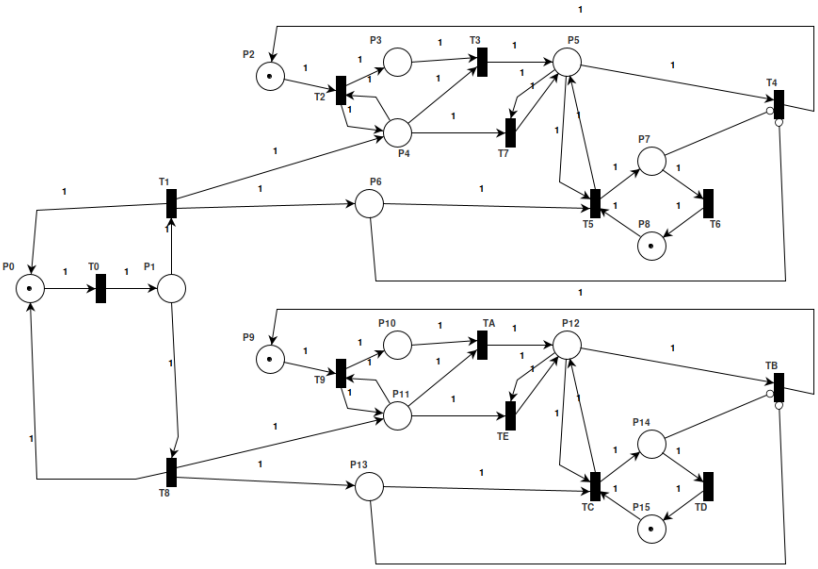

Figura 2: Red de Petri doble núcleo

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.

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 çomunicar"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
- **XMLParser:** Parsea las características de la red de Petri desde un archivo *xml*

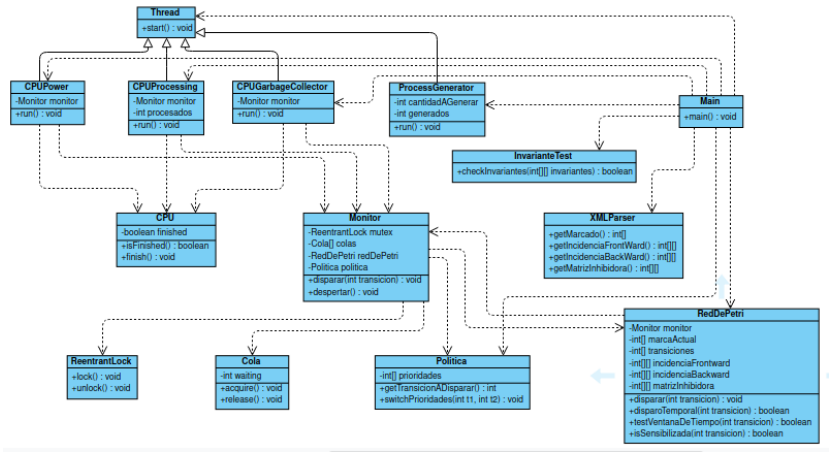

Figura 3: Diagrama de clases simplificado

Varios métodos y/o atributos fueron omitidos para mayor claridad en el diagrama de clases de la *figura 3*.

Para información más detallada acerca de las clases y sus funciones, se puede consultar la documentación[2] hecha por *Doxygen*[3].

2.4. Politica

La clase *politica* se encarga de facilitar la elección de a qué hilo despertar en el caso en que haya más de uno durmiendo.

La forma de hacer esto es mediante prioridades: se le asigna una prioridad **distinta** a cada una de las transiciones de la red.

Cuando querramos hacer uso de la politica, simplemente le indicamos cuáles son las transiciones listas para dispararse y ella nos indicará cuál de esas transiciones es la que tiene la mayor prioridad.

Además, la clase *politica* cuenta con un método capaz de intercambiar prioridades entre dos transiciones. Esto podría ser útil, en el caso en que se quiera depositar más tareas en un CPU que en el otro.

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

[3] `www.doxygen.nl`

2.5. 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[4]. 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

Figura 4: Análisis de P-Invariantes en Pipe

La *figura 4* 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 5* las invariantes de transición que existen en nuestro sistema.

[4] `http://pipe2.sourceforge.net`

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

Figura 5: Análisis de T-Invariantes en Pipe

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[5] de texto. Luego, la clase *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[6] de texto.

Resultados:

Al finalizar una ejecución, se loggea el estado de los análisis de invariantes. Un ejemplo se puede ver en la *figura 6*.

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

Figura 6: Ejemplo del análisis de invariantes de una ejecución

Se hicieron pruebas[7] con 100, 1000 y 2000 procesos y con ambos procesos trabajando en forma equilibrada (mismo tiempo de procesamiento). Además, se realizaron 10 muestras por cada cantidad de procesos. Los resultados, en la *tabla 1*.

[5] src/T-Invariantes.txt
[6] src/T-InvariantesErr.txt
[7] informe/pruebas.ods

Procesos	Transiciones totales (*avg*)	Transiciones restantes (*avg*)
100	552,6	11,5
1000	5538,4	13,8
2000	11061	14

Cuadro 1: Resultados de análisis. Cada fila equivale a 10 muestras

Como se dijo antes, para que los chequeos sean positivos, la cantidad de transiciones restantes debe ser nula.

Claramente, los chequeos de t-invariantes son negativos, siendo esto, obviamente, no deseado.
Sin embargo, la cantidad de transiciones restantes (que no cumplen con una invariante) es relativamente la misma para todos los casos. Esto quiere decir que la cantidad de transiciones restantes no dependen de la cantidad de transiciones totales.

2.6. Tiempos

A continuación, en la *tabla 2*, se describen los tiempos que lleva el programa en ejecutarse. Los tiempos de la red son los siguientes:

- $ArrivalRate = 10$ [ms]
- $ServiceRate = 15$ [ms]
- $StandByDelay = 30$ [ms]

Procesos	Tiempo de ejecución (*avg*)
500	2,59
1000	5,16
1500	7,76
2000	10,35
2500	12,95
3000	15,53
3500	18,13
4000	20,74

Cuadro 2: Resultados de análisis. Cada fila equivale a 5 muestras

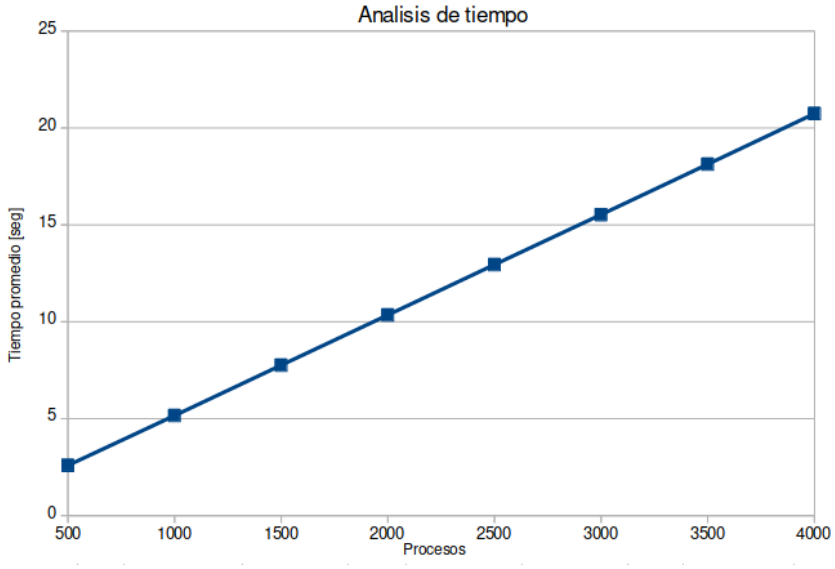

Figura 7: Gráfica de cantidad de procesos vs tiempos

Se observa una clara linealidad a medida que aumenta la cantidad de procesos.

3. Conclusión

En el presente trabajo práctico se llevó a cabo la implementación de un sistema que modela dos procesadores trabajando concurrentemente, orquestados por una red de Petri. Esta última nos da la ventaja de poder controlar que el sistema funcione correctamente, chequeando que las propiedades de la red sean las correctas. Además, el uso de redes de Petri nos da la posibilidad de escalar nuestro sistema sin mayores problemas.

4. Apéndice

4.1. Repositorio del proyecto

`https://github.com/sarquis88/final-concurrente`