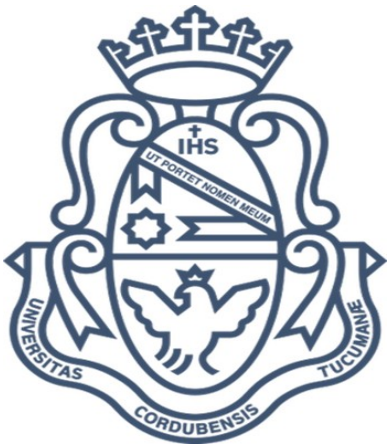
FACULTAD DE CIENCIAS EXACTAS, FÍSICA Y NATURALES
PROGRAMACIÓN CONCURRENTE
2019



INFORME TP3

Grupo: Los concurrentes

Saucedo, Juan Pablo
Martín, Tomas
Mercau, Nehemías
Mas Casariego, Sebastián Andrés

Profesores: Dr. Ing. Micolini Orlando
Ing. Ventre, Luis
Ing. Mauricio, Ludemann

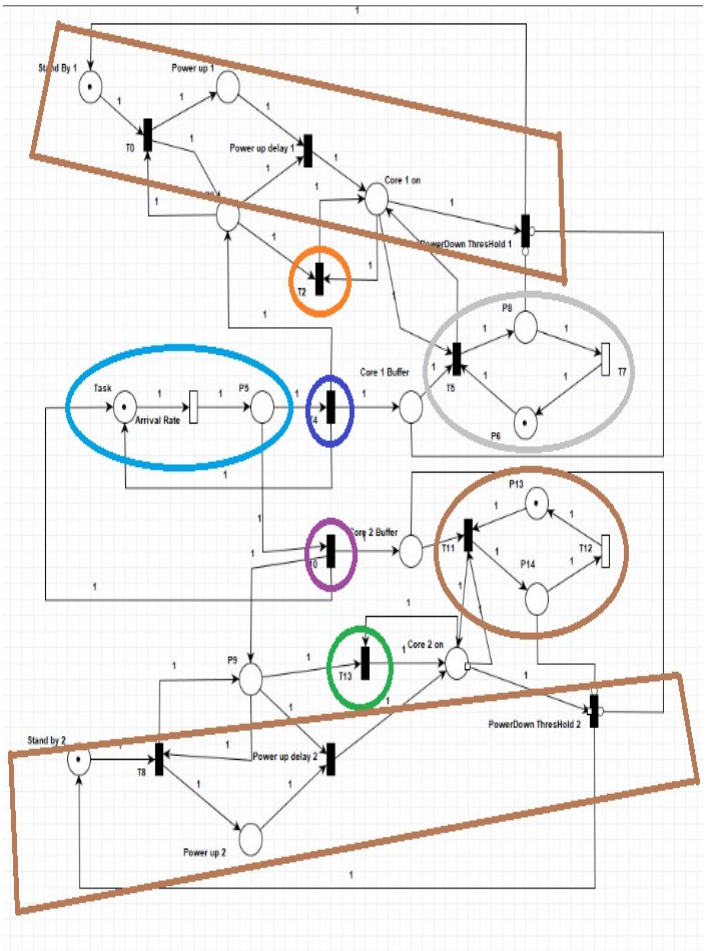
PROBLEMA: *Se debe implementar un simulador de un procesador con dos núcleos, extendiendo una red de Petri dada; se deben determinar la cantidad de threads a usar, y se deben tomar distintos tiempos para las transiciones temporizadas.*

RESOLUCIÓN Y CRITERIOS ADOPTADOS

Se construyó un monitor de concurrencia usando la Red de Petri extendida: el monitor crea una condición por cada transición que existe en la Red de Petri, y hace esperar a los threads sobre dichas condiciones en caso de que la transición que se intenta disparar, no se encuentre sensibilizada. Cuando se realiza un disparo exitoso, la RdP le notifica al monitor cuáles transiciones han sido habilitadas, y luego el monitor señaliza las condiciones asociadas a dichas transiciones.

Para determinar la cantidad de threads, se usó como base a los invariantes de marcado y de transición de la Red de Petri: los invariantes de marcado representan un recurso único, por lo que será deseable que lo maneje un único thread; además, con los invariantes, se puede identificar en la Red de Petri, cuales transiciones corresponden a un proceso secuencial. Para optimizar el tiempo de ejecución, es deseable realizar tareas de forma concurrente, por lo que cada invariante (proceso secuencial) será asignado a un thread distinto. De esta forma, se logra ejecutar el programa con un alto grado de concurrencia. Por ejemplo se puede observar el invariante de las plazas que corresponden a la llegada de tareas: es deseable que la llegada de tareas, sea independiente del resto de la PN, y se pueda hacer independientemente de si hay algún core trabajando, o de si los cores están encendidos o apagados.

Finalmente, decidimos usar un thread para cada invariante de transición de la PN, por lo que quedan 4 threads corriendo los invariantes(encendido, apagado, procesamiento del core 1 y 2), y luego otros 2 threads que corren las transiciones T2 y T13 (son los garbage collector, véase la imagen de la PN), ya que estas transiciones son totalmente independientes de las otras tareas, y de los invariantes. Además se agregan otros 2 threads, uno para cada transición que alimenta cada buffer. Por último, se agrega otro thread para el invariante de marcado de la llegada de tareas. Quedan en total, 9 threads para correr el programa.



ANÁLISIS DE LAS PROPIEDADES GENERALES DE LA RED DE PETRI

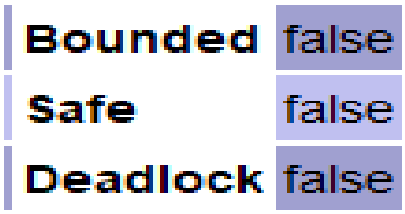
P-Invariant equations:

M(P13) + M(P14) = 1
M(P6) + M(P8) = 1
M(Core1 on) + M(Power up1) + M(Stand By1) = 1
M(Core2 on) + M(Power up2) + M(Stand By2) = 1
M(P5) + M(Task) = 1

Aquí, podemos observar los invariantes de marcado (véase, grafo de la PN), que se mencionaron en el punto anterior. Estas invariantes, ayudan a determinar, qué tareas se realizarán de forma secuencial.

La Red de Petri es limitada, no segura, y está libre de interbloqueo:

- Es limitada porque la cantidad de tokens de las plazas alcanzables no supera un número natural k.
- No es segura, ya que para cada marca alcanzable, cada lugar no contiene necesariamente cero o un token.
- Está libre de interbloqueo ya que contiene transiciones que se pueden disparar para no llegar a un punto muerto. Si está libre de interbloqueo, es una red viva.



La red de Petri que representa el problema:

- No es una Máquina de estado ya que las transiciones tienen más de un lugar de entrada y lugar de salida.
- No es un Gráfico de Marcado ya que las plazas tienen más de una transición de salida y de entrada
- No es de Libre elección ya que en un conflicto las transiciones involucradas tienen más de un una plaza de entrada. Como no se cumple que es simple tampoco será de libre elección.
- No es de Libre elección extendida ya que en un conflicto las transiciones involucradas no tienen el mismo conjunto de plazas como entrada.
- No es Simple ya que una transición involucrada se ve afectada por más de un conflicto.

- No es Simple extendida ya que una transición involucrada se ve afectada por más de un conjunto de entradas. Si no ocurre que es simple tampoco será simple extendida.

Petri net classification results

State Machine	false
Marked Graph	false
Free Choice Net	false
Extended Free Choice Net	false
Simple Net	false
Extended Simple Net	false

RESULTADOS

Se probó el sistema con los siguientes parámetros:

A) Se eligen retrasos para el arrival rate, proceso del core 1 y proceso de core 2 de 10ms, 50ms, 50ms respectivamente.

$$T_T: Tareas\ Totales = 1000\ tareas$$

$$\alpha: Arrival\ rate = 0.1\ \frac{tareas}{ms}$$

$$\beta_1: Service\ rate\ Core\ 1 = \frac{1}{50}\ \frac{tareas}{ms} = 0.02\ \frac{tareas}{ms}$$

$$\beta_2: Service\ rate\ Core\ 2 = \frac{1}{50}\ \frac{tareas}{ms} = 0.02\ \frac{tareas}{ms}$$

Y se obtuvieron los siguientes resultados:

$$t_T: Tiempo\ total\ de\ ejecución = 26240\ ms$$

$$T_1: Tareas\ procesadas\ por\ el\ core\ 1 = 500\ tareas$$

$$T_2: Tareas\ procesadas\ por\ el\ core\ 2 = 500\ tareas$$

B) Aquí se duplica el tiempo de proceso de uno de los core respecto al punto A

$$T_T: Tareas\ Totales = 1000\ tareas$$

$$\alpha: Arrival\ rate = 0.1\ \frac{tareas}{ms}$$

$$\beta_1: Service\ rate\ Core\ 1 = \frac{1}{50}\ \frac{tareas}{ms} = 0.02\ \frac{tareas}{ms}$$

$$\beta_2: Service\ rate\ Core\ 2 = \frac{1}{100}\ \frac{tareas}{ms} = 0.01\ \frac{tareas}{ms}$$

Y se obtuvieron los siguientes resultados:

$$t_T: Tiempo\ total\ de\ ejecución = 45165\ ms$$

$$T_1: Tareas\ procesadas\ por\ el\ core\ 1 = 558\ tareas$$

$$T_2: Tareas\ procesadas\ por\ el\ core\ 2 = 442\ tareas$$

C) Aquí se triplica el tiempo de proceso de uno de los core respecto al punto A

$$T_T: Tareas\ Totales = 1000\ tareas$$

$$\alpha: Arrival\ rate = 0.1\ \frac{tareas}{ms}$$

$$\beta_1: Service\ rate\ Core\ 1 = \frac{1}{50}\ \frac{tareas}{ms} = 0.02\ \frac{tareas}{ms}$$

$$\beta_2: Service\ rate\ Core\ 2 = \frac{1}{150}\ \frac{tareas}{ms} = 6.66 * 10^{-3}\ \frac{tareas}{ms}$$

Y se obtuvieron los siguientes resultados:

$$t_T: Tiempo\ total\ de\ ejecución = 64450\ ms$$

$$T_1: Tareas\ procesadas\ por\ el\ core\ 1 = 576\ tareas$$

$$T_2: Tareas\ procesadas\ por\ el\ core\ 2 = 424\ tareas$$

Analizando los resultados y el sistema en sí, podemos concluir primero, que obviamente el procesamiento de las tareas no puede terminar antes de que se produzcan, y a partir de ese tiempo de arribo se va a tardar lo que tarde el core más lento en procesar las tareas que le quedan en el buffer.

Si el retraso en la llegada de tareas es mayor que el tiempo máximo que se tarda en procesarlas, entonces el tiempo total usado por el sistema va a ser simplemente el producto entre el tiempo de llegada de cada tarea por la cantidad de tareas totales.

En caso de que las tareas llegan más rápido de lo que se las procesa:

$$t_{arribo} = \frac{T_T}{\alpha} + T_T * 2.5 \qquad (1)$$

En nuestras computadoras, se tarda 2,5 ms extra en producir cada tarea.

Las tareas restantes por procesar al momento en el que termina la producción de tareas viene dada por la siguiente fórmula:

$$T_{restantes} = T_T - t_{arribo} * (\beta_1 + \beta_2) \qquad (2)$$

Estas se dividen equitativamente en cada buffer, por lo que en el momento en que termina la producción de tareas, cada core tiene asignadas

$$\frac{T_{restantes}}{2}$$

tareas en su buffer respectivo.

Por lo que podemos escribir que:

$$t_{TOTAL} = t_{arribo} + \frac{T_{restantes}}{2 * min(\beta_1; \beta_2)} \qquad (3)$$

Aplicando a los 3 casos anteriores (3):

$$t_{arribo} = \frac{1000}{0.1} + 1000 * 2.5 = 12500 ms$$

Para A)

$$T_{restantes} = 1000 - 12500 * \left(\frac{1}{50} + \frac{1}{50}\right) = 500$$

Para B)

$$T_{restantes} = 1000 - 12500 * \left(\frac{1}{50} + \frac{1}{100}\right) = 625$$

Para C)

$$T_{restantes} = 1000 - 12500 * \left(\frac{1}{50} + \frac{1}{150}\right) = 667$$

Lo que implica que el tiempo total en cada caso es:

Para A) $$t_{TOTAL} = 12500 + \frac{500}{\frac{2*1}{50}} = 25000 ms$$

Para B) $$t_{TOTAL} = 12500 + \frac{625}{\frac{2*1}{100}} = 43750 ms$$

Para C) $$t_{TOTAL} = 12500 + \frac{667}{\frac{2*1}{150}} = 62525 ms$$

Se observan dos cosas: la primera es que las tareas tienden a agruparse mayormente en el core mas rápido, lo cual es lógico ya que la decisión de a que buffer alimentar, se toma en base al marcado de las plazas de los buffers. Si un core es mas lento, es lógico que acumule mas marcas en su plaza, y por tanto la política elija mas seguido al buffer del core mas rápido.

Lo segundo que se puede observar es que mientras mas lento sea el core mas lento de los dos, mas tiempo de ejecución va a necesitar el sistema. A pesar de esto, el core mas rápido compensa en cierta medida la lentitud del otro core gracias a la distribución equitativa de las tareas.

Estos resultados son estimativos, y pueden variar ligeramente con respecto al tiempo real de ejecución.

T-Invariants y P-Invariants

Los invariantes de transición son 4: dos para los encendidos de los núcleos, y dos para el proceso de cada tarea.

Para comprobar los invariantes T, y verificar que se respetan las propiedades estructurales de la PN, se guardan las transiciones de interés en una cadena, y luego con una expresión regular, se verifica cada invariante por separado. Cuando se cumple un invariante, este es eliminado de la cadena. Si al final de la ejecución, la cadena esta vacía, es porque se cumplen los invariantes de transición.

Los invariantes de plaza se verifican cada vez que se dispara una transición. Esto se hace verificando que la suma de las marcas de los invariantes de plaza, sea correcto.En cualquier caso(éxito o fracaso) y tanto para los P-Invariants como para los T-Invariants, se escribe un mensaje de información al final de los registros de ejecución de programa.

Conclusiones:

Inicialmente se contempló la posibilidad de asignar un mismo thread al encendido, apagado del core, y del proceso de las tareas en el núcleo. Esto se debe a que el encendido del core, y el proceso de las tasks, son ambas tareas excluyentes, y por tanto podrían ser ejecutadas secuencialmente por el mismo thread. Si se implementara el programa como se mencionó anteriormente, no se ganaría nada en tiempo de ejecución: la decisión final de separar las tareas de encendido y proceso en distintos threads se basa en el hecho, de que si le mismo thread tuviera que encender y procesar, se debería mantener en el thread información sobre la Red de Petri, que ayude a determinar cuál invariante debe ejecutarse a continuación; esto no es deseable, lo mejor es que al thread se le asigne una única tarea, sin darle información sobre el estado del recurso y que el monitor se encargue de demorar al thread hasta que se pueda realizar su tarea.

Adicionalmente, en una de las primeras versiones del programa se tenía un mismo thread asignado a las transiciones de llegada de tasks y a la alimentación de los buffers. De la misma forma que en el caso anterior, para implementarlo de esta forma es necesario que el thread tenga información sobre qué buffer debería alimentar lo cual no es deseable.

El uso de 9 threads minimiza el tiempo de ejecución de este simulador, al mismo tiempo que minimiza el uso de recursos del procesador. Sin embargo, siempre que uno de los núcleos sea mas lento que el otro, va a generar un cuello de botella que va a ralentizar todo el sistema.