Universidad Nacional Autónoma de México

Facultad de Ciencias

Ciencias de la Computación
Semestre 2026-1

Computación Concurrente

Práctica 1

Alejandro Jacome Delgado	320011704
Ana Cristina Cuevas García	318269263
Isaac Robledo Ramírez	320140655

Gilde Valeria Rodríguez Jiménez	Profesor
Isabel Espino Gutiérrez	Ayudante
Dulce Julieta Mora Hernández	Ayudante
Christian Alfredo Solís Calderón	Ayud. Lab.

28 de agosto de 2025

a. Lee lo siguiente `https://www.evanjones.ca/software/threading-linus-msg.html` y comparte en máximo 4 líneas de computadora a que se refiere Linus Torvalds con un contexto de ejecución y cómo se relaciona con la definición en la sección 1 de esta práctica.

L. Torvalds se refiere al contexto de ejecución (COE) como los distintos estados ejemplificando al los del CPU, memoria, permisos, o de comunicación. Así, con una forma tradicional de ver, se tiene un proceso con casi todo lo que conocemos como contexto, y el thread que tiene el proceso y estados del CPU, pero en Linux esto lo unifica en un COE.

b. ¿Cuántos hilos tiene disponibles tu computadora?
Ejecuta `Runtime.getRuntime().availableProcessors()`, si son más de uno en el equipo escriban el de cada uno.

- Jacome Delgado Alejandro:
  Mi máquina tiene 12 hilos (6 núcleos físicos)
- Cuevas Garcia Ana Cristina:
  Mi máquina cuenta con 12 hilos,
- Robledo Ramírez Isaac:
  Mi máquina contiene 16 hilos.

El programa `CantidadHilos.java` guarda en una variable los procesadores disponibles en tiempo de ejecución e imprime junto con un mensaje, dicha variable.

c. Revisa el programa `Determinante concurrente` y responde ¿Cuánto tiempo tarda en ejecutarse?

El programa está diseñado para obtener el determinante de una matriz de 3x3 de manera concurrente.

Descripción `DeterminanteConcurrente.java`:
La clase extiende de la clase `Thread`, esta se encarga de asignar cada producto (diagonales de la matriz para calcular el determinante) a un thread diferente instancia de esta clase; cuando los threads terminen, hace las sumas y restas necesarias para obtener el determinante, imprimiendo la duración del intervalo, junto con el determinante.

Al ejecutar 20 veces el programa tomamos que el programa tarda un tiempo promedio de $965189\,ns$, lo cual se puede ver en la siguiente gráfica:

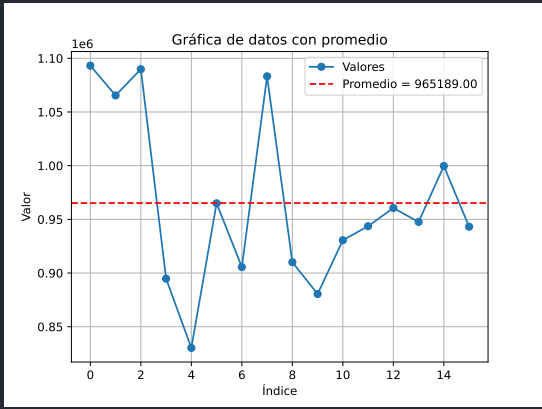

Figura 1: Gráfica de los valores con su promedio (en milisegundos)

d. El programa `Determinante concurrente` está implementado extendiendo la clase `Thread`. Implementa el programa utilizando la interfaz `Runnable`.

   Descripción `DeterminanteConcurrenteRunnable.java`:
   La clase implementa la clase `Runnable`, esta se encarga de asignar cada producto (diagonales de la matriz para calcular el determinante) a un thread diferente, los cuales fueron creados a través de un Runnable específico instancia de la clase; cuando los threads terminen, el resultado de cada thread, hace las sumas y restas necesarias para obtener el determinante, imprimiendo la duración del intervalo, junto con el determinante.

e. Implementa el programa `Determinante concurrente` de forma secuencial.

   Descripción `DeterminanteSecuencial.java`:
   La clase se encarga de resolver los productos dados por diagonales de la matriz una a una; cuando el cálculo termine, el resultado será la resta de las suma de los productos de las diagonales necesarias para obtener el determinante, finalmente imprime la duración del intervalo, junto con el determinante.

f. Implementa del programa `Determinante concurrente` para dos hilos (en vez de seis).

   Descripción `DeterminanteSecuencial2Hilos.java`:
   La clase se encarga de resolver los productos dados por diagonales de la matriz una a una; cuando el cálculo termine, el resultado será la resta del resultado que ejecutarán dos threads, siendo las suma de los productos de las diagonales necesarias para obtener el determinante, el primer thread ejecutando la parte positiva y el otro, la parte negativa. Finalmente imprime la duración del intervalo, junto con el determinante.

g. Compara las 3 implementaciones: el programa `Determinante concurrente` para dos hilos, para seis hilos y el programa secuencial. Responde: ¿A qué se debe el orden en el que se ordenan los tiempos de ejecución de cada programa?

En orden más rápido a más lento tenemos...

a. `DeterminanteSecuencial` con $1400\,ns$

b. `DeterminanteConcurrente2Hilos` con $464900\,ns$

c. `DeterminanteConcurrente` con $712100\,ns$

La razón por la cual el programa secuencial fue mas veloz que los programas concurrentes, es por que el programa no es lo suficientemente grande/largo/pesado para que la concurrencia sea efectiva, ya que en Java, al crear las instancias que implementan Runnable y las instancias de Thread, es lo que hacen pesado el trabajo, esto lo podemos ver en la siguiente Figura pues se imprimieron las cadenas al empezar, después de crear las instancias y al finalizar el cálculo; y como podemos apreciar, el intervalo mayor es entre el inicio y al crear las instancias. Por ende, no es conveniente, tendríamos que implementarlo en un programa pesado para que valiese la pena, ya que el hecho de que los programas concurrentes deban crear, iniciar y unir hilos para poder realizar los cálculos requeridos, es contraproducente, pues en este caso al ser un programa pequeño y de sencilla resolución, conviene el programa secuencial.

```
$ java DeterminanteConcurrenteRunnable
Entrada: 2000ns
Start: 8781200ns
Finish: 9676300ns
Program took 9854500ns, result: -18
```

Figura 2: Resultado de ejecutar DeterminanteConcurrenteRunnable.java

h. Si utilizas la *Ley de Amdahl* entre el programa `Determinante concurrente` para dos hilos y el programa secuencial. ¿El resultado es mayor o menor a 1? ¿Por qué?

Aquí el resultado del secuencial es

$$\begin{aligned}\frac{1}{(1-P)+\frac{P}{n}} &= \frac{1}{(1-0)+\frac{0}{1}} \\ &= \frac{1}{1+0} \\ &= \frac{1}{1} \\ &= 1\end{aligned}$$

Al no tener trabajo que pudieramos acelerar, la cantidad sigue siendo 1.

Aquí el resultado del programa con 2 hilos, suponiendo el 50 % del trabajo es paralelizable.

$$\begin{aligned}\frac{1}{(1-P)+\frac{P}{n}} &= \frac{1}{(1-0.5)+\frac{0.5}{2}} \\ &= \frac{1}{0.5+0.25} \\ &= \frac{1}{0.75} \\ &= 1.333^{-}\end{aligned}$$

Aunque el porcentaje de trabajo paralelizable aumente o descienda (por obvias razones, no descienda a 0), al tener trabajo paralelizable, esto va a aumentar el tiempo en el que se realiza la tarea, acelerando el proceso, por lo tanto será mayor a 1.

i. Describe con tus propias palabras en máximo dos líneas para qué sirve el método `join()`. Si no utilizas el método `join()` en `Determinante Concurrente`, ¿sigue funcionando?

El método `join()`, donde un `ThreadB` ejecuta `ThreadA.join()`, entonces `ThreadB` continuará ejecutándose cuando `ThreadA`, el que se unió, acabe su ejecución.

Si se elimina la estructura *try-catch* completo, aunque el programa termina, el funcionamiento no es el esperado, pues el contador siempre varía en el resultado, casi siempre terminando en 0, o variando con -2 y -14, muy raramente en -12 o -16, etc.

Referencias

[1] Torvalds Linus. *Re: proc fs and shared pids.* Linux Kernel Archive. URL: https://www.evanjones.ca/software/threading-linus-msg.html.

[2] Nathan. *java - Runtime.getRuntime().availableProcessors().* StackOverflow. URL: https://stackoverflow.com/questions/11877947/runtime-getruntime-availableprocessors.