Escuela Colombiana de Ingeniería

Arquitecturas de Software

Introducción al paralelismo - hilos

Trabajo individual o en parejas

Entrega: Martes en el transcurso del día. Entregar: Fuentes y documento PDF con las respuestas.

Parte I Hilos Java

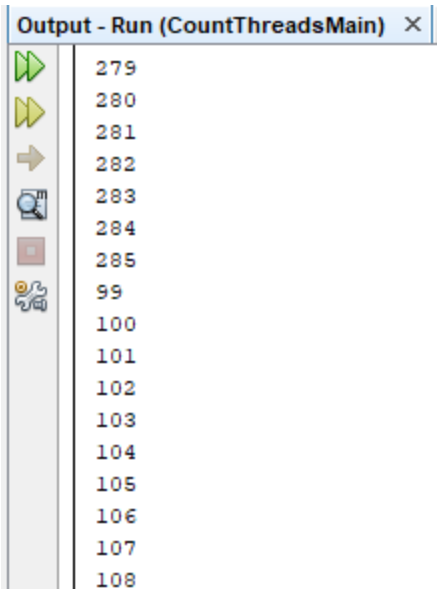
1. De acuerdo con lo revisado en las lecturas, complete las clases CountThread, para que las mismas definan el ciclo de vida de un hilo que imprima por pantalla los números entre A y B.
2. Complete el método **main** de la clase CountMainThreads para que:
   i. Cree 3 hilos de tipo CountThread, asignándole al primero el intervalo [0..99], al segundo [99..199], y al tercero [200..299].

```
Thread t1 = new Thread (new CountThread(0, 99));
Thread t2 = new Thread (new CountThread(99, 199));
Thread t3 = new Thread (new CountThread(200, 299));
```

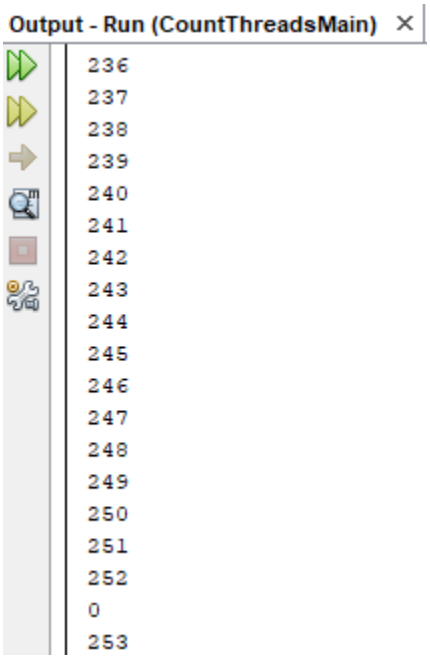
   ii. Inicie los tres hilos con 'start()'.

```
t1.start();
t2.start();
t3.start();
```

   iii. Ejecute y revise la salida por pantalla.



iv. Cambie el incio con 'start()' por 'run()'. Cómo cambia la salida?, por qué?.



- Cuando se usa `start()` el hilo se crea y el código que está en `run()` es ejecutado en dicho nuevo hilo haciendo que se ejecute en paralelo. Cuando se usa `run()` directamente, ninguna nueva instancia se va a crear haciendo que el código que está en `run()` se ejecute en un hilo presente y en secuencia.

Parte II Hilos Java

Para un software de vigilancia automática de seguridad informática se está desarrollando un componente encargado de validar las direcciones IP en varios miles de listas negras (de host maliciosos) conocidas, y reportar aquellas que existan en al menos cinco de dichas listas.

Para 'refactorizar' este código, y hacer que explote la capacidad multi-núcleo de la CPU del equipo, realice lo siguiente:

1. Cree una clase de tipo Thread que represente el ciclo de vida de un hilo que haga la búsqueda de un segmento del conjunto de servidores disponibles. Agregue a dicha clase un método que permita 'preguntarle' a las instancias del mismo (los hilos) cuantas ocurrencias de servidores maliciosos ha encontrado o encontró.

```
public class SearchSegmentThread extends Thread{


    @Override
    public void run(){

    }

    public int instances(){

        return 0;
    }
}
```

2. Agregue al método 'checkHost' un parámetro entero N, correspondiente al número de hilos entre los que se va a realizar la búsqueda (recuerde tener en cuenta si N es par o impar!).

```
 * @param ipaddress suspicious host's IP address.
 * @param n quantity of threads
 * @return  Blacklists numbers where the given host's IP address was found.
 */
public List<Integer> checkHost(String ipaddress, int n){
```

   Modifique el código de este método para que divida el espacio de búsqueda entre las N partes indicadas, y paralelice la búsqueda a través de N hilos. Haga que dicha función espere hasta que los N hilos terminen de resolver su respectivo sub-problema, agregue las ocurrencias encontradas por cada hilo a la lista que retorna el método, y entonces calcule (sumando el total de ocurrencias

encontradas por cada hilo) si el número de ocurrencias es mayor o igual a *BLACK_LIST_ALARM_COUNT*. Si se da este caso, al final se DEBE reportar el host como confiable o no confiable, y mostrar el listado con los números de las listas negras respectivas. Para lograr este comportamiento de 'espera' revise el método join del API de concurrencia de Java. Tenga también en cuenta:

- Dentro del método checkHost Se debe mantener el LOG que informa, antes de retornar el resultado, el número de listas negras revisadas VS. el número de listas negras total (línea 60). Se debe garantizar que dicha información sea verídica bajo el nuevo esquema de procesamiento en paralelo planteado.

- Se sabe que el HOST 202.24.34.55 está reportado en listas negras de una forma más dispersa, y que el host 212.24.24.55 NO está en ninguna lista negra.

```
INFORMACIÓN: HOST 202.24.34.55 Reported as NOT trustworthy
ago 16, 2018 12:06:16 AM edu.eci.arsw.blacklistvalidator.HostBlackListsValidator checkHost
INFORMACIÓN: Checked Black Lists:80.000 of 80.000
The host was found in the following blacklists:[29, 10034, 20200, 31000, 70500]

INFORMACIÓN: HOST 212.24.24.55 Reported as trustworthy
ago 16, 2018 12:02:10 AM edu.eci.arsw.blacklistvalidator.HostBlackListsValidator checkHost
INFORMACIÓN: Checked Black Lists:80.000 of 80.000
The host was found in the following blacklists:[]
```

Parte II.I Para discutir el Martes (NO para implementar aún)

La estrategia de paralelismo antes implementada es ineficiente en ciertos casos, pues la búsqueda se sigue realizando aún cuando los N hilos (en su conjunto) ya hayan encontrado el número mínimo de ocurrencias requeridas para reportar al servidor como malicioso. Cómo se podría modificar la implementación para minimizar el número de consultas en estos casos?, qué elemento nuevo traería esto al problema?

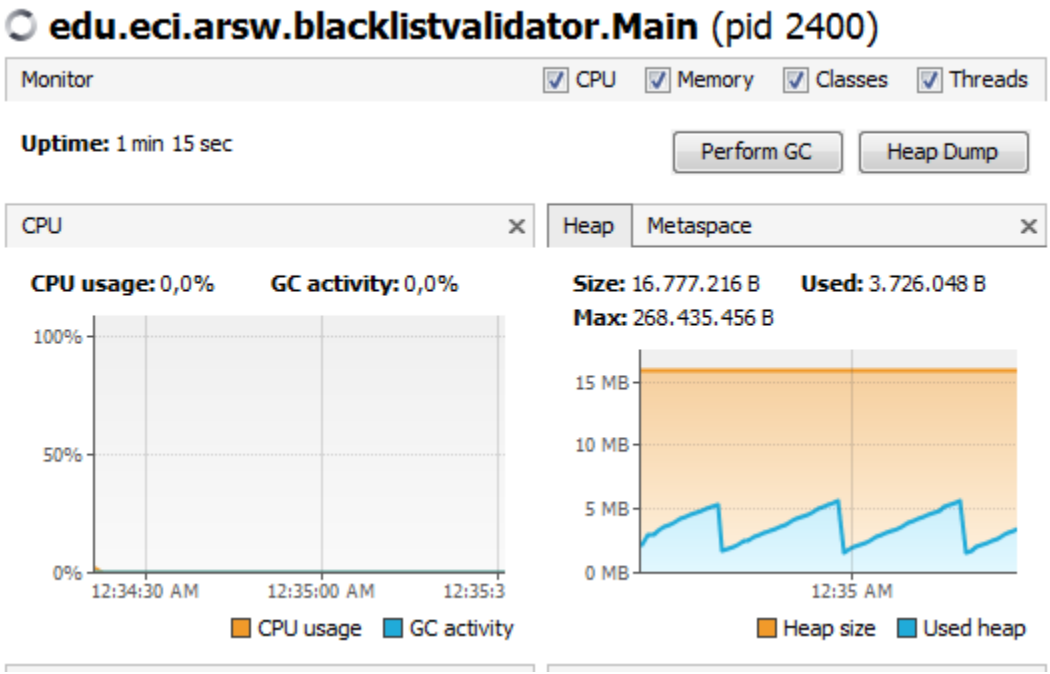
Posiblemente con la creación de un thread de chequeo sobre las ocurrencias de cada thread de segmento de búsqueda.

Parte III Evaluación de Desempeño

A partir de lo anterior, implemente la siguiente secuencia de experimentos para realizar las validaciones de direcciones IP dispersas (por ejemplo 202.24.34.55), tomando los tiempos de ejecución de los mismos (asegúrese de hacerlos en la misma máquina):

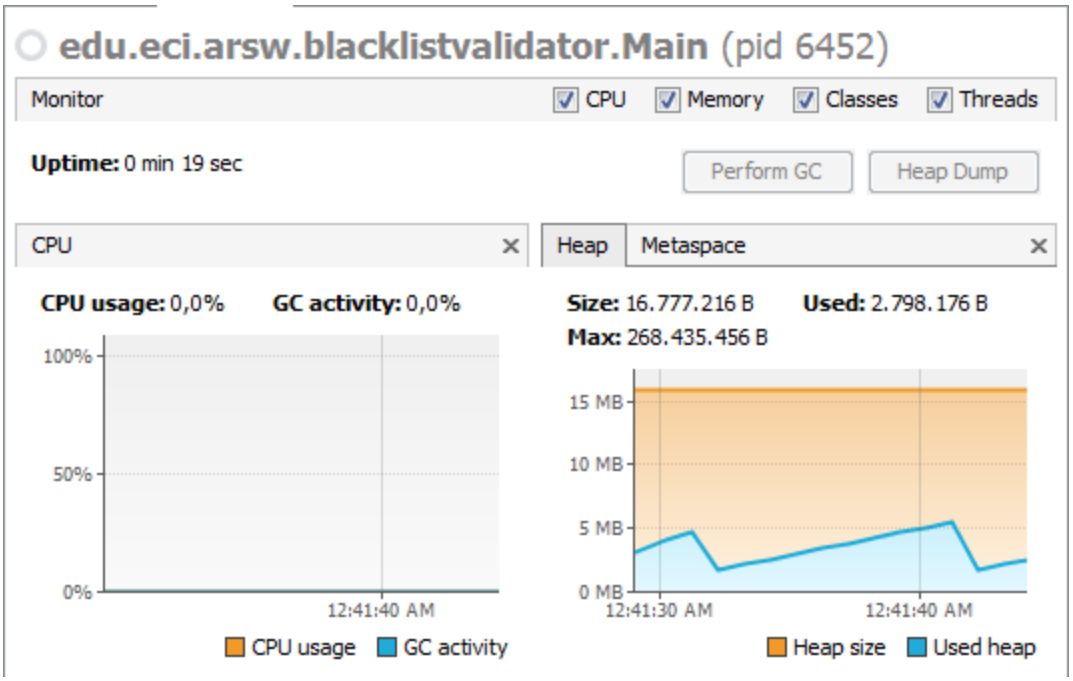
Al iniciar el programa ejecute el monitor jVisualVM, y a medida que corran las pruebas, revise y anote el consumo de CPU y de memoria en cada caso.

1. Un solo hilo. **t= 1:22.302s**

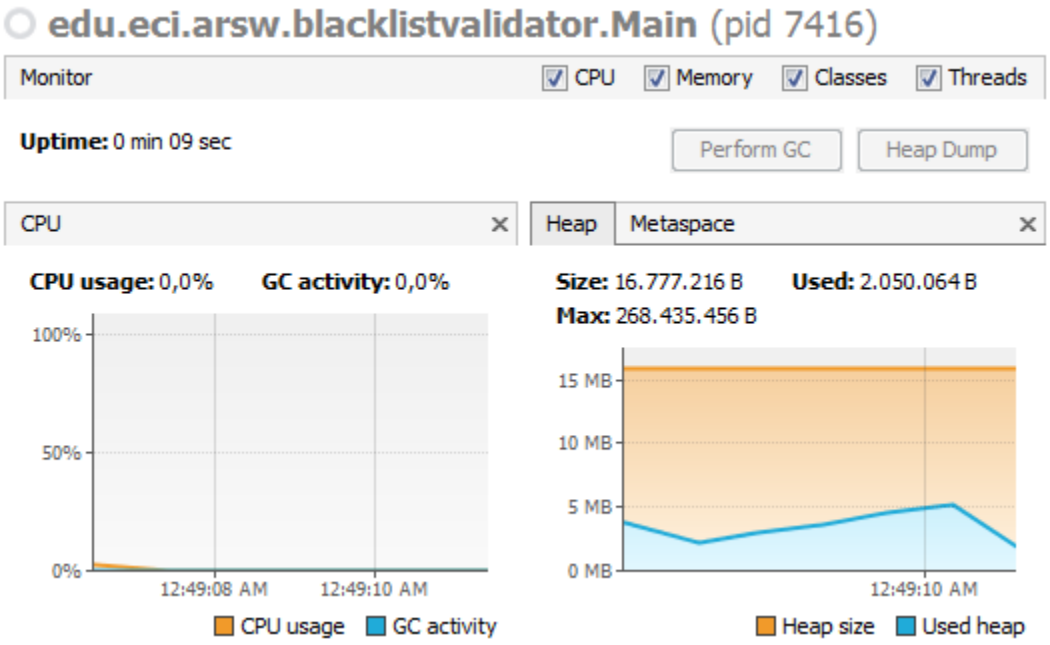


2. Tantos hilos como núcleos de procesamiento (haga que el programa determine esto haciendo uso del API Runtime). **t= 21.081s**

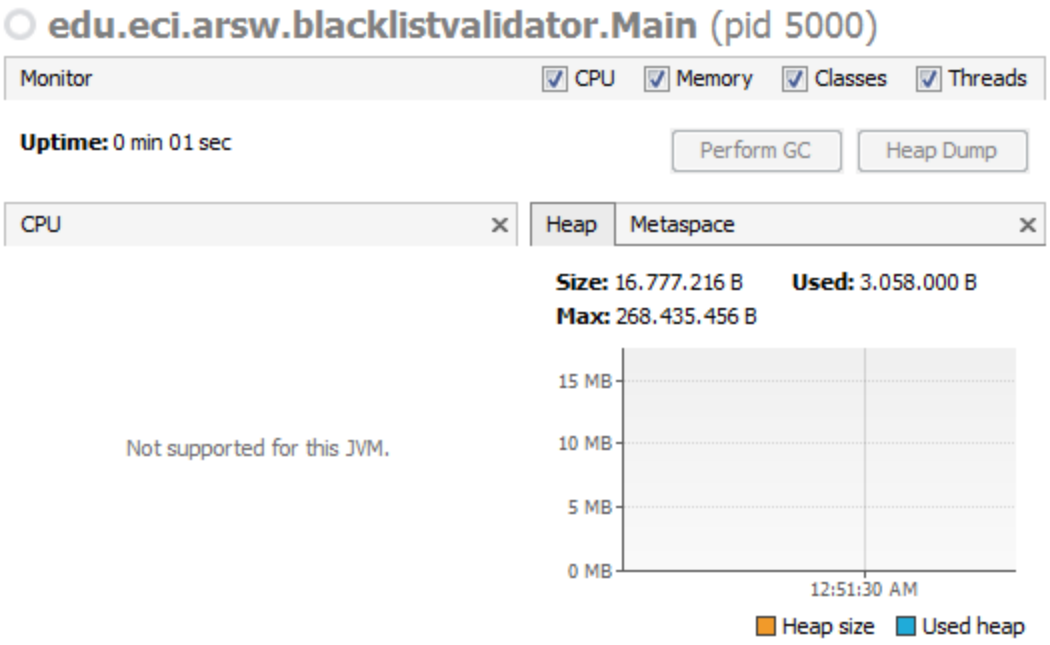
```
Runtime run = Runtime.getRuntime();
int avaProc=run.availableProcessors();
List<Integer> blackListOcurrences=hblv.checkHost("202.24.34.55",avaProc);
```



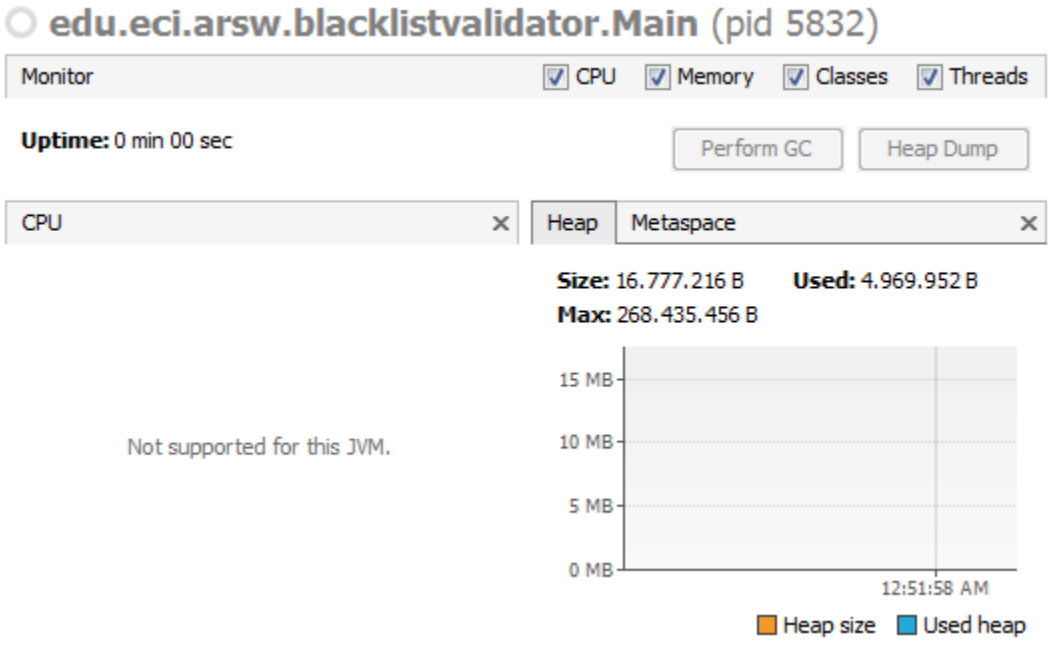
3. Tantos hilos como el doble de núcleos de procesamiento. **t= 11.104s**



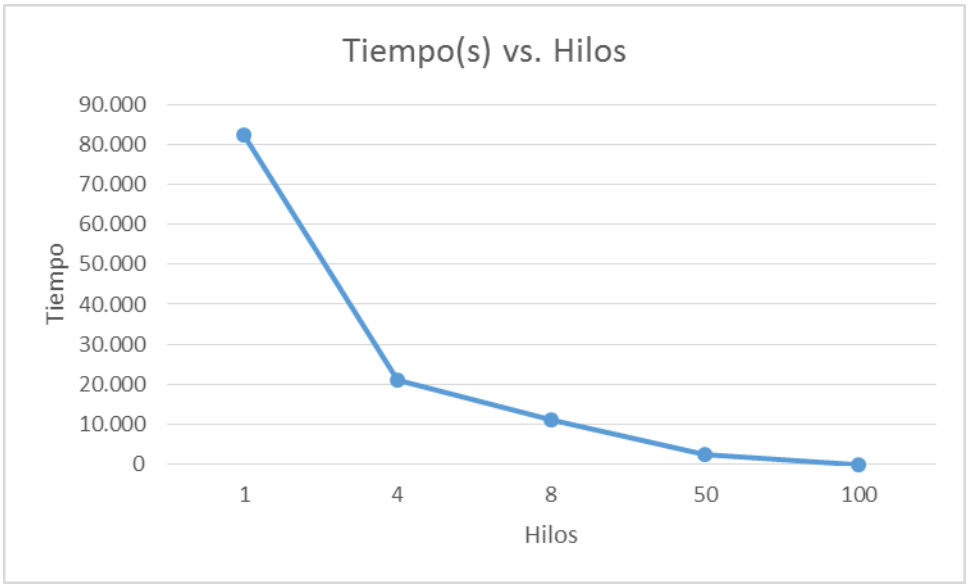
4. 50 hilos. **t= 2.289s**



5. 100 hilos. **t= 1.460s**



Con lo anterior, y con los tiempos de ejecución dados, haga una gráfica de tiempo de solución vs. número de hilos. Analice y plantee hipótesis con su compañero para las siguientes preguntas (puede tener en cuenta lo reportado por jVisualVM):



1. Según la ley de Amdahls:

   $$S(n) = \frac{1}{(1-P) + \frac{P}{n}}$$

   , donde *S(n)* es el mejoramiento teórico del desempeño, *P* la fracción paralelizable del algoritmo, y *n* el número de hilos, a mayor *n*, mayor debería ser dicha mejora. Por qué el mejor desempeño no se logra con los 500 hilos?, cómo se compara este desempeño cuando se usan 200?.

   No se logra mejor desempeño debido a que el thread scheduler selecciona los threads según su prioridad ubicandolos en el estado Running, al ser demasiados, la JVM se mantiene cambiando el estado de cada thread, adicionando así mayor tiempo a que si fueran menos threads.

2. Cómo se comporta la solución usando tantos hilos de procesamiento como núcleos comparado con el resultado de usar el doble de éste?

   $$\frac{82.302\ s}{21.081\ s} = 3.904\ (4\ hilos)$$

   $$\frac{82.302\ s}{11.104\ s} = 7.282\ (8\ hilos)$$

   Es 1.865 veces más rápido

3. De acuerdo con lo anterior, si para este problema en lugar de 100 hilos en una sola CPU se pudiera usar 1 hilo en cada una de 100 máquinas hipotéticas, la ley

de Amdahls se aplicaría mejor?. Si en lugar de esto se usaran c hilos en 100/c máquinas distribuidas (siendo c es el número de núcleos de dichas máquinas), se mejoraría?. Explique su respuesta.

-Si, el thread scheduling al cambiar de estado un solo hilo, es insignificante, entonces, la mejora sería mucho mayor.

- Suponiendo que las maquinas usan los c núcleos físicos aprovecharía al 100% la cpu de cada una, según la ley de amdahls el rendimiento mejoraría considerablemente.