Semáforos

Objetivo

El objetivo de la guía es el de conocer el uso de los semáforos para resolver problemas reales con el uso del lenguaje de programación Java. Presentará los mecanismos que ofrece Java para la creación y uso de Semáforos.

Competencias

- Conocer los mecanismos para el uso de Semáforos en Java
- Desarrollar destreza en el uso de los semáforos
- Aplicar los semáforos en el solución de un problema real

Semáforos en Java

Un semáforo controla el acceso a un recurso compartido mediante el uso de un contador. Si el contador es mayor que cero, entonces se permite el acceso. Si es cero, entonces se niega el acceso. Lo que el contador está contando son permisos (permit's) que permiten el acceso al recurso compartido. Por lo tanto, para acceder al recurso, un hilo debe tener un permiso (permit) del semáforo.

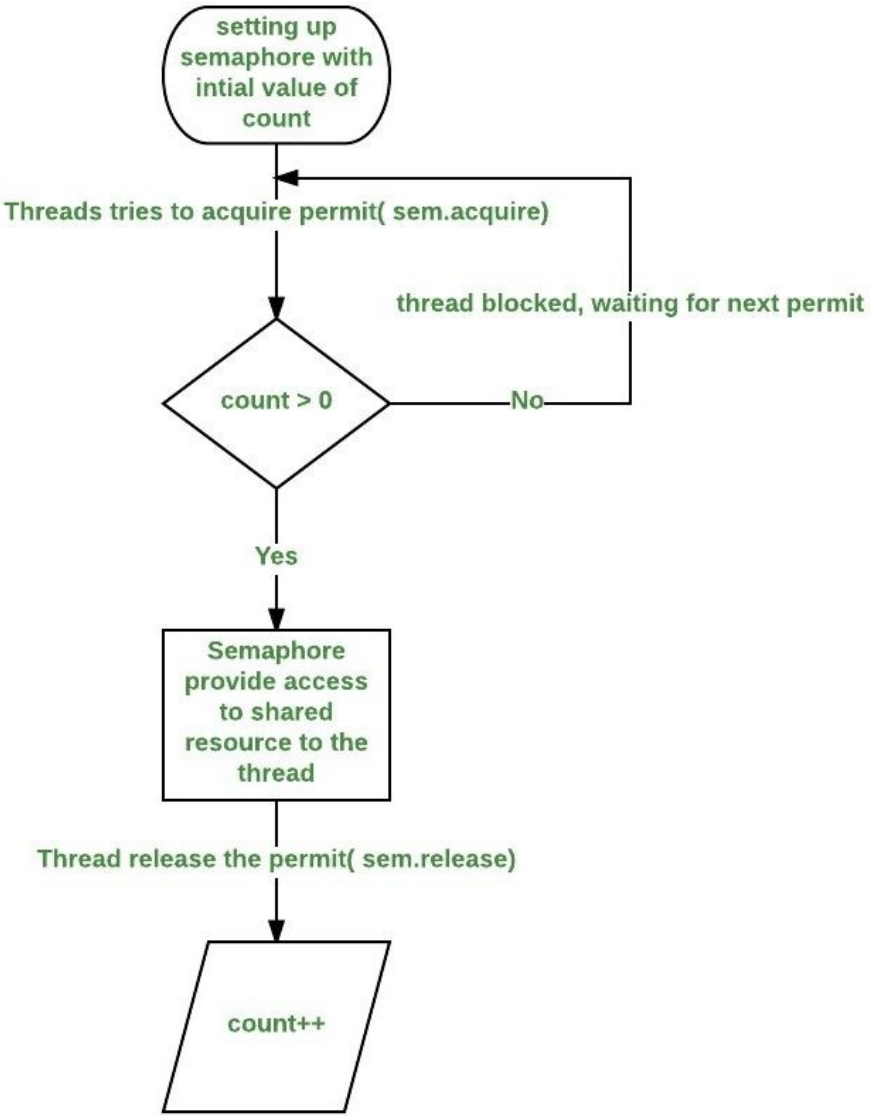
Trabajando con el semáforo

En general, para utilizar un semáforo, el hilo que quiere acceder al recurso compartido intenta adquirir un permiso (permit).

- Si el contador del semáforo es mayor que cero, entonces el hilo adquiere un permiso, lo que hace que el contador del semáforo sea decrementado.
- De lo contrario, el hilo será bloqueado hasta que se pueda obtener un permiso.
- Cuando el hilo ya no necesita acceder al recurso compartido, libera el permiso, lo que hace que el conteo del semáforo sea incrementado.
- Si hay otro hilo esperando un permiso, entonces ese hilo adquirirá un permiso en ese momento.

La clase Semaphore

Java proporciona la clase Semaphore en el paquete java.util.concurrent la cual implementa este mecanismo.



Constructores en la clase Semáforo:

Hay dos constructores en la clase Semáforo.

1. Semáforo (núm. Int)
2. Semáforo (int num, boolean how)

Aquí, num especifica el número de permisos inicial. Por lo tanto, especifica el número de subprocesos que pueden tener acceso a un recurso compartido en cualquier momento. Si es uno, entonces sólo un hilo puede acceder al recurso en cualquier momento. De forma predeterminada, todos los subprocesos en espera reciben un permiso en un orden no definido. Al establecer how a true, puede asegurarse de que a los hilos en espera se les concede un permiso en el orden en el que solicitaron acceso.

Uso de semáforos como mutex (que impiden condiciones de carrera)

Podemos usar un semáforo para bloquear el acceso a un recurso, cada hilo que quiere utilizar ese recurso debe llamar primero a *acquire*() antes de acceder al recurso para adquirir el bloqueo. Cuando el hilo se hace con el recurso, debe llamar a release() para liberar el bloqueo. He aquí un ejemplo que demuestra esto:

```
// java program to demonstrate
// use of semaphores Locks
import java.util.concurrent.*;

//A shared resource/class.
class Shared
{
        static int count = 0;
}

class MyThread extends Thread
{
        Semaphore sem;
        String threadName;
        public MyThread(Semaphore sem, String threadName)
        {
          super(threadName);
          this.sem = sem;
          this.threadName = threadName;
        }

        @Override
        public void run() {

          // run by thread A
          if(this.getName().equals("A"))
          {
                System.out.println("Starting " + threadName);
                try
                {
                  // First, get a permit.
                  System.out.println(threadName + " is waiting for a permit.");

                  // acquiring the lock
                  sem.acquire();

                  System.out.println(threadName + " gets a permit.");

                  // Now, accessing the shared resource.
                  // other waiting threads will wait, until this
                  // thread release the lock
                  for(int i=0; i < 5; i++)
                  {
```

```
                    Shared.count++;
                    System.out.println(threadName + ": " + Shared.count);

                    // Now, allowing a context switch -- if possible.
                    // for thread B to execute
                    Thread.sleep(10);
                }
                } catch (InterruptedException exc) {
                    System.out.println(exc);
                }

                // Release the permit.
                System.out.println(threadName + " releases the permit.");
                sem.release();
          }

          // run by thread B
          else
          {
                System.out.println("Starting " + threadName);
                try
                {
                  // First, get a permit.
                  System.out.println(threadName + " is waiting for a permit.");

                  // acquiring the lock
                  sem.acquire();

                  System.out.println(threadName + " gets a permit.");

                  // Now, accessing the shared resource.
                  // other waiting threads will wait, until this
                  // thread release the lock
                  for(int i=0; i < 5; i++)
                  {
                        Shared.count--;
                        System.out.println(threadName + ": " + Shared.count);

                        // Now, allowing a context switch -- if possible.
                        // for thread A to execute
                        Thread.sleep(10);
                  }
                  } catch (InterruptedException exc) {
                        System.out.println(exc);
                  }
                  // Release the permit.
                  System.out.println(threadName + " releases the permit.");
                  sem.release();
          }
        }
}

// Driver class
public class SemaphoreDemo
{
        public static void main(String args[]) throws InterruptedException
        {
          // creating a Semaphore object
          // with number of permits 1
```

```
        Semaphore sem = new Semaphore(1);

        // creating two threads with name A and B
        // Note that thread A will increment the count
        // and thread B will decrement the count
        MyThread mt1 = new MyThread(sem, "A");
        MyThread mt2 = new MyThread(sem, "B");

        // stating threads A and B
        mt1.start();
        mt2.start();

        // waiting for threads A and B
        mt1.join();
        mt2.join();

        // count will always remain 0 after
        // both threads will complete their execution
        System.out.println("count: " + Shared.count);
      }
}

Output:

Starting A
Starting B
B is waiting for a permit.
B gets a permit.
A is waiting for a permit.
B: -1
B: -2
B: -3
B: -4
B: -5
B releases the permit.
A gets a permit.
A: -4
A: -3
A: -2
A: -1
A: 0
A releases the permit.
count: 0
```

Nota: La salida puede ser diferente en diferentes ejecuciones del programa anterior, pero el valor final de la variable de recuento siempre permanecerá 0.

Explicación del programa anterior:

- El programa utiliza un semáforo para controlar el acceso a la variable count, que es una variable estática dentro de la clase Shared. Shared.count se incrementa cinco veces por el hilo A y se decrementa cinco veces por el hilo B.Para evitar que estos dos subprocesos accedan a Shared.count al mismo tiempo, se permite el acceso sólo después de que se ha obtenido un permiso del semáforo de control. Una vez

que se haya completado el acceso, se libera el permiso. De esta manera, sólo un hilo a la vez usará Shared.count, como muestra la salida.

- Observe la llamada a sleep() dentro del método run() dentro de la clase MyThread. Se utiliza para "probar" que los accesos a Shared.count son sincronizados por el semáforo. En run(), la llamada a sleep() hace que el hilo invocador haga una pausa entre cada acceso a Shared.count. Normalmente, esto permitiría que se ejecute el segundo subproceso. Sin embargo, debido al semáforo, el segundo hilo debe esperar hasta que el primero haya liberado el permiso, lo que ocurre sólo después de que todos los accesos por el primer hilo estén completos. Así, Shared.count primero se incrementa cinco veces por el hilo A y luego se decrementa cinco veces por el hilo B. Los incrementos y decrementos no se entremezclan en código de ensamblado.
- Sin el uso del semáforo, los accesos a Shared.count por ambos subprocesos se habrían producido simultáneamente, y los incrementos y decrementos estarían entremezclados. Para confirmar esto, intente comentar las llamadas a acquire () y release (). Al ejecutar el programa, verá que el acceso a Shared.count ya no está sincronizado, por lo que no siempre obtendrá el valor de cuenta 0.

Asignación

- Utilice la referencia #2 de la Guía para conocer más sobre la clase Semáforo, sobre todo sus métodos.
- Utilice la referencia #3 en la sección **Limiting connections** para conocer una técnica para limitar el número de conexiones de cualquier tipo.

Estas 2 referencias deben ser utilizadas por Usted para desarrollar la asignación de la guía 04.1

Bibliografía

1. Gaurav Miglani, GeeksforGeeks, Semáforos en Java
   http://www.geeksforgeeks.org/semaphore-in-java/
2. Gaurav Miglani, GeeksforGeeks, Java.util.concurrent.Semaphore class in Java
   http://www.geeksforgeeks.org/java-util-concurrent-semaphore-class-java/
3. Craig Fichel, Java Code Geeks, Java Concurrency Tutorial – Semaphores,
   https://www.javacodegeeks.com/2011/09/java-concurrency-tutorial-semaphores.html
4. App Shah, Crunchify,com, What is Java Semaphore and Mutex – Java Concurrency MultiThread explained with Example
   http://crunchify.com/what-is-java-semaphore-and-mutex-java-concurrency-multithread-explained-with-example/