Prácticas Concurrencia y Distribución (18/19)

Arno Formella, Anália García Lourenço, Hugo López Fernández, David Olivieri

semana 8:(29 marzo – 02 abril)

2. Semana 8 (29/03–02/04): Semaphores y Thread Executors

Objetivos: Implementar un semáforo y estudiar su comportamiento dentro del modelo de consumidor productor

1. **(P4: para entregar en grupo de práctica)**: Variables locales del hilo e Interrupts

   Un semáforo es un objeto que controla el acceso a un recurso común. Antes de acceder al recurso, un hilo debe adquirir un permiso del semáforo. Después de terminar con el recurso, el hilo debe devolver el permiso al semáforo.

   a) Dado el siguiente código, completa el código necesario para los métodos down () y up (). Usa esta implementación en el código del productor-consumidor de la semana pasada.

```
public class MySemaphore {
   public MySemaphore (int initialValue) {
      // PARA HACER: Implementar esto
   }
   // initialize a sempahore with value zero
   public MySemaphore () {
      this(0);
   }

   public void down () {
      //  PARA HACER: Implementar esto
   }
   public void up () {
      //  PARA HACER:Implementar esto
   }
}
```

   b) Vuelve a escribir el código del productor-consumidor (del código de la semana pasada), ahora con el tipo de datos `MySemaphore` para controlar la cola de bloqueo.

   c) La API de Java viene con una implementación de un semáforo. Reemplaza `MySemaphore` con la implementación de `Semaphore` de Java. Compara los resultados de cada una de las implementaciones. Explica el resultado ¿Se comporta como se esperaba?

2. **(P3: para entregar dentro de una semana)**: Executors. El propósito de este problema es explorar algunas características muy básicas del servicio Executor. Para esto, exploramos dos versiones diferentes del uso del Ejecutor: una con `run()` y la otra con `call()`.

a) **Runnable:** Escriba una tarea ejecutable que sobreescribe el método `run()` para calcular el factorial. En la función Main, use las interfaces `Executor` y `Executors` junto con el método `execute` para calcular factorial de N números, cada uno generado aleatoriamente. En este ejemplo, Recuerde que un generador de números aleatorios se hace con el clase `Random`.

- Lanzar el `executor` con el número de procesadores disponible usando `availableProcessors`
- La función factorial se puede escribir de la siguiente manera, usando un modo de espera para simular más tiempo de cálculo.

```
// Si el numero es  0 o 1, devolver  1 valor
if ((num==0)||(num==1)) {
    result=1;
} else {
    // Else, calcular la factorial
    for (int i=2; i<=number; i++) {
        result*=i;
        Thread.sleep(20);
    }
}
```

- Cada tarea `Runnable` debe imprimir el valor o proporcionar un método para acceder a su resultado.
- Supervise el progreso de los hilos en el método `Main` (utilizando un bucle que esté activo mientras el los hilos siguen funcionando).
- Después de apagar el Executor, ¿qué sucede si intenta ejecutar un nueva tarea?

b) **Callable; Hilos que devuelven valores:** Ahora la tarea Runnable devolverá los resultados de la factorial. Esto se hace con el método de `call()`. Esto es implementado por:

```
public class FactorialTask implements Callable<Integer>
```

De esta manera, el método `call()` de `FactorialTask` puede devolver el valor del factorial a un objeto `Future`.

- En particular, en el código principal `Main`, los resultados se pueden recopilar en una lista de objetos `Future`, definidos de la siguiente manera:
  `List<Future<Integer>> resultList=new ArrayList<>();`
- La llamada a `submit()` del executor devuelve un objeto Future<Integer>que se puede almacenar como:
  `Future<Integer> result=executor.submit(calculator);`
  y añadido a la lista de objetos del `Future`: `resultList.add(result);`
- Usando un thread-pool de tamaño fijo (usando el método `newFixedThreadPool()` del `ThreadPoolExecutor`), debe lanzar las tareas de `FactorialTask`.
- Desde el código principal, verifique continuamente el estado de las tareas en bucle usando el método `getCompletedTaskNumber()` del executor:
- De esta manera, debe recopilar la información de cada tarea para determinar si se ha completado o no (verdadero | falso). Escribe un mensaje indicando si las tareas que gestiona han finalizado o no. Esto se puede hacer usando el método `isDone()`, por ejemplo:

```
for (int i=0; i<resultList.size(); i++) {
     Future<Integer> result=resultList.get(i);
     System.out.printf("Main: Task %d: %s\n",i,result.isDone());
}
```

- Escribe los resultados obtenidos por cada hilo/tarea. Para cada objeto Future, obtiene el valor de la factorial devuelto, utilizando el método `get()`. Como ejemplo, parte de la salida podría ser como lo siguiente:

```
Main: Number of Completed Tasks: 8
Main: Task 0: true
Main: Task 1: true
Main: Task 2: true
Main: Task 3: true
Main: Task 4: true
Main: Task 5: true
Main: Task 6: true
Main: Task 7: true
Main: Task 8: false
Main: Task 9: false
Main: Results
Core: Task 0: 120
Core: Task 1: 720
Core: Task 2: 1
Core: Task 3: 720
Core: Task 4: 5040
Core: Task 5: 1
Core: Task 6: 40320
Core: Task 7: 720
Core: Task 8: 2
Core: Task 9: 6
```