Programación Concurrente

Trabajo Práctico

Se desea implementar en Java usando métodos *synchronized* una clase "monitor" que encapsule el comportamiento del Juego de la Vida de John Conway. El juego de la vida es una simulación en un universo en forma de grilla bidimensional en donde cada celda puede estar viva o muerta. La simulación progresa generación tras generación siguiendo los cambios en las celdas de la grilla a partir de una configuración inicial. Cada celda interactúa con sus 8 celdas vecinas siguiendo las siguientes reglas:

1. Una celda viva con menos de dos vecinos vivos muere de soledad.
2. Una celda viva con dos o tres vecinos vivos, continua viviendo.
3. Una celda viva con más de tres vecinos vivos, muere por sobrepoblación.
4. Una celda muerta con exactamente tres vecinos vivos se convierte en una celda viva por efecto de reproducción.

Junto con el enunciado se provee una interfaz gráfica (adaptada de la publicada por Edwin Martin) que utiliza la grilla y la muestra por pantalla. El código entregado está compuesto por tres paquetes. El paquete gameoflife contiene los archivos necesarios para levantar la interfaz gráfica y no deben ser modificados. El paquete shapes contiene utilidades para configuraciones iniciales predefinidas. El paquete solution está destinado a contener todos los archivos que consideren necesarios para implementar la solución.

La solución debe respetar la siguiente estructura:

1. Una clase `Buffer` (implementada como un monitor utilizando métodos synchronized) que actúa como una cola FIFO concurrente de capacidad acotada. Es decir, bloquea a un consumidor intentando sacar un elemento cuando está vacía y bloquea a un productor intentando agregar un elemento cuando está llena. La capacidad del Buffer debe ser un parámetro configurable.
2. Una clase `Worker` que extiende de `Thread` y realiza la actualización de la grilla. Un `Worker` debe tomar una región sobre la cual trabajar de un `Buffer` conocido al momento de su creación. Si la región es inválida el `Thread` debe preparase para finalizar su ejecución.
3. Una clase `ThreadPool`, que se encarga de instanciar e iniciar la cantidad de `Workers` pedida por un usuario.
4. Cualquier otra clase auxiliar que considere necesaria.
5. Una clase `GameOfLifeGrid` que implemente la interfaz utilizada por la interfaz gráfica que se presenta a continuación. Tenga en cuenta que la solución sólo será correcta si distribuye el trabajo equitativamente entre todos los threads y genera el mismo resultado independientemente de la cantidad de threads utilizados.

Al iniciar la simulación se debe delegar la iniciación de los threads necesarios en la clase `ThreadPool` y luego en cada iteración se debe introducir de a una las regiones a resolver en el `Buffer`. Cada `Worker` en funcionamiento debe tomar regiones de a una del `Buffer` y realizar la actualización. Cabe destacar que es inadmisible utilizar una cantidad de threads menor a la solicitada por el usuario salvo cuando se tienen menos celdas que threads.

```
public interface CellGrid {

  /** Retorna el estado de una celda (si esta viva o no). */
  public boolean getCell( int col, int row );

  /** Setea el estado de una cedla (viva o no). */
  public void setCell( int col, int row, boolean cell );

  /** Retorna la dimension de la grilla. */
  public Dimension getDimension();

  /** Cambia la dimension de la grilla (manteniendo de
    * ser posible el contenido de las celdas). */
  public void resize( int col, int row );

  /** Limpia la grilla (setea todas las celdas como muertas). */
  public void clear();

  /** Setea el numero de threads a ser utilizado por la simulacion.
    * El trabajo debe distribuirse equitativamente entre todos los
    * threads. */
  public void setThreads(int threads);

  /** Retorna la cantidad de generaciones calculadas desde la
    * ultima llamada a clear. */
  public int getGenerations();

  /** Calcula la siguiente generacion con respecto al estado actual
    * de la grilla. Para ello utiliza tantos threads como se hayan
    * seteado con el metodo setThreads, y distribuye el trabajo
    * equitativamente. El metodo solo debe terminar cuando todos
    * los threads auxiliares hayan terminado su trabajo. */
  public void next();

}
```