Programación Concurrente

Trabajo Práctico

Se desea implementar en Java usando métodos *synchronized* una clase "monitor" que encapsule el comportamiento de una lista de enteros garantizando un correcto acceso concurrente. La lista debe estar implementada usando un arreglo como representación interna para almacenar los valores. Además, no pueden usarse otras estructuras de datos de la librería estándar de java ni del paquete de concurrencia salvo la clase `Thread`. La lista debe proveer las siguientes operaciones:

- `size` que retorna el tamaño de la lista
- `isEmpty` que indica si la lista está o no vacía
- `contains` que indica si la lista contiene o no un elemento dado
- `add` que agrega un elemento al final de la lista (si no hay suficiente espacio en el array debe crearse uno nuevo del doble de tamaño y copiar todos los elementos contenidos en el original)
- `peek` que retorna el primer valor de la lista (sin modificarla)
- `pop` que retorna y remueve el primer valor de la lista (si la lista está vacía debe bloquear al thread ejecutando el método)

Se requiere implementar además un método `sort` que ordene la lista usando una versión concurrente del algoritmo de *mergesort*. El algoritmo *mergesort* ordena una lista dividiendola en dos sucesivamente hasta llegar a listas indivisibles (de longitud menor o igual a uno). Estas listas indivisibles cumplen trivialmente la propiedad de estar ordenadas. El algoritmo luego une sucesivamente pares de sublistas ya ordenadas hasta reconstruir la lista completa. Observe que cada una de las sublistas construidas a partir de la unión de dos sublistas ya ordenadas también está ordenada, por lo que la lista final queda ordenada. En pseudocódigo:

```
mergesort(list) {
  if (list.size() <= 1) return;
  left = list.sublist(0, list.size()/2);
  right = list.sublist(list.size()/2, list.size());
  mergesort(left);
  mergesort(right);
  list = merge(left, right);
}

merge(left, right) {
  result = [];
  while (!left.isEmpty() && !right.isEmpty()) {
    if (left.peek() <= right.peek())
      result.add(left.pop());
    else
      result.add(right.pop();
  }
  result.addAll(left);
  result.addAll(right);
  return result;
}
```

La versión concurrente de este algoritmo utiliza threads de forma tal que las llamadas recursivas se hagan "simultáneamente". No obstante, al ordenar listas grandes la cantidad de threads que se levantan puede resultar perjudicial, por lo que el usuario de la lista debe indicar cuál es la máxima cantidad de threads que pueden estar activos al mismo tiempo (**debe ser un parámetro del método `sort`**). Cabe destacar que es inadmisible ordenar la lista (o cualquiera de las sublistas) con una cantidad menor de threads que la máxima permitida salvo cuando se tienen menos elementos que threads permitidos.