Trabalho 2 de Programação Concorrente: Tabelas de dispersão

Murilo Rosa (up201900689)
Nazar Berbeka (up201907148)

Junho 2022

Conteúdo

Introdução

O objetivo deste trabalho prático é desenvolver quatro implementações concorrentes de tabelas de dispersão em Java, utilizando diferentes mecanismos para sincronização de acessos aos mesmos. A implementação de tabelas de dispersão usa um array de listas ligadas para guardar os elementos e o hash code do elemento para calcular em que lista este será inserido. Neste relatório são descritas as quatro implementações.

Implementação

As implementações das tabelas são baseadas numa classes já fornecida `HSet0` e contém as seguintes funções:

- `size` - devolve o tamanho do conjunto;
- `add` - adiciona o elemento ao conjunto;
- `remove` - remove o elemento do conjunto;
- `contains` - testa se o elemento é contido no conjunto;
- `waitFor` - espera enquanto o elemento não está no conjunto;
- `rehash` - redimensiona a tabela.

HSet1

Esta implementação usa `ReentrantLock` e condições (`Condition`) para sincronização de acessos a tabela em vezes dos blocos `synchronized`. A implementação desta classe é idêntica ao do `HSet0`, mas com todos os blocos `synchronized (this)` substituídos por blocos `try/finally`: a aquisição do lock ocorre antes do bloco `try` e a sua libertação ocorre no bloco `finally`. Uma outra diferença na implementação é nas funções `waitFor` e `add`: as instruções `wait()` e `notifyAll()` foram substituídas por uma condição `wait_for_elem`. A função `add` notifica todos os threads quando o elemento é adicionado ao conjunto, enquanto `waitFor` espera enquanto este elemento não estiver na tabela.

HSet2

Tal como `HSet1`, esta implementação usa a classe `ReentrantReadWriteLock`, que é similar a `ReentrantLock` mas com 'locks' de leitura (`ReadLock`) e escrita (`WriteLock`) separados. No caso da condição nas funções `add` e `waitFor` continuamos a usar `Condition`, que são compatíveis com `WriteLock`. A vantagem nessa implementação deve-se ao fato de permitirmos múltiplos acessos paralelos nos 'locks' de leitura. Por isso, usamos o 'lock' de escrita apenas onde necessário,

nomeadamente em `add`, `remove`, `waitFor` e `rehash`. Para além disso, seguimos o mesmo padrão de adquirir o 'lock' e processar os dados dentro de um bloco `try/catch`.

HSet3

Para esta implementação usamos também `ReentrantReadWriteLock`, entretanto aproveitamos o fato de usarmos um array de listas ligadas para termos 'locks' de leitura e escrita separados por lista ligada. Inicialmente temos um `ReentrantReadWriteLock` associado a cada lista ligada, e em caso de `rehash`, associamos a um conjunto de listas ligadas. Assim, usamos o mesmo padrão de blocos `try/catch`, exceto pelo fato de que precisamos obter apenas o 'lock' associado a devida lista ligada e operação. No caso das condições usadas em `add` e `waitFor`, teremos uma condição (`Condition`) associada a cada `ReentrantReadWriteLock`. Apenas nas funções de `rehash` e `size`, precisamos obter todos os 'locks' de escrita e leitura, respetivamente.

HSet4

Por fim, esta ultima implementação usa biblioteca scalaSTM e exige manipulação manual dos nós nas listas ligadas. Por isso, a tabela contém um array de nós, em que cada nó contém apontador para o próximo nó e para o nó anterior. Para um adição/remoção de elementos mais simples, todos as listas ligadas são inicializadas com dois nós sentinelas tal como ilustrado na figura 2.1: o primeiro nó aponta para o último; o último aponto para o primeiro. Desta forma, todos as adições/remoções vão ocorrer entre estes dois elementos. Para distinguir os nós sentinelas de nós internos, o campo `value` do nó sentinela é `null`, desta forma é mais fácil testar se o nó é último da lista (`node.value != null` em vez de `node.next.get() != null`).

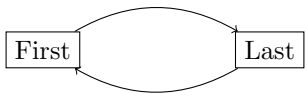

Figura 2.1: Nós sentinelas. O First é o primeiro nó da lista, enquanto o Last é último.

De resto, a implementação desta classe é semelhante à classe `HSet0`, com blocos `synchronized` substituídos por `STM.atomic()` e com manipulação manual de nós. Os comandos `singalAll()` e `wait()` nas funções `add` e `waitFor`, respetivamente, foram removidos, pois os threads são automaticamente sincronizados por STM. A classe também contém funções auxiliares para inicializar as sentinelas (`set_sentinels`), adicionar o elemento no inicio da lista sem verificar se ele está contida na tabela e sem incrementar o tamanho (`add_no_check`) e

para receber o primeiro nó da lista em que o elemento deve ser colocado (`get`). Quando a tabela é redimensionada, criamos um novo array com o dobro do tamanho e substituímos o array corrente por novo array, guardando a referencia para o array velho. De seguida, os nós sentinelas são inicializados e os elementos do array velho são adicionados ao array novo.

Conclusão

Nesse trabalho conseguimos implementar as quatro tabelas de dispersão com todas as funcionalidades pedidas, com todas elas passando com sucesso os teste dados como medida.