Projecto de programação

Eduardo R. B. Marques, DCC/FCUP

2018/19

Contents

1 Introdução

1.1 Sumário

Neste projecto terá de implementar estruturas de dados concorrentes, avaliar o seu desempenho, e em alguns casos validar a sua correção via testes usando a ferramenta Cooperari.

As estruturas de dados em causa compreendem a implementação de TADs para a representação de stacks (`Stack`) e conjuntos (`Set`). É fornecido código base para a implementação, validação e avaliação de desempenho das classes em causa.

1.2 Realização, entrega e apresentação

O trabalho pode ser realizado individualmente ou em grupo por 2 alunos e terá de ser entregue até **3 de Junho**, e apresentado em altura a combinar com os alunos do dia de **5 a 7 de Junho**.

A entrega do trabalho deve ser feita por email ao docente contendo como anexos um arquivo (ex. ZIP) com o seu código e um relatório (ex. formato PDF) a descrever o trabalho que fizeram e resultados de avaliação com um limite de 5 páginas, excluindo página de rosto e possíveis apêndices com fragmentos de "traces" gerados pelo Cooperari caso ache necessário.

Caso não tenha realizado algum dos itens descritos no enunciado abaixo, mencione-o explicitamente no relatório.

1.3 Avaliação

A avaliação terá em conta a qualidade do trabalho desenvolvido em termos de implementação, validação e exposição; e sua exposição no relatório e discussão durante a apresentação.

Para além disso, o trabalho descrito nas secções 2.3.2 e 3.3, algo mais exigente, terá um peso de 20 % na avaliação final.

1.4 Recursos e scripts

O código base e o Cooperari estão disponíveis na página da disciplina, secção *Trabalhos*. Deverá descomprimir ambos os arquivos para directórios diferentes e adicionar a pasta `dir_de_topo/cooperari-0.3/bin` à variável de ambiente `PATH`; defina essa configuração por exemplo em `.bashrc` ou `.bash_profile` da seguinte forma:

```
export PATH=dir_de_topo/cooperari-0.3/bin:$PATH
```

O código fonte está disponível na pasta `pc_projecto/src` e deverá ser compilado para a pasta `pc_projecto/classes`. Use os scripts do Cooperari para o processo de compilação e execução de testes ou programas, executando partir da pasta de topo `pc_projecto`:

- `cjavac` para compilar todo o código;
- `cjunit pc.stack.AllTests` para executar testes Cooperari sobre implementações de `Stack`;
- `cjunitp pc.stack.AllTests` para executar os mesmos testes em modo preemptivo;
- `cjava pc.stack.StackBenchmark` para executar o programa de "benchmark" sobre implementações de `Stack`;

- `cjava pc.set.SetTest` para executar o programa de teste sobre implementações de `Set`;
- `cjava pc.set.SetBenchmark` para executar o programa de "benchmark" sobre implementações de `Set`.

2 Implementações de stacks

Em `src/pc/stack` encontra o interface `Stack` para uma stack com capacidade arbitrária, análoga à discutidas nas aulas, mas com as seguinte diferenças introduzidas para facilitar a validação do código:

- A operação `push(x)` deve apenas aceitar valores `x != null`, e lançar a excepção `IllegalArgumentException` caso contrário;
- A operação `pop()` deve devolver `null` se a stack estiver vazia.

Para ajudar e "inspirar" a implementação de classes pedidas abaixo, são dadas 3 implementações correctas de stacks baseadas no uso de locks:

- `ALinkedStack`: implementação não-bloqueante baseada no uso de uma lista ligada de nós (similar a `AStack` discutida na aula de laboratório 3).
- `LLinkedStack`: implementação bloqueante baseada no uso de uma lista ligada de nós (similar a `LStack` discutida na aula de laboratório 3).
- `LArrayStack`: implementação bloqueante baseada no uso de um array que cresce dinamicamente quando a capacidade do mesmo é excedida.

2.1 Acerto e extensão do código de validação

Se executar

```
cjunit pc.stack.AllTests
```

as 3 classes mencionadas acima serão testadas pelo Cooperari usando os testes codificados em `StackTest`.

Observe que `test3` falha porque as asserções feitas em termos de valores possíveis para a execução não se encontram completamente especificadas.

Analise a forma como os testes estão programados, e:

- Complete o código de `test3` por forma a considerar todas as execuções linearizáveis possíveis. Anote as histórias sequenciais equivalentes em cada caso como é exemplificado.
- Codifique um novo teste `test4` que seja uma variação de `test3` movendo uma das operações à sua escolha da segunda ronda de "fork-join" para a primeira.

2.2 ALinkedStackASR: variante de ALinkedStack

No código de `ALinkedStackASR` programe uma variante de `ALinkedStack`, substituindo o uso de `AtomicReference` e da classe interna `State` pelo uso de de AtomicStampedReference.

Basicamente, um objeto `AtomicStampedReference` permite operações atómicas, incluindo `compareAndSet`, sobre um par `(o,n)` onde `o` é uma referência e `n` é um inteiro.

Valide a correcção da implementação usando os mesmos testes que concebeu para `LLinkedStack` (altere `AllTests` para o efeito).

2.3 Stacks não-bloqueantes baseadas no uso de arrays

2.3.1 Implementações incorrectas

Em `AArrayStackV1` e `AArrayStackV2` são dadas implementações incorrectas (não-linearizáveis) de stacks baseadas em arrays e uso de instruções atómicas.

O que estará errado com as classes? Executando os testes em modo preemptivo, usando o utilitário `cjunitp` em vez de de `cjunit`, é pouco provável que os testes falhem! Mas isso acontece em modo cooperativo ...

Explique o que está errado no código das classes, interpretando os traço de execução em cada caso para cada teste que falha.

Note que as implementações também não lidam com o requisito de redimensionamento do array para guardar os elementos na stack, mas não é por isso que os testes falham; o tamanho inicial de 16 não é em qualquer caso ultrapassado pelos testes.

2.3.2 Programando uma implementação correcta ...

Em `AArrayStack` programe uma implementação correcta de uma stack baseada em arrays. Sugere-se que o ponto de partida seja o código de `AArrayStackV2`.

Pense no uso de instruções atómicas em dois passos distintos, o primeiro garantindo um acesso sem interferências ao estado o objecto, e o segundo libertando esse acesso, que será o ponto de linearização. Nessa "zona de acesso", considere também o requisito de redimensionamento do mesmo em caso de a sua capacidade estar cheia (basta usar código similar ao já dado em `LArrayStack`).

Use o Cooperari para validar a sua implementação, e no relatório descreva sumariamente os traços gerais da implementação.

2.4 Avaliação de desempenho

Use o programa `StackBenchmark` para comparar as várias implementações correctas de stacks (ignore `AArrayStackV1/V2`), e com ou sem back-off exponencial habilitado para as implementações não-bloqueantes.

Atendendo a variações entre execuções, repita as execução 5 vezes e reporte os valores médios observados para cada implementação e variante na forma de uma tabela no relatório. No relatório faça também uma apreciação geral dos resultados, e mencione s características básicas do ambiente em que foram executados os testes: sistema operativo, número de CPU "cores", e memória RAM disponível.

3 Implementações de conjuntos

O interface `Set` em `src/pc/Set.java` exprime um TAD para um conjunto com as seguintes operações:

- `size()`: devolve o tamanho do conjunto;
- `add(e)`: adiciona elemento `e` ao conjunto;
- `remove(e)`: remove elemento `e` do conjunto;
- `contains(e)`: testa se elemento `e` pertence ao conjunto.

Consideramos implementações bloqueantes deste interface usando uma representação interna baseada em uma tabela de "hashing" com o típico esquema de endereço aberto: os elementos do conjuntos são dispersos por entradas na tabela de "hashing", em que cada entrada contém uma lista ligada de elementos implementada por objectos de tipo LinkedList); note que esta classe não tem qualquer mecanismo de sincronização entre threads.

Para simplificar as implementações:

- as operações `add(e)`, `remove(e)` e `contains(e)` lançam `IllegalArgumentException` quando `e == null`;
- a tabela de "hashing" não é redimensionada ("rehashed") como usual quando atinge um certo nível de preenchimento.

Deverá usar e adaptar se achar conveniente o programa `SetTest` para validação básica das suas implementações.

```
cjava pc.set.SetTest
```

Não serão usados testes Cooperari para validação, já que a versão actual não suporta "yield points" em associação a objectos ReentrantLock e ReentrantReadWriteLock que serão necessários nas implementações consideradas abaixo.

3.1 LHashSet

A classe `LHashSet` contém o esqueleto base para a implementação de um conjunto, incluindo um objecto `ReentrantLock` que é inicializado mas não usado no resto do código. Modifique o código por forma a que cada operação seja devidamente protegida pelo uso do `ReentrantLock`.

3.2 Uso de um lock por entrada na tabela de hashing

Considere agora uma evolução de `LHashSet`, onde em vez de um lock global, usamos um lock por entrada na tabela de "hashing" de tal forma que uma thread só bloqueia se houver outra thread a aceder à mesma entrada da tabela. Podemos então ter paralelismo nas operações entre threads diferentes, quando estas não acedem à mesma entrada.

Sugere-se a criação de um array de locks logo no construtor com a mesma dimensão da tabela de "hashing", ou converter as entradas da tabela em pares locks/lista.

No relatório faça um sumário das modificações operadas ao código em relação à primeira versão de `LHashSet`.

3.3 Uso de "read-write" locks

Observe que as operação `add()` e `remove()` podem alterar o estado interno da implementação, mas `contains()` não. Pesquisas simultâneas de elementos sobre o mesma entrada da tabela de "hashing" podem potencialmente ser feitas, desde que não haja outra thread a aceder à mesma entrada para adicionar ou remover elementos.

Para permitir isso, converta a implementação anterior para usar locks do tipo ReentrantReadWriteLock. Em linha com o conceito de "read-write" lock, um objecto `ReentrantReadWriteLock` permite múltiplos locks de leitura ("read locks") em simultâneo, desde que não haja nenhum lock de escrita activo ("write lock"), mas em qualquer caso que apenas uma thread detenha um lock de escrita.

A ideia será então que `contains()` só necessite de adquirir um lock de leitura para a entrada na tabela de "hashing", enquanto que `add()` e `remove()` precisam de adquirir locks de escrita.

No relatório faça um sumário das modificações operadas ao código em relação à versão anterior de `LHashSet`.

3.4 Avaliação de desempenho

Faça uma avaliação para as três implementações usando o programa `SetBenchmark`, com e sem "fairness" habilitada, em moldes similares ao da avaliação conduzida para implementações de `Stack`.

Na execução de `SetBenchMark` as threads aleatoriamente invocam as operações `add()`, `remove()` e `contains()` com probabilidades de respectivamente 10 %, 10 % e 80 %. Numa segunda ronda de avaliação altere as probabilidades para 20 %, 20 %, e 60 %. Apresente os resultados para as 2 rondas de avaliação.