# Relatório Projeto de Programação
# Implementação de Estruturas de dados com suporte para concorrência

Guilherme Martins Amado 201608528
José Pedro Martins Moreira Teixeira de Sousa 201503443

## 1 - Resolução dos exercícios do enunciado

As implementações AArrayStackV1 e AArrayStackV2 não tem semáforos nas atribuições, tanto do método pop() como do método push(E elem), pelo que podem ocorrer de forma desordenada. Isto só acontece em modo cooperativo, já que em mode preemptivo não existe o risco de duas threads estarem a fazer o mesmo ao mesmo tempo.

## 2 - Avaliação e desempenho

Os benchmarks foram executados numa máquina com as seguintes caracteristicas:

- CPU: i7 4 cores e 8 threads
- RAM: 16 GB
- Sistema Operativo: Linux Mint 19 Cinnamon

## 2.1 - Stacks

Sao executados 5 diferentes benchmarks e a média destes está representada na tabela seguinte (Mops/s):

| | Número de Threads | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 |
| LLinkedStack | 33.718 | 7.190 | 10.440 | 8.136 | 8.896 | 7.962 |
| LArrayStack | 35.308 | 7.300 | 10.680 | 8.240 | 9.008 | 8.081 |
| ALinkedStack s/ backoff | 34.678 | 8.344 | 6.708 | 5.182 | 5.407 | 4.930 |
| ALinkedStack c/ backoff | 34.726 | 31.125 | 31.040 | 28.280 | 21.690 | 17.090 |
| ALinkedStackASR s/ backoff | 35.122 | 8.081 | 6.470 | 5.406 | 5.254 | 5.544 |
| ALinkedStackASR c/ backoff | 34.234 | 31.360 | 28.160 | 26.260 | 21.670 | 17.130 |
| AArrayStack s/ backoff | 26.234 | 5.410 | 3.9120 | 3.387 | 2.120 | 1.202 |
| AArrayStack c/ backoff | 26.077 | 25.976 | 23.550 | 23.410 | 17.390 | 14.250 |

Podemos concluir que o LLinkedStack, o LArrayStack o ALinkedStack s/ backoff, ALinkedStackASR s/ backoff e AArrayStack s/ backoff pioram a performance com o número de threads enquanto as restantes implementações quase mantém a performance.
Também se conclui que a diferença que o backoff faz não só tem impacto como também é positiva, no sentido em que ajuda a manter o número de operações pois gere de melhor forma o *busy spinning*.

Numa tentativa de melhor distinguir a performance da LLinkedStack da LArrayStack tentamos aumentar o número de threads por teste. No entanto, não tivemos sucesso dado que ambas se mantém ainda assim bastante semelhantes.

| | Numero de Threads | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| LLinkedStack | 8.35 | 7.80 | 7.94 | 8.38 | 7.58 | 7.32 | 7.40 | 7.50 |
| LArrayStack | 7.78 | 7.81 | 8.24 | 8.32 | 8.21 | 8.52 | 8.19 | 7.59 |

## 2.2 - Sets

### 2.2.1 - LHashSet

A execução deste ficheiro executa sem locks ou qualqueres outros monitores. Um programa que simplesmente faz operações de inserção, remoção a um dado Set.

A média dos resultados finais é representada na tabela seguinte (Mops/s):

| | Número de Threads | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 |
| Sem fairness | 15.016 | 4.281 | 4.086 | 3.170 | 3.474 | 3.100 |
| Com fairness | 13.994 | 0.316 | 0.244 | 0.233 | 0.218 | 0.200 |

### 2.2.2 - LHashSetLocksArray

À semelhança do *LHashSet* mas com o funcionamento através de *locks* da biblioteca *ReentredLock* do Java. Assim, estes locks irão bloquear uma ou mais threads que tentam aceder a uma *critical section*, estando uma já a executar nessa secção.

A média dos resultados finais é representada na tabela seguinte (Mops/s):

| | Número de Threads | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 |
| Sem fairness | 14.256 | 8.241 | 5.182 | 3.410 | 3.711 | 3.784 |
| Com fairness | 13.268 | 7.906 | 4.784 | 3.038 | 2.482 | 2.446 |

### 2.2.3 - LHashSetLoksLockRW

Neste estrutura é usado o *ReentrantReadWrite*, isto é, sempre que uma thread ler da estrutura, faz um bloqueio de leitura (read). Sempre que uma thread escrever, usa um bloqueio de escrita.

Desta forma, sempre que uma operação de escrita acontece, nenhuma outra thread pode ler a estrutura. No entanto, pode haver múltiplas threads a ler ao mesmo tempo, desde que nenhuma esteja a escrever.

A média dos resultados finais é representada na tabela seguinte (Mops/s):

| | Número de Threads | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 |
| Sem fairness | 13.846 | 8.0739 | 5.192 | 3.470 | 4.538 | 4.412 |
| Com fairness | 13.420 | 8.110 | 5.029 | 3.198 | 2.464 | 2.392 |

### 2.2.4 - Conclusões sobre Sets

Concluímos que, no geral, com *fairness* há um menor número de operações devido ao facto de haver muitas mais mudanças de contexto, de forma a não deixar nenhuma thread em starvation. Esta diferença é mais notável no LHashSet e no LHashSetLoksLockRW.

## 3 - Conclusões Finais

A existência de vários métodos de concorrência dá um maior leque de possibilidades de implementação de locks, já que tanto podemos implementar semáforos com variáveis atómicas assim como com locks das bibliotecas java.

Também constatamos que o uso de backoff é bastante benéfico dado que não há “perdas de tempo” pelas threads.

Por outro lado, não podemos constatar que o uso de fairness seja sempre benéfico ou maléfico. Pode ser benéfico distribuir carga pelas várias threads (numa implementação dum servidor em que cada thread corresponde a um cliente não queremos deixar nenhum cliente à espera) mas também pode ser desvantajoso se isto significar que há uma grande perda de operações por thread. Compreende-se portanto ser comum fazer implementações com número fixo de threads, de forma a melhor extrair performance.