Projeto de programação

PROGRAMAÇÃO CONCORRENTE (CC3037), 2019/20

Cristiana Morais | up201505454
Sara Sousa | up201504217
Professor: Eduardo R. B. Marques
DCC/FCUP, 28/06/2020

Índice

Introdução 2

Filas concorrentes 2

Filas baseada em monitores 2

Filas baseada em STM 3

Fila baseada em primitivas atómicas 5

Análise de execução linearizável 6

Avaliação de desempenho 7

Desafio extra 10

Crawler 11

Conclusão 11

Bibliografia 11

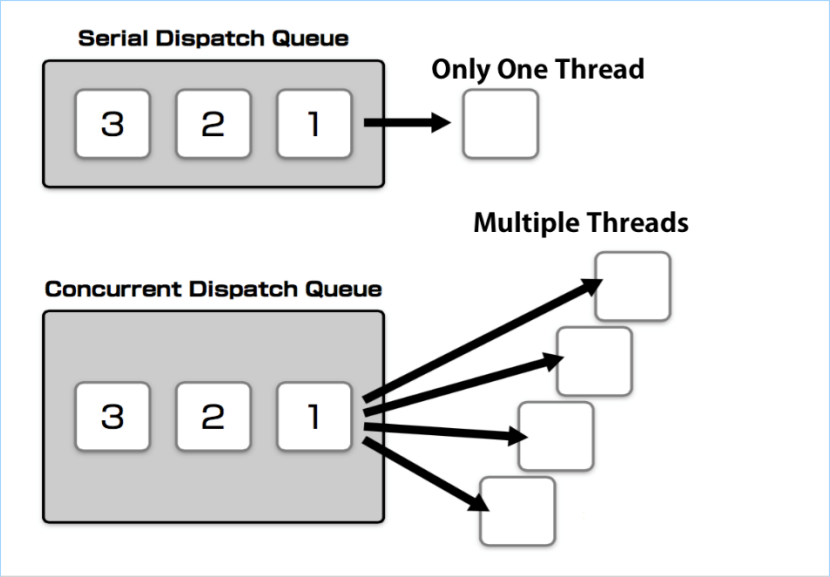


Fig1.

Introdução

Este trabalho foi realizado no âmbito da disciplina de programação concorrente em que se pretendia implementar filas concorrentes, suportadas por um array, usando 3 técnicas de programação "multi-threaded" distintas, Monitores Java, STM e primitivas atómicas sendo que no caso da implementação baseada em primitivas atómicas o código foi modificado de forma a suportar um esquema de "back-off" exponencial.

FILAS CONCORRENTES

- ## FILAS BASEADA EM MONITORES

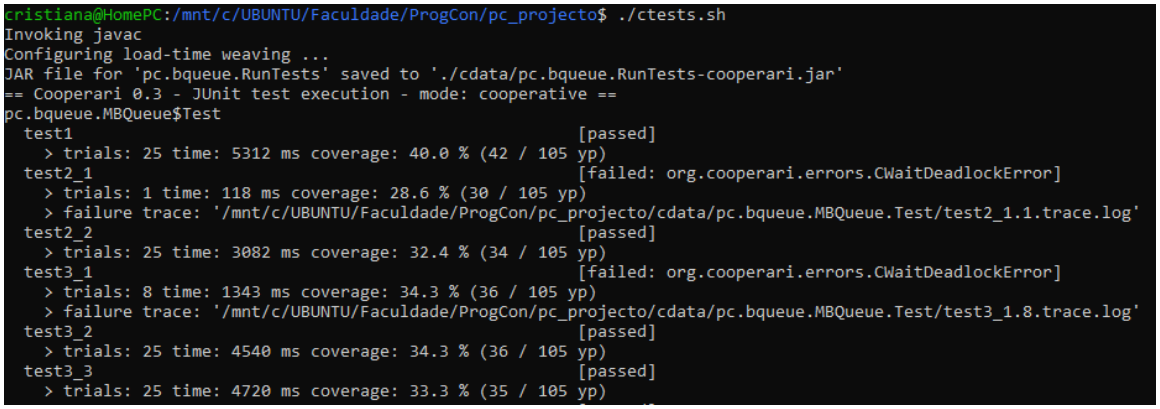
Em MBQueue é dada uma implementação de uma fila com capacidade fixa baseada no uso do suporte "built-in" em Java para monitores/locks. Já em MBQueueU é uma variante de MBQueue.java de forma a suportar filas sem limite de capacidade.

Ao analisar o ficheiro "MBQueue.java" e antes de efetuar qualquer alteração foi possível verificar que existiam alguns erros:

```
cristiana@HomePC:/mnt/c/UBUNTU/Faculdade/ProgCon/pc_projecto$ ./ctests.sh
Invoking javac
Configuring load-time weaving ...
JAR file for 'pc.bqueue.RunTests' saved to './cdata/pc.bqueue.RunTests-cooperari.jar'
== Cooperari 0.3 - JUnit test execution - mode: cooperative ==
pc.bqueue.MBQueue$Test
  test1                                                [failed: java.lang.NullPointerException]
    > trials: 1 time: 360 ms coverage: 40.4 % (44 / 109 yp)
    > failure trace: '/mnt/c/UBUNTU/Faculdade/ProgCon/pc_projecto/cdata/pc.bqueue.MBQueue.Test/test1.1.trace.log'
  test2_1                                              [failed: org.cooperari.errors.CWaitDeadlockError]
    > trials: 2 time: 178 ms coverage: 35.8 % (39 / 109 yp)
    > failure trace: '/mnt/c/UBUNTU/Faculdade/ProgCon/pc_projecto/cdata/pc.bqueue.MBQueue.Test/test2_1.2.trace.log'
  test2_2                                              [failed: java.lang.NullPointerException]
    > trials: 2 time: 164 ms coverage: 34.9 % (38 / 109 yp)
    > failure trace: '/mnt/c/UBUNTU/Faculdade/ProgCon/pc_projecto/cdata/pc.bqueue.MBQueue.Test/test2_2.2.trace.log'
  test3_1                                              [failed: org.cooperari.errors.CWaitDeadlockError]
    > trials: 1 time: 167 ms coverage: 30.3 % (33 / 109 yp)
    > failure trace: '/mnt/c/UBUNTU/Faculdade/ProgCon/pc_projecto/cdata/pc.bqueue.MBQueue.Test/test3_1.1.trace.log'
  test3_2                                              [failed: org.cooperari.errors.CWaitDeadlockError]
    > trials: 1 time: 196 ms coverage: 30.3 % (33 / 109 yp)
    > failure trace: '/mnt/c/UBUNTU/Faculdade/ProgCon/pc_projecto/cdata/pc.bqueue.MBQueue.Test/test3_2.1.trace.log'
  test3_3                                              [failed: org.cooperari.errors.CWaitDeadlockError]
    > trials: 1 time: 199 ms coverage: 30.3 % (33 / 109 yp)
    > failure trace: '/mnt/c/UBUNTU/Faculdade/ProgCon/pc_projecto/cdata/pc.bqueue.MBQueue.Test/test3_3.1.trace.log'
```

No teste 1, por exemplo, nota-mos que foi obtido o erro "java.lang.NullPointerException" porque no AtomicInteger a está a ser feito set() com uma queue vazia pois o resultado de q.remove() é null uma vez que um monitor faz enter no método add() executa o primeiro bloco synchronized independente do segundo e a thread é adicionada ao conjunto de espera "wait-set".

Outro possível problema relacionado com o erro poderá ser o facto de uma vez que o lock é libertado, antes de o size ser atualizado, outra thread pode escrever em cima de um elemento que já nao se encontra nao queue, o que não ativa nenhuma exceção.
Eliminado o segundo bloco de synchronized alguns dos erros são corrigidos como podemos observar em baixo.

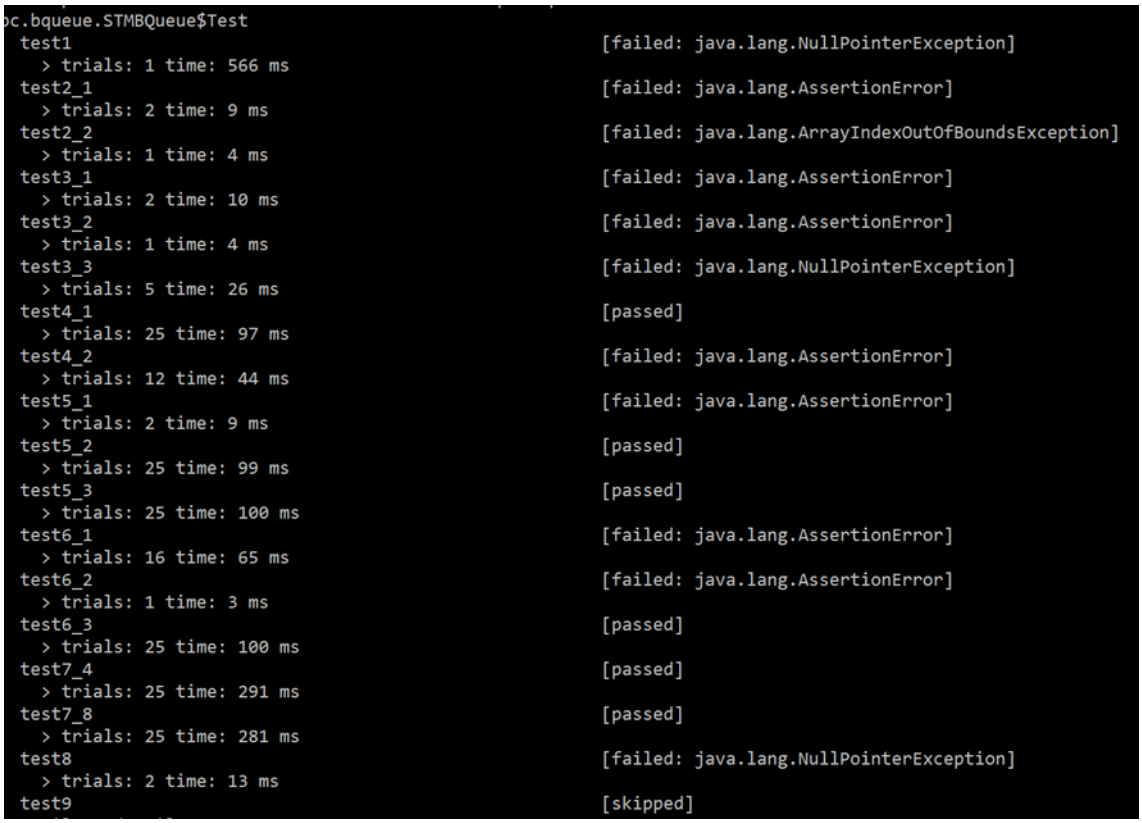


Um outro erro é o "org.cooperari.errors.CWaitDeadlockError" uma vez existem threads ainda "vivas" que ficam à espera indefinidamente. Isto deve-se ao facto de após o wait() ser aplicado um notify, que entrega uma notificação a apenas uma das threads no “wait-set”, de forma aleatória.
Ao ser alterado para notifyAll() verificamos que todos os testes passam.

No caso de MBQueueU as ideias anteriores mantêm-se apenas sendo necessário criar um "newArray" com o dobro do tamanho do array e no final adicionar ao array esse "newArray".

- FILAS BASEADA EM STM

Quando são realizados os testes em modo preemptivo ao ficheiro “STMQueue.java” notamos que existem vários erros, esses erros estão na imagem abaixo:

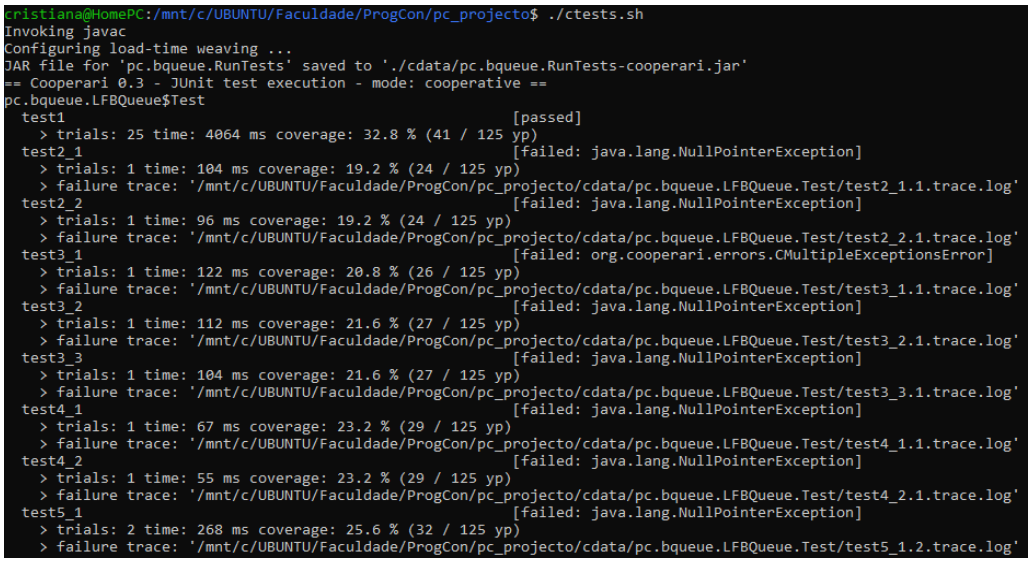


As correções que tiveram de ser feitas foram: na função add acrescentar o STM.increment ao bloco STM.atomic, pois estando de fora esta linha não vai pertencer ao objeto atómico fazendo com que seja possível não ser executado juntamente com as outras partes desse objeto levando a alguns erros acima por começar a ser adicionado outro item à queue mesmo que a ação anterior ainda não tenha feito o incremente ao size dessa queue depois de ter isso adicionado o primeiro item; e na função remove apagar o primeiro bloco STM.atomic que lá estava e colocar o if que lá existia no STM.atomic que estava no return desta função, desta forma, assim como aconteceu com a alteração anterior, a condição vai ficar dentro do objeto atómico e vai levar a que seja executada junto com o resto do mesmo, não havendo o problema de esta ser executado e dar um certo resultado mas depois haja uma mudança e como a condição não é executada outra vez iria dar uma resposta que estava certa antes mas no momento em que essa resposta é utilizada pelo resto do código já não o é.

No caso da STMQueueU a ideia é basicamente a mesma ao anterior só que alem de termos de aumentar o tamanho do array passamos de usar um objeto TArray.View<E> para um objeto Ref.View<TArray.View<E>>.

- FILA BASEADA EM PRIMITIVAS ATÓMICAS

Assim que compilamos os erros que obtemos foram:



De forma a resolver os problemas a ideia foi sugerida foi usar a classe Rooms já continha um parâmetro de backoff e uma lógica de suporte associada.
Usando variáveis head e tail, que são variáveis atómicas e pontos de sincronização, e que quando ativadas garantem que uma ordem na execução sendo que pode causar sobreposição na ocorrência de outras ações em outros métodos o que pode causar erros.
Assim sendo, a ideia de implementação foi que, tendo em conta estes fatores, a cada iteração no while(true), uma thread faz enter num quarto, executa as ações e faz leave desse quarto(em que o foi associado ao metodo add 0, ao remove 1 e ao size 2), sendo que assim que esta entra no quarto nao existe nenhuma condição que limite o número de threads naquele quarto, garantindo que pelo menos uma thread prossegue e a saída no fim do while é garantida, sendo que é garantido também que caso a operação não seja bem sucedida a saída do quarto é garantida visto que como foi referido é necessário garantir que apenas um método possa ser executado ao mesmo tempo, para manter então a ordem correta de execução.

Sobre o uso do backoff é necessário referir que as entradas são feitas dentro de um ciclo de espera ativa que garante que, enquanto a sua funcionalidade não seja concluída, múltiplas chamadas serão feitas a cada entrada na sala, o que ativa este mecanismo presente em Rooms.

Já em LBQueueU é implementada de forma a suportar filas sem capacidade fixa. Para lidar com o redimensionamento do array foi usado um objecto AtomicBoolean que funcione como "flag" de exclusão mútua no acesso ao array, possibilitando o redimensionamento do mesmo, definido como addElementFlag.compareAndSet(false, true) que coloca as outras threads em espera ativa.

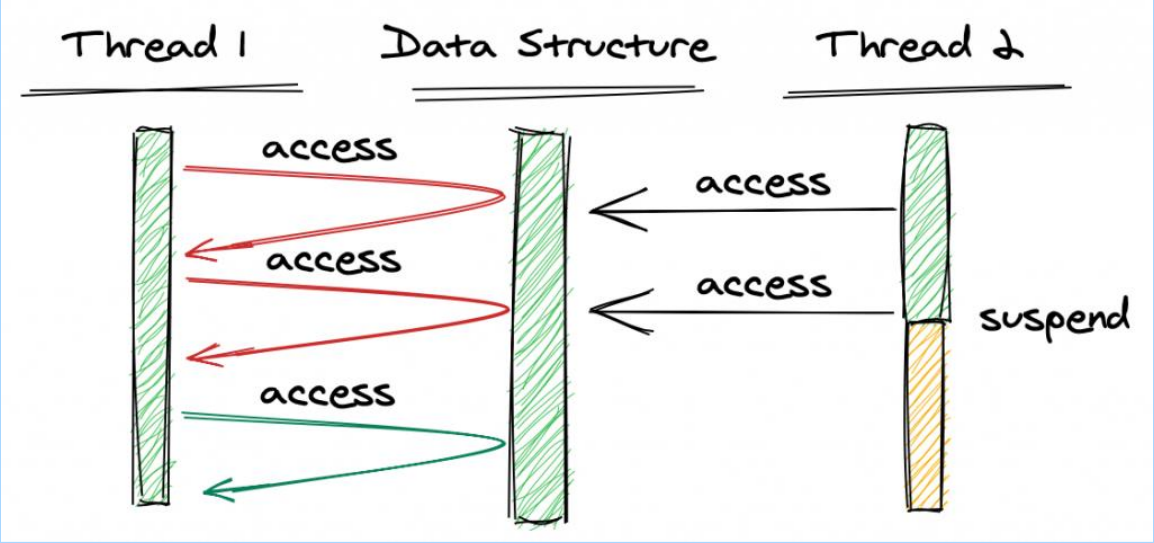


Fig2. ***Exemplo da utilização do Lock Free.***

ANÁLISE DE EXECUÇÃO LINEARIZÁVEL

Em relação a este ponto tentamos fazer, mas sem sucesso.

AVALIAÇÃO DE DESEMPENHO

Os benchmarks foram executados cinco testes em duas máquinas, com as caraterísticas abaixo descritas, fazendo um total de 10 testes, e estes foram realizados sem nenhum programa aberto. A ordem de grandeza são thousand ops/s per thread .

- **Cristiana:**

CPU : Intel® Core (TM) i7 -3630QM CPU @ 2.40GHz com 4 Cores
RAM : 8,00 GB
Sistema operativo: ubuntu 18.04 LTS em Windows 10

Nº THREADS	IMPLEMENTAÇÃO	TESTE 1	TESTE 2	TESTE 3	TESTE 4	TESTE 5
2	MBQueueU	1484.55	1334.29	1270.73	1304.40	1360.87
2	backoff=y, LFBQueueU	1324.08	2314.95	857.59	1470.08	1437.14
2	backoff=n, LFBQueueU	1153.93	1094.68	1089.71	1100.46	1140.20
2	STM, STMBQueueU	47541.00	46919.73	41552.74	43086.02	47296.80
4	Monitor-based, MBQueueU	619.16	837.15	799.98	771.02	846.88
4	Lock-free backoff=y, LFBQueueU	832.12	840.00	875.84	905.41	760.46
4	Lock-free backoff=n, LFBQueueU	436.84	448.70	433.07	456.65	433.83
4	STM, STMBQueueU	35549.56	37462.57	27551.94	46713.08	28573.90
8	Monitor-based, MBQueueU	314.36	395.19	404.19	417.36	419.14
8	Lock-free backoff=y, LFBQueueU	428.73	399.77	415.12	444 43	392.81

8	Lock-free backoff=n, LFBQueueU	199.82	190.79	196.19	196.93	186.51
8	STM, STMBQueueU	21944.67	22331.38	20311.18	23674.23	21768.44
16	Monitor-based, MBQueueU	149.43	149.98	142.69	154.71	150.88
16	Lock-free backoff=y, LFBQueueU	212.36	218.06	206.42	185.39	213.81
16	Lock-free backoff=n, LFBQueueU	80.25	82.85	79.01	79.53	77.21
16	STM, STMBQueueU	10972.64	10843.98	10287.12	11553.69	10710.31
32	Monitor-based, MBQueueU	74.12	71.87	70.37	74.07	74.14
32	Lock-free backoff=y, LFBQueueU	87.94	84.51	94.81	130.77	92.79
32	Lock-free backoff=n, LFBQueueU	30.55	29.41	31.01	33.80	30.04
32	STM, STMBQueueU	5068.96	5212.96	5228.29	5858.32	5319.41

- **Sara:**

CPU : Intel® Core (TM) i7 -8700 CPU @ 3.20GHz com 6 Cores
RAM : 32,00 GB
Sistema operativo: ubuntu 18.04 LTS em Windows 10

Nº THREADS	IMPLEMENTAÇÃO	TESTE 1	TESTE 2	TESTE 3	TESTE 4	TESTE 5
2	MBQueueU	2923.29	3514.79	3746.67	3155.00	3734.72
2	backoff=y, LFBQueueU	3988.93	3712.94	7201.68	5753.27	5682.86
2	backoff=n, LFBQueueU	2693.60	2406.55	1943.13	2290.80	2314.93
2	STM, STMBQueueU	1054.28	3756.53	1270.01	1161.43	15440.59
4	Monitor-based, MBQueueU	1752.51	1121.62	1871.00	1855.94	1362.86
4	Lock-free backoff=y, LFBQueueU	2320.19	2031.57	1194.87	2491.45	2157.43
4	Lock-free backoff=n, LFBQueueU	889.20	866.72	873.08	805.93	1006.98
4	STM, STMBQueueU	351.89	307.33	344.14	352.64	354.17
8	Monitor-based, MBQueueU	685.72	751.75	706.18	782.42	741.72
8	Lock-free backoff=y, LFBQueueU	915.10	1067.73	1119.24	1221.41	1383.98
8	Lock-free backoff=n, LFBQueueU	378.51	388.08	404.05	346.83	373.66
8	STM, STMBQueueU	112.32	107.48	116.04	120.03	115.40
16	Monitor-based, MBQueueU	377.97	390.01	378.49	387.24	384.55
16	Lock-free backoff=y, LFBQueueU	431.61	423.32	460.42	434.77	470.48

16	Lock-free backoff=n, LFBQueueU	145.46	148.82	146.40	159.61	147.57
16	STM, STMBQueueU	43.19	36.66	43.58	45.68	40.86
32	Monitor-based, MBQueueU	195.79	193.85	194.60	193.06	172.20
32	Lock-free backoff=y, LFBQueueU	199.31	193.57	207.86	187.46	253.60
32	Lock-free backoff=n, LFBQueueU	52.53	56.68	61.68	67.06	45.45
32	STM, STMBQueueU	20.27	17.59	21.07	21.54	21.43

Em função dos resultados obtidos podemos concluir que à medida que o número de threads aumenta a diferença entre os resultados de cada implementação é mais notória, em parte poderá ser devido ao facto de o computador nao conter núcleos suficientes para gerir as threads, no sentido em que com o aumento destas a complexidade também é aumentada, bem como um aumento na sincronização entre elas.

Ao longo da execução, também importante referir que houve um certo atraso na apresentação dos tempos especialmente no STM, e como podemos observar nas tabelas acima, esse foi o que obteve mais ops/s, mas também a que obteve uma maior discrepância de valores, à medida que as threads aumentavam, no sentido em que foi baixado esse valor.

Em relação ao caso da utilização do backoff no lock free, este melhorou significativamente o desempenho em comparação com a alternativa de espera ativa.

DESAFIO EXTRA

Não chegamos a realizar este ponto.

CRAWLER

Não chegamos a realizar este ponto.

CONCLUSÃO

Com este trabalho conseguimos aprofundar melhor a implementa filas concorrentes suportadas por um array através destas três técnicas de programação “multi-threaded” . Podemos verificar que o aumento de número de threads faz com que as implementações reduzam o seu desempenho, e que com o uso do backoff este melhora significativamente, quando comparado com a alternativa de espera ativa, e ainda que a utilização de muitas threads significa uma maior sincronização, mas um maior aumento na complexidade de execução da parte do computador.

BIBLIOGRAFIA

IMAGENS:

- https://www.google.com/url?sa=i&url=https%3A%2F%2Fpplware.sapo.pt%2Finformacao%2Funiversidade-do-porto%2F&psig=AOvVaw3Qn1FfihpmIhyVEg-Up8kR&ust=1593443378266000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCKjSruTtpOoCFQAAAAAdAAAAABAK
- https://www.google.com/url?sa=i&url=https%3A%2F%2Fstackoverflow.com%2Fquestions%2F31157224%2Fwhy-calling-dispatch-sync-in-current-queue-not-cause-deadlock&psig=AOvVaw0RzfHpAqA-c01xnZd1ULzW&ust=1593446527216000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCNCwr4LxpOoCFQAAAAAdAAAAABAE
- https://www.baeldung.com/wp-content/uploads/2020/05/threads_lockfree-1024x482-1.png
- As restantes fotos foram prints, que tiramos durante a realização do trabalho, a partir do computador de cada uma de nós.

INVESTIGAÇÃO:

- https://en.wikipedia.org/wiki/Monitor_(synchronization)
- https://pt.wikipedia.org/wiki/Thread_safety
- https://en.wikipedia.org/wiki/Non-blocking_algorithm
- https://en.wikipedia.org/wiki/Software_transactional_memory
- https://en.wikipedia.org/wiki/Compare-and-swap
- Foram também utilizados os slides da página na cadeira que podem ser consultados no seguinte link: https://www.dcc.fc.up.pt/~edrdo/aulas/pc/aulas/
- Foi ainda instalado o cooperari que caso não esteja presente no repositório pode ser encontrado aqui: https://www.dcc.fc.up.pt/~edrdo/aulas/pc/recursos/