

Segundo Exercício-Programa

Implementação de Readers e Writers

Prof. Dr. Clodoaldo Aparecido de Moraes Lima

Clarice Ferreira de Abreu - 102770878

São Paulo

2019

Sumário

1. **Introdução…….. 3**
2. **Resultados…… 6**
3. **Análise e Discussão…… 9**
4. **Referências…… 10**
5. **Anexos…… 11**

1. Introdução

No presente trabalho foi desenvolvido uma aplicação que lê os dados de um arquivo texto e o transforma numa estrutura de dados na forma de um arranjo de Strings. Esse arquivo corresponde ao texto "*A Treatise Concerning the Principles of Human Knowledge*", de George Berkeley (1710), formatado de modo que haja apenas uma palavra (e pontuação satélite) por linha, num total de 36.242 linhas e, portanto, 36.242 itens na estrutura de dados.

A estrutura de dados é então acessada por objetos leitores e escritores que o fazem por meio de *Threads*. Para tal, foi criada uma região crítica para administrar o acesso concorrente desses objetos à estrutura de dados. Essa região é concorrida por 100 *Threads* distintas nas quais misturam-se aleatoriamente leitores e escritores. A aplicação executa o acesso a região crítica variando a proporção de leitores e escritores de 0 leitores e 100 escritores a 100 leitores e 0 escritores. Para cada uma dessas proporções a aplicação realiza o acesso concorrente 50 vezes e tira a média do tempo gasto.

Quando um leitor acessa a região crítica ele lê 100 palavras aleatória do arranjo de Strings e as salva em uma variável local. Quando um escritor acessa a região crítica ele altera 100 palavras aleatorias para "MODIFICADA". Ambos leitores e escritores dormem por 1ms antes de sair da região crítica.

Para tanto, foram desenvolvidas 3 implementações:

1. Bloqueio da região crítica por meio de uma classe *Lock* que impossibilita que duas *Threads* a acessem simultaneamente;
2. Através de *Readers* e *Writers* priorizando os leitores
3. Atráves de *Readers* e *Writers* priorizando os escritores

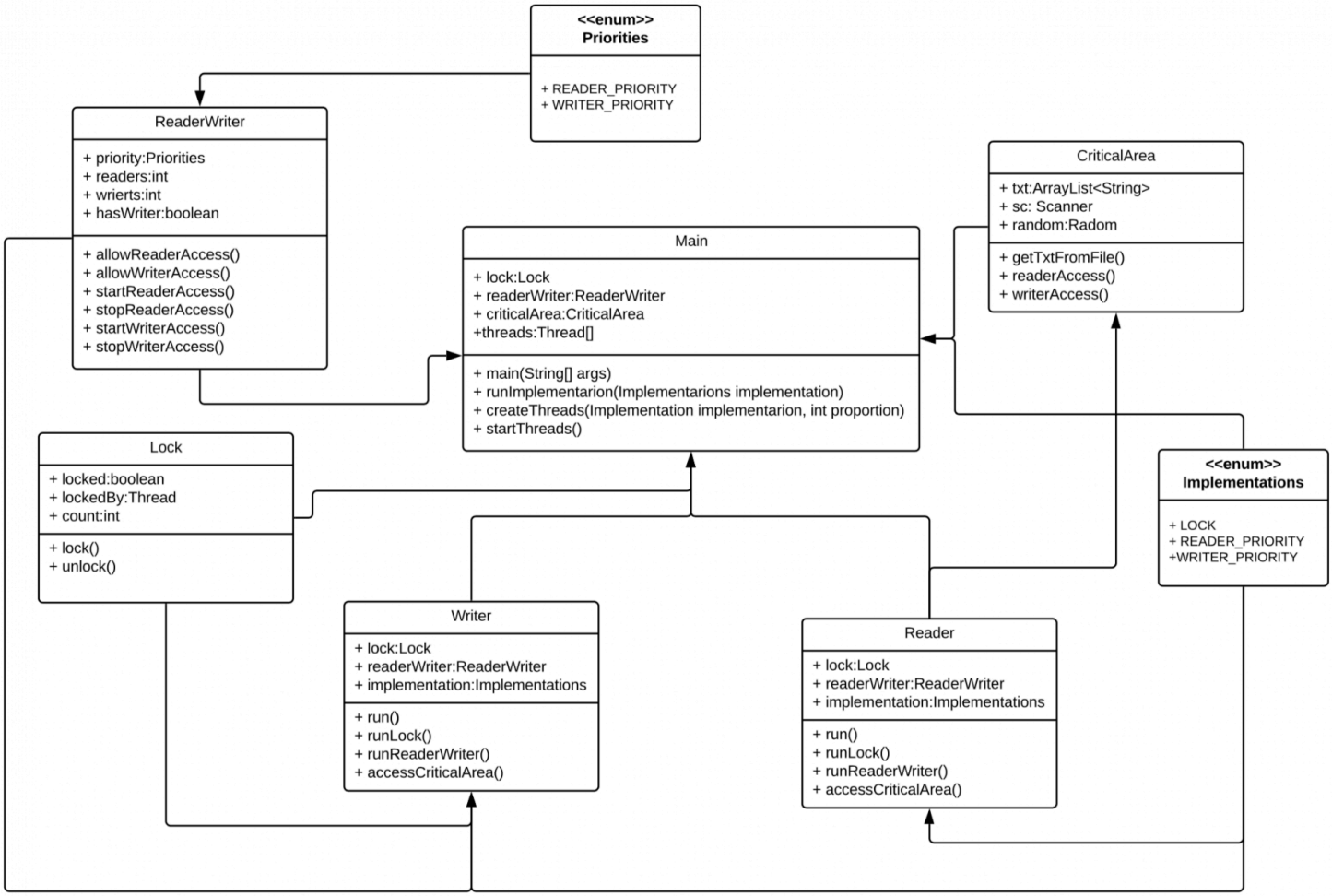


Diagrama 1

Na implementação 1 (Diagrama 2), quando uma *Thread* tenta acessar a região crítica, independente de ela conter um objeto leitor ou escritor, é checado se já há alguma *Thread* região crítica, ou seja, se ela está bloqueada. Se estiver, aguarda até que seja liberada. Uma vez liberada, a *Thread* entra na região crítica, bloqueia-a, executa seu procedimento e desbloqueia-a.

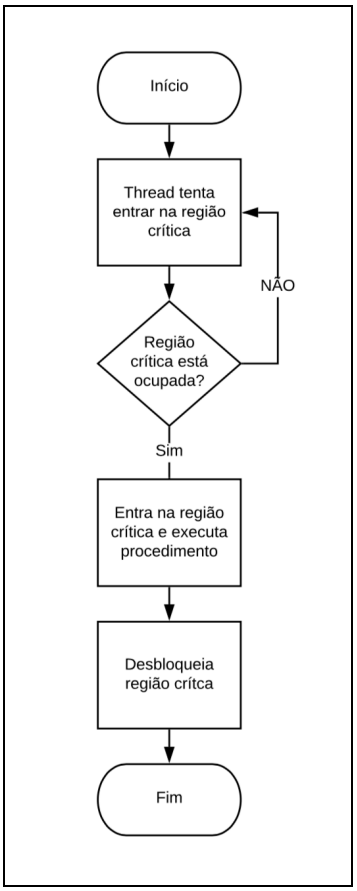


Diagrama 2

Na implementação 2 (Diagrama 3), quando uma *Thread* que possui um leitor tenta acessar a região crítica é incrementado o número de leitores esperando para entrar, então é checado se já há um escritor na região e, se sim, espera ele sair, se não entra na região, faz seu procedimento e saí da mesma decrementando o número de leitores. Já quando uma Thread de escritor tenta acessar a região crítica, é checado se há algum leitor tentando esperando para acessá-la e, se sim, espera até que não haja mais nenhum, se não entra na região crítica, realiza seu procedimento e sai. Múltiplos leitores podem acessar a região crítica simultaneamente, mas não é possível que múltiplos escritores façam o mesmo.

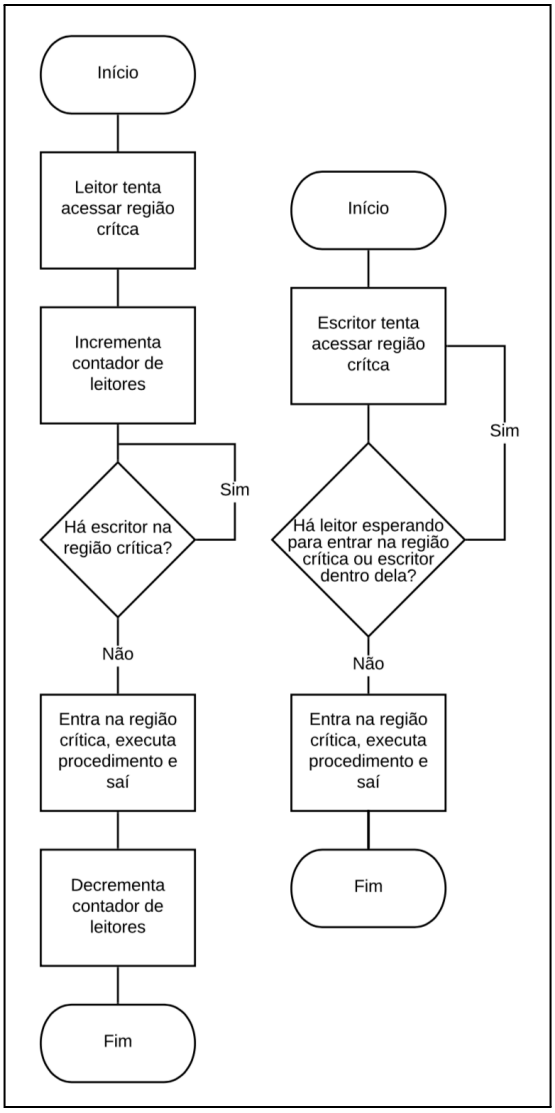


Diagrama 3

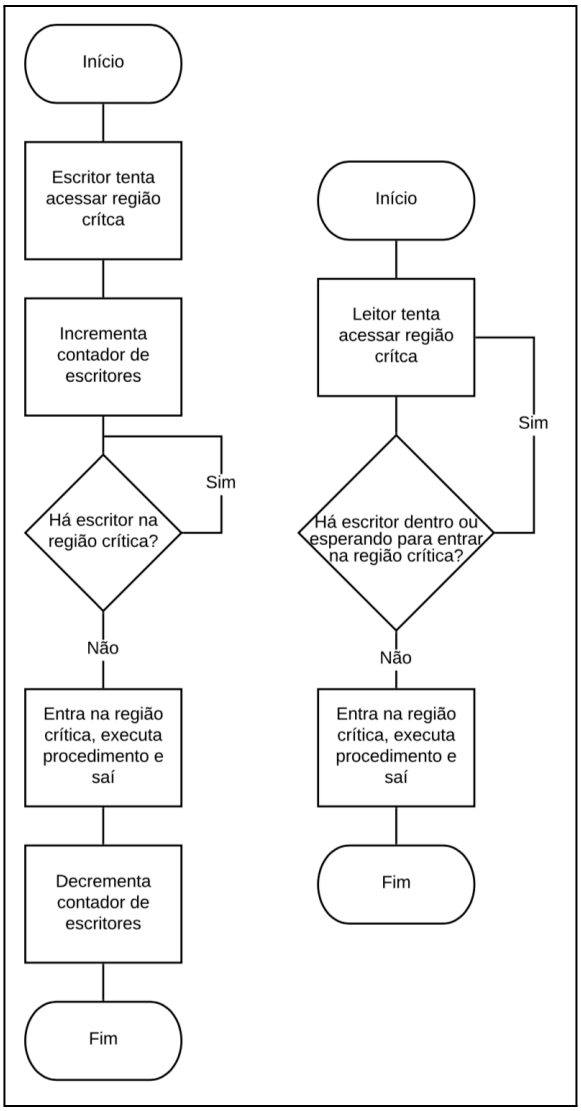


Diagrama 4

Na implementação 3 (Diagrama 4), o procedimento é similar ao da implementação 2, mas dessa vez os escritores têm a preferência, ou seja, quando um escritor tenta acessar a região crítica o contador de escritores incrementa e os leitores só poderão acessar quando esse contador for igual a 0.

2. Resultados

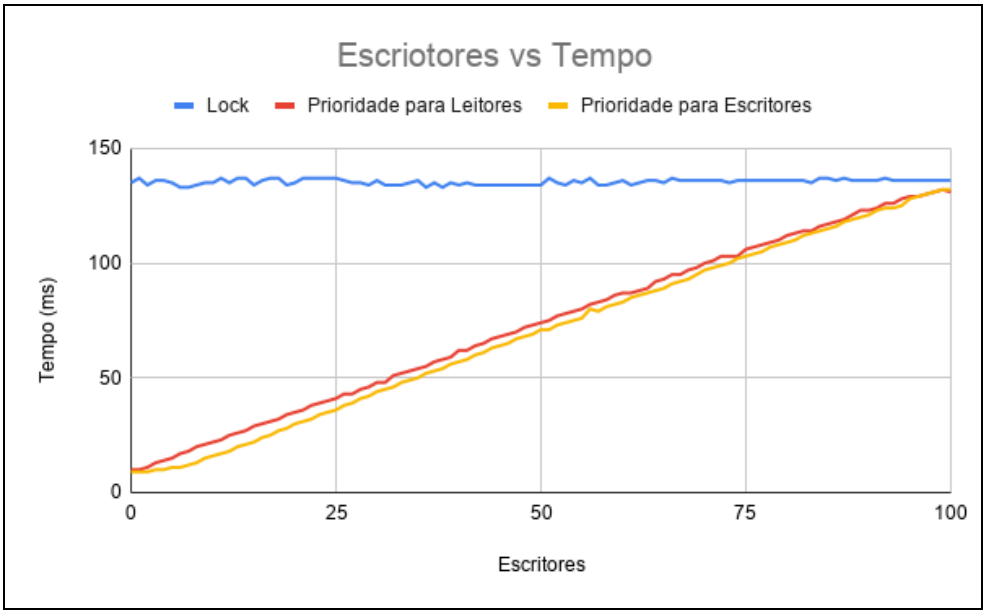
Os resultados da aplicação foram obtidos através de *logs* ao *console*. Após a realização das 50 execuções de cada proporção era impresso no *console* a quantidade

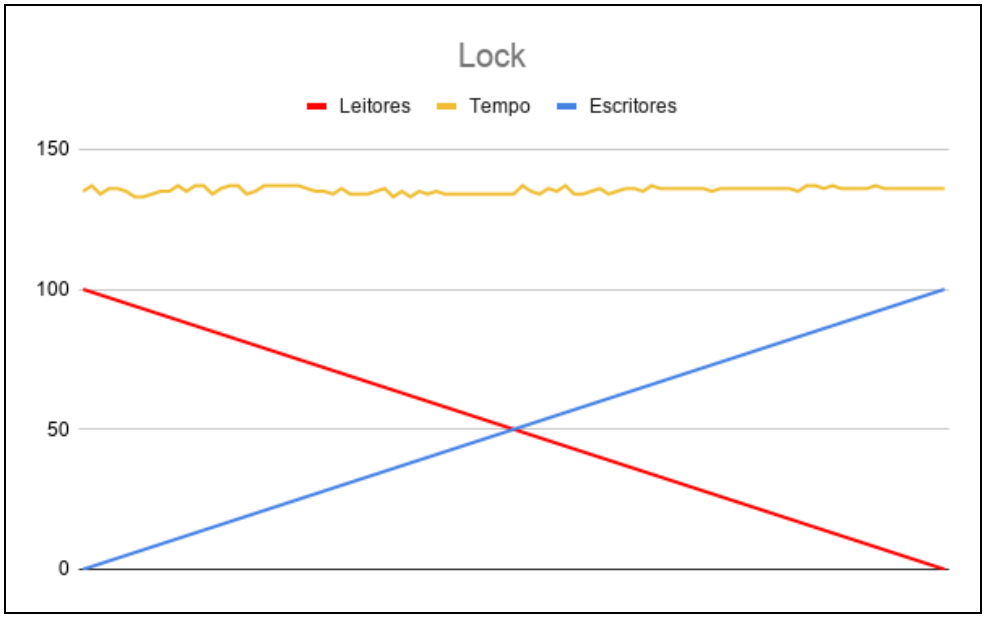
de escritores, leitores e o tempo médio para realização dessas execuções. Os *logs* foram formatados de forma a facilitar a conversão dos resultados em planilhas *.csv*. Todas as planilhas obtidas para demonstração dos resultados encontram-se no anexo deste trabalho (Anexo 1).

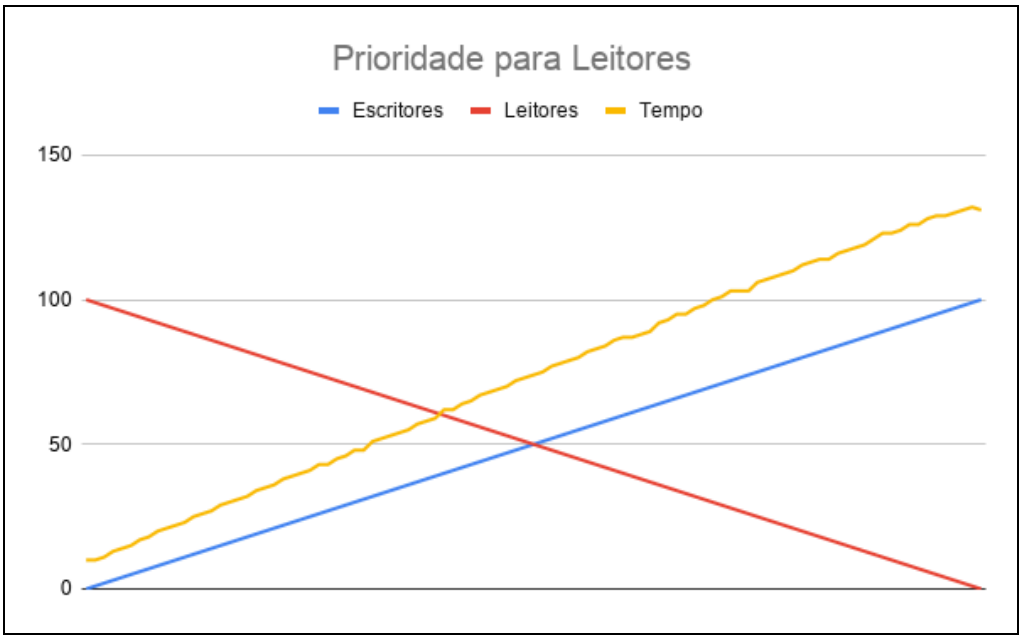
Na implementação 1, utilizando o *Lock* para bloquear toda a região crítica, o tempo de execução médio se manteve aproximadamente constante em torno de 135 ms para todas as proporções. Nas implementações 2 e 3, utilizando *Readers* e *Writers*, o tempo médio cresce proporcionalmente ao crescimento de escritores.

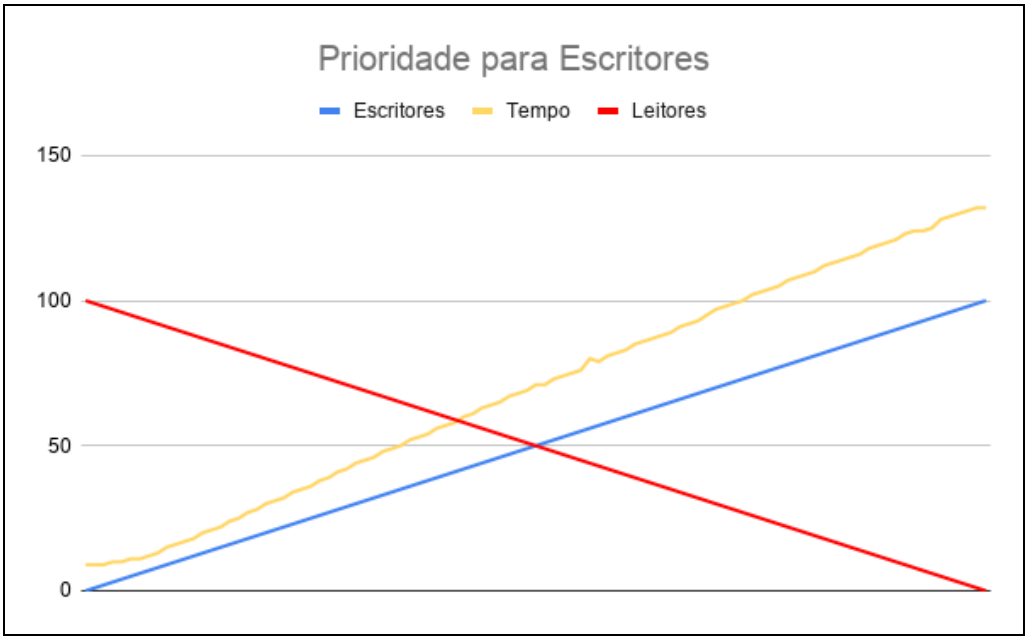
Quantidade		Tempo (ms)		
Escritores	Leitores	Lock	Prioridade para Leitores	Prioridade para Escritores
0	100	135	10	9
5	95	135	15	11
10	90	135	22	16
15	85	134	29	22
20	80	135	35	30
25	75	137	41	36
30	70	136	48	44
35	65	136	54	50
40	60	134	62	57
45	55	134	68	64
50	50	134	74	71
55	45	135	80	76
60	40	136	87	83
65	35	135	93	89
70	30	136	100	97
75	25	136	106	103

80	20	136	112	109
85	15	137	117	115
90	10	136	123	121
95	5	136	129	128
100	0	136	131	132









3. Análise e Discussão

Analisando os gráficos é claro observar que a implementação de *Readers* e *Writers* apresenta vantagem em relação ao tempo médio de execução para acesso da região crítica, principalmente quando há maior número de leitores em relação ao número de escritores. Nesse sentido, quando o número de escritores é baixo, o tempo médio de acesso das 100 *Threads* chegou a ser 13x menor utilizando *Readers* e

Writers em relação uso do *Lock*. Isso se dá porque, nesse tipo de implementação, múltiplos objetos leitores podem acessar a região crítica ao mesmo tempo, agilizando muito o processo. Entretanto, quando o número de escritores passa a crescer o tempo médio tende a se igualar com o da implementação usando *Lock*, uma vez que quando há um objeto escritor na região nenhum outro objeto pode acessá-la, gerando o mesmo comportamento do Lock.

É possível perceber, também, que quando a priorização é dada para escritores e o número desses é baixo, a curva de crescimento de tempo é menos acentuada em relação a priorização por leitores. Entrentato, a medida que o número de escritores vai aumentando essa diferença torna-se cada vez menos significativa.

4. Referências

1. FRIESSEN, Jeff. Java 101: Understanding Java threads, Part 1: Introducing threads and runnables. Disponível em <http://www.javaworld.com/javaworld/jw-05-2002/jw-0503-java101.html>
2. FRIESSEN, Jeff. Java 101: Understanding Java threads, Part 2: Thread synchronization. Disponível em <http://www.javaworld.com/javaworld/jw-06-2002/jw-0607-java101.html>
3. FRIESSEN, Jeff. Java 101: Understanding Java threads, Part 3: Thread scheduling and wait/notify. Disponível em <http://www.javaworld.com/javaworld/jw-07-2002/jw-0703-java101.html>
4. FRIESSEN, Jeff. Java 101: Understanding Java threads, Part 4: Thread groups, volatility, and thread-local variables. Disponível em <http://www.javaworld.com/javaworld/jw-08-2002/jw-0802-java101.html>
5. PINHEIRO, Daniel R. S.. Threads: O Problema de Leitores e Escritores Implementado em C#. Disponível em <https://pt.slideshare.net/rafaelov/artigo-threads-o-problema-dos-leitores-e-escritores-implementado-em-c-rafael-oliveira-vasconcelos>

5. Anexos

1. Planilha com os resultados e gráficos obtidos: https://docs.google.com/spreadsheets/d/11BmSf45dPNE7SWekCPGrSXex0Ecgbm2TdAFKcqtuu7k/edit?usp=sharing
2. Código comentado: https://github.com/clariceabreu/ReaderAndWritersProblem