



UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO



Relatório Leitor/Escritor

Computação Concorrente-2019.2

Professora:

-Silvana Rosetto

Grupo:

-Marcelo Campanelli Nóbrega, DRE: 118067978

-Nathan Vieira Magalhães, DRE: 118038131

Descrição:

Neste trabalho, implementamos a lógica de leitor/escritor com balanceamento equilibrado entre as threads usando a linguagem java e seus

mecanismos de sincronização. Também foi usado java para o programa de verificação.

Programa:

1) Monitor:

No monitor estão presentes as variáveis usadas para condição, e os métodos utilizados pelas threads, que serão chamados através de um monitor. Nesses métodos, as ações das threads são registradas tanto no terminal do usuário quanto em um arquivo txt, para depois ser possível verificar se tudo ocorreu ok com o programa. Vai aqui um exemplo de método do monitor, no caso, o entra leitor (não achamos necessário colocar fotos dos outros métodos pois os casos são análogos e é possível ter o acesso ao código fonte):

```
public synchronized void EntraEscritor() throws InterruptedException {
    //printWriter.println("EntraEscreve()");

    while (escrevendo>0||lendo>0)
    {
        System.out.println("Escritor espera");
        printWriter.println("Espera()");
        wait();
    }
    escrevendo++;
    printWriter.println("Escreve()");
    escriturasseguidas++;
    leiturasseguidas=0;
}
```

2) Leitor/Escritor:

Foram criadas duas classes que estendem threads, leitor e escritor, ambas recebem um id da thread e um monitor em seu construtor. As duas implementam os métodos run e a lógica básica de entrada e saída. Lembrando que as threads leitoras criam um arquivo de saída e nele imprimem o valor que leram, enquanto a classe Escritor só altera o buffer e não precisa de arquivo.

```

class Leitor extends Thread{
    public int id;
    Monitor monit;
    PrintWriter printWriter;

    public Leitor(int idthread,Monitor m) throws IOException {
        id=idthread;
        monit=m;
        FileWriter fileWriter= new FileWriter(id+".txt",false);
        printWriter= new PrintWriter(fileWriter);
    }

    @Override
    public void run() {
        while (monit.getNumLeitura()>0) {
            try {
                monit.Entraleitor();
                System.out.println("Thread [" + id + "] leu o valor : " + programa.bufferescrita);
                System.out.println("Leituras seguidas: " +monit.getLeiturasseguidas());

                printWriter.println(programa.bufferescrita);
                monit.Saileitor();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        printWriter.close();
    }
}

```

3) Balanceamento para evitar inanição:

Naturalmente, os leitores têm prioridade para a lida, já que nenhum leitor bloqueia outro leitor, enquanto os escritores bloqueiam todo mundo. Para evitar uma inanição entre as threads, implementamos uma lógica de que um tipo de thread só pode agir um número máximo de vezes seguidas. Criamos dois contadores, um para contar leituras seguidas, e outro para contar escrituras seguidas, dessa forma, quando alguma dessas variáveis atingem a cota superior, a próxima thread que fosse ser incrementada é bloqueada, até alguma thread diferente executar, nisso ela zera a variável que sinaliza ações repetidas. Aqui vai um exemplo, no método de diminuir as variáveis de condição, decrementamos o número de leituras global e checamos se a variável global “leituras seguidas” ultrapassa seu valor limite, se sim, a thread é bloqueada. Assim, nenhum tipo de thread vai agir mais que 10 vezes seguidas, e nós conseguimos garantir a ausência de inanição.

```
public synchronized void decreaseLeituras()
{
    numleitura--;
    if(leiturasseguidas>10)
    {
        try {
            printWriter.println("protegeseguidaLeitura()");

            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

4) Main:

No programa main, declaramos as variáveis globais assumindo valores passados pela linha de comando, e instanciamos um monitor global onde serão executados os métodos. Criamos os vetores de threads escritoras e leitoras, logo em seguida instanciamos essas threads, depois colocamos para rodarem, depois disso foi feito um join para esperar todas as threads terminarem e ser assim possível fechar o monitor.

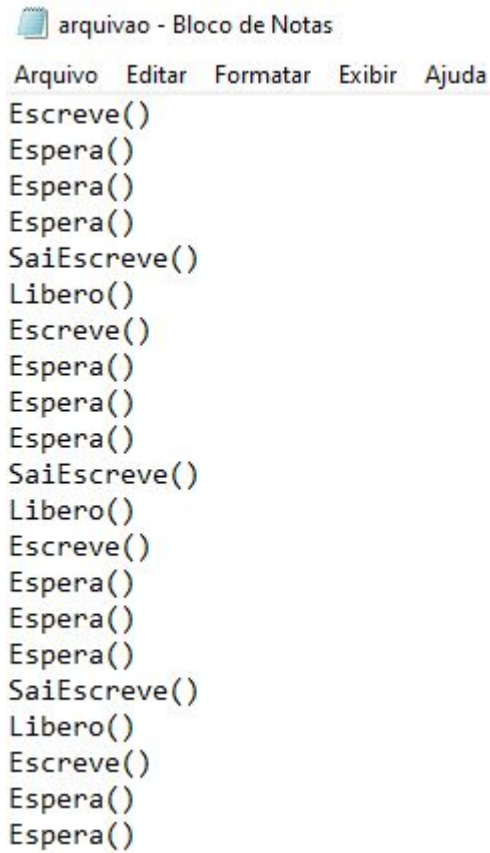
5) Casos de teste:

Rodei o programa com duas threads de cada tipo e coloquei para cada uma fazer 10 leituras/escrituras, com o arquivo de saída “arquivao”, mesmo com a quantidade baixa de threads, foi possível ver o balanceamento funcionar:

```
C:\Users\celoc\facul\trabs\Comp Conc\ultimo\src>java programa 2 2 10 10 arquivao
Leitor espera
Thread [1] Escreveu no buffer : 1
Escritor espera
Escrituras seguidas: 1
Leitor espera
Sai escritor
Leitor espera
Thread [1] Escreveu no buffer : 1
Escritor espera
Leitor espera
Escrituras seguidas: 2
Sai escritor
Leitor espera
Thread [1] Escreveu no buffer : 1
Escritor espera
Leitor espera
Escrituras seguidas: 3
Sai escritor
Leitor espera
Thread [1] Escreveu no buffer : 1
Escritor espera
Leitor espera
Escrituras seguidas: 4
Sai escritor
```

```
Escrituras seguidas: 10
Sai escritor
Esperando por conta de mais leituras seguidas 10
Escritor espera
Thread [1] leu o valor : 1
Thread [2] leu o valor : 1
Leituras seguidas: 2
Leituras seguidas: 2
Thread [1] leu o valor : 1
Thread [2] leu o valor : 1
Leituras seguidas: 4
Leituras seguidas: 4
Thread [1] leu o valor : 1
Thread [2] leu o valor : 1
Leituras seguidas: 6
Leituras seguidas: 6
Thread [1] leu o valor : 1
Thread [2] leu o valor : 1
Leituras seguidas: 8
Leituras seguidas: 8
Thread [1] leu o valor : 1
Thread [2] leu o valor : 1
Leituras seguidas: 10
Leituras seguidas: 10
Esperando por conta de mais leituras seguidas 10
Sai leitor
Esperando por conta de mais leituras seguidas 10
Thread [2] Escreveu no buffer : 2
Escritor espera
Escrituras seguidas: 1
Sai escritor
```

Arquivo de saída:



```
Arquivo  Editar  Formatar  Exibir  Ajuda
Escreve()
Espera()
Espera()
Espera()
SaiEscreve()
Libero()
Escreve()
Espera()
Espera()
Espera()
SaiEscreve()
Libero()
Escreve()
Espera()
Espera()
Espera()
SaiEscreve()
Libero()
Escreve()
Espera()
Espera()
```

Programa verificador:

A lógica do programa verificador foi bem simples, para verificar se o programa funcionou, basta passar o nome do arquivo na linha de comando ao executar o programa, se o programa estiver correto, sairá na tela que tudo ocorreu de maneira esperada. Para o programa ler todo o arquivo e saber o que fazer a cada linha, criamos algumas variáveis globais para checar as condições que as threads proporcionaram, também foi feito um switch case, para cada ação que o programa lê, ele também faz uma, aqui um exemplo, quando o programa

lê “Le()” no arquivo, ele checa inanição ou ação mútua entre threads:

```
static void Le()
{
    if(escrevendo!=0)
    {
        exit("Não é possível ler quando existe alguém escrevendo");
    }
    if(lendoseguidos>15)
    {
        exit("Lendo apos 15 leituras seguidas erro " +lendoseguidos);
    }
    lendo++;
    lendoseguidos++;
    escrevendoseguidos=0;
}
```

Todos os métodos tem suas checagens, é possível ver melhor no código fonte.

Caso de teste:

Rodamos no “arquivao.txt” para checar se tudo ocorreu de boa:

```
C:\Users\celoc\facul\trabs\Comp Conc\ultimo\src>java verificador arquivao
Escreve()
Espera()
Espera()
Espera()
SaiEscreve()
Libero()
Escreve()
Espera()
Espera()
Espera()
SaiEscreve()
Libero()
Escreve()
Espera()
Espera()
Espera()
SaiEscreve()
Libero()
Escreve()
Espera()
Espera()
Espera()
SaiEscreve()
```

...


```
SaiLe()  
Esperaleiseguidas()  
SaiLe()  
Libero()  
Esperaleiseguidas()  
Escreve()  
Espera()  
SaiEscreve()  
Libero()  
Escreve()  
Espera()  
SaiEscreve()  
Libero()  
Le()  
SaiLe()  
Libero()  
Tudo OK com o programa
```

Programa verificador rodou com sucesso, indicando que a execução do programa foi correta.

Conclusão:

Conseguimos implementar o programa e o programa verificador de forma com que as threads leitoras e escritoras agiram de maneira balanceada, cumprindo com o que foi pedido.

