UNIVERSIDADE FEDERAL DE CAMPINA GRANDE - UFCG
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA (CEEI)
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO (DSC)

Disciplina: Programação Concorrente
Professor: Thiago Emmanuel Pereira
Aluno: **José Ivan Silva da Cruz Júnior**
Lucas Christopher de Souza Silva
Período: 2018.2

Lista 1 (Java)

1ª questão

Uma abstração bastante usada em programação concorrentes são os canais. Um canal recebe mensagens enviadas por processos (threads) remetentes. Processos recipientes lêem as mensagens enviadas no canal. Mensagens devem ser lidas na ordem que entraram no canal. Uma vez lida, a mensagem não pode ser lida novamente.

O canal deve ter uma capacidade máxima, ou seja, ao atingir o limite, novas mensagens não podem ser enviadas para o canal imediatamente. Considere que o construtor do canal recebe um inteiro que indica sua capacidade máxima. Mensagens não podem ser descartadas. Implemente a interface abaixo para o canal, usando quaisquer mecanismos de coordenação e controle de concorrência da linguagem Java, exceto as estruturas de dados de *Concurrent Collections.* Considere tanto critérios de corretude quanto de eficiência (p.ex evite spin locks quando possível).

```
public interface Channel {
        public void putMessage(String message);
        public String takeMessage();
}
```

Link do repositório

2ª questão

Considere um sistema que precisa consultar um site web através de uma requisição HTTP. A chamada HTTP é encapsulada por uma API com um único método:

String request(String serverName)

Por motivos de tolerância à falhas, há três mirrors disponíveis com a mesma informação. Nesse sistema, escreva uma função que consulta os três mirrors (cujos

servernames são: "**mirror1.com**", "**mirror2.br**" e "**mirror3.edu**") e retorna a resposta que chegar primeiro. A função deve implementar a seguinte assinatura:

String reliableRequest()

Justifique as decisões importantes em sua implementação. Por exemplo, as primitivas de concorrência usadas.

Link do repositório

3ª questão

Considere uma extensão ao sistema anterior em que você deve escrever uma nova função que retorna o resultado da execução de ***reliableRequest*** ou um erro, se a execução desta durar mais do que 2 segundos.

Link do repositório

4ª questão

Crie uma função que executa indefinidamente a função ***reliableRequest****,* definida anteriormente, enquanto espera uma sinalização de parada enviada por outro fluxo de execução.

Link do repositório

5ª questão

Realize uma comparação de desempenho entre os seguintes pares de estruturas de dados alternativas. Sua resposta deve conter uma descrição dos experimentos realizados, o código para executar os experimentos, bem como o código para analisar os dados experimentais. Além disso, uma breve descrição dos resultados alcançados com plots que os ilustram.

a. ConcurrentHashMap vs Collections.synchronizedMap
b. CopyOnWriteArrayList vs Collections.synchronizedList

Link do repositório

Comentário:

Uma grande diferença entre coleções sincronizadas e a API de *Concurrent Collections* é o desempenho. Isso se dá devido o fato de *Concurrent Collections* relaxarem algumas garantias que as as coleções sincronizadas oferecem. Estas mantêm *locks* por toda a execução dos seus métodos, podendo levar a grandes tempos de espera caso o tamanho das coleções sejam muito grandes ou as operações executadas por *threads* sejam muito custosas.

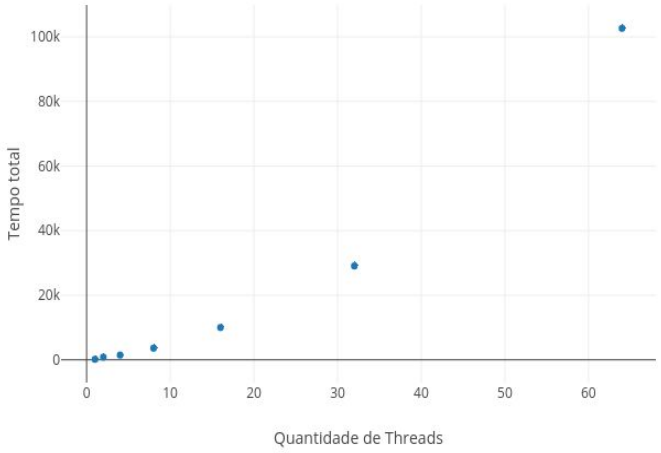
Já em *Concurrent Collections*, as estruturas de dados são especialmente projetadas para casos onde concorrência é uma preocupação. Por exemplo, um *HashMap* pode ser acessado de forma concorrente, caso duas threads não estejam acessando o mesmo bucket , isso leva a uma melhor utilização da estrutura em termos de acesso concorrente.

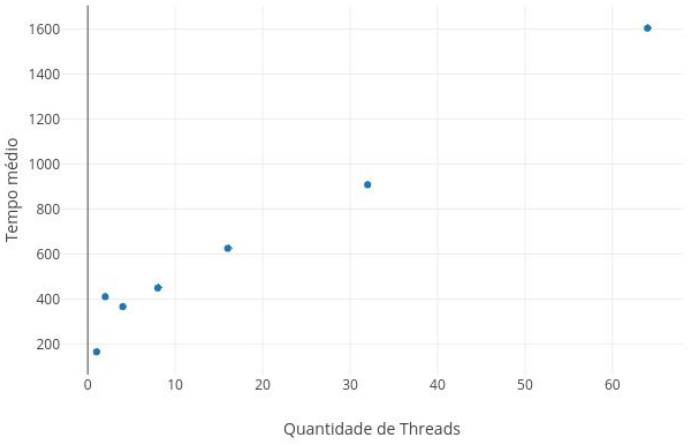
No entanto, tais decisões do *Concurrent Collections* podem levar os métodos a ficarem *weakly-consistent* , isto é, os valores sendo manipulados não refletem o estado da coleção, de fato. Podemos tomar como exemplo dessa característica, um *ConcurrentHashMap*. As operações *size* e *isEmpty* têm uma semântica fraca, ou seja, *size* pode retornar somente o tamanho aproximado e *isEmpty* pode não corresponder à realidade. Segue abaixo, o plot correspondente ao teste de desempenho referente as operações de adicionar e consultar (*put* e *get,* no caso dos mapas e *add* e *get*, no caso das listas, respectivamente).

Considerações:

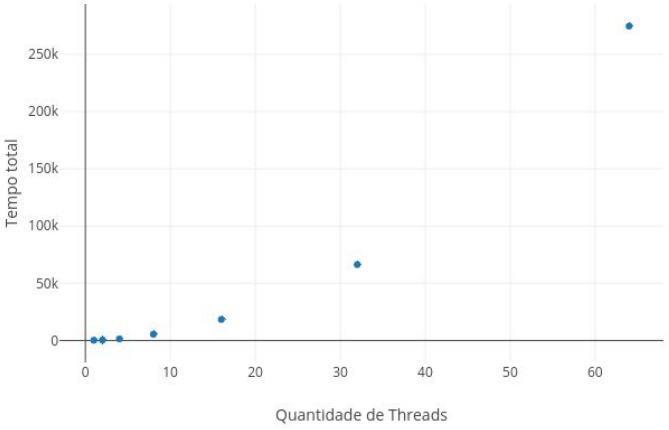
- Quantidade de threads: Tamanho do pool de threads;
- Tempo total: tempo total da operação (em milisegundos);
- Tempo médio: tempo médio da execução de cada thread (em milisegundos);
- Tamanho da entrada para todas as operações: 100.000 (cem mil).

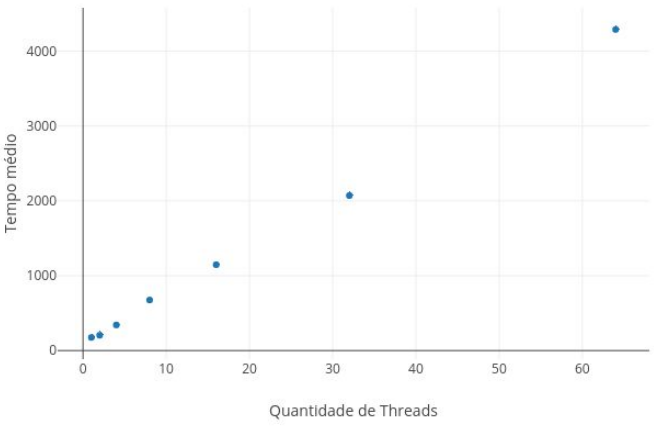
ConcurrentHashMap



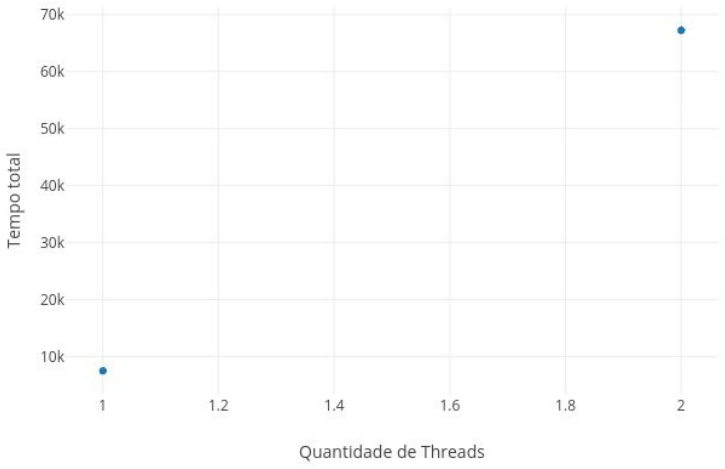


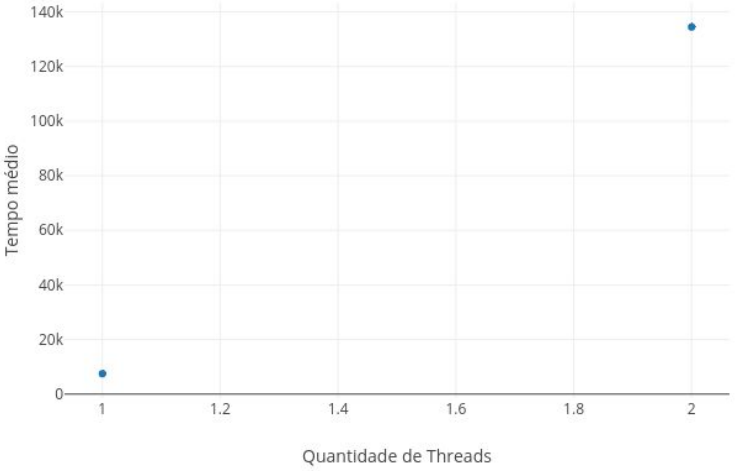
Collections.synchronizedMap



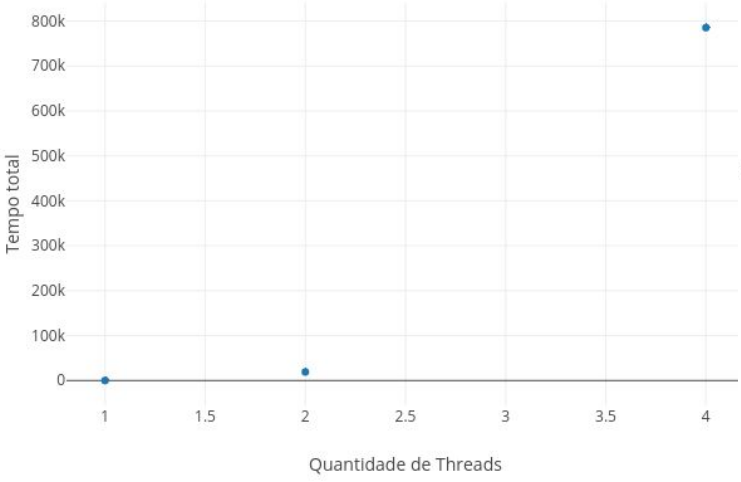


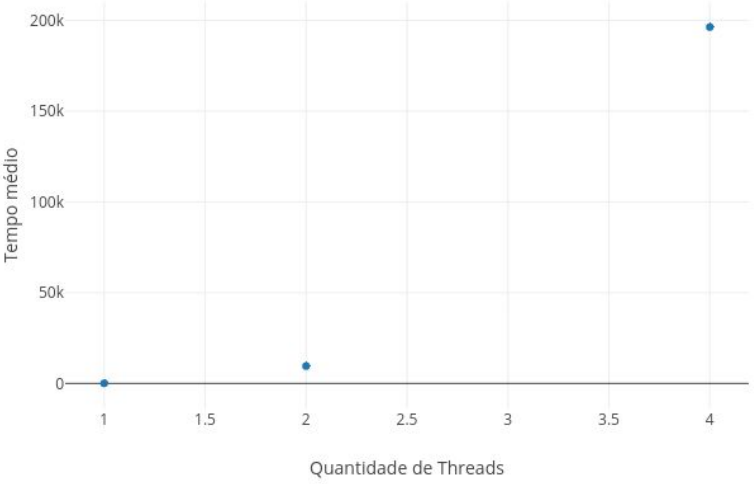
CopyOnWriteArrayList





Collections.synchronizedList





6ª questão

Implemente em Java, um programa em que uma thread gere valores aleatórios, enquanto uma segunda thread verifique se os valores recebidos são pares ou ímpares, e os imprima na saída padrão, caso sejam pares. Os valores devem ser recebidos e impressos na mesma ordem que foram gerados. Todo valor gerado deve ser recebido e impresso, caso seja par (vide o exemplo 2 do material de golang). Não use os objetos do pacote java.util.concurrent

Link do repositório

7ª questão

Construa um pipeline de threads em Java, em que uma thread gere strings aleatórias, enquanto uma segunda filtre as strings que contém somente valores alpha, e uma terceira escreva os valores filtrados na saída padrão. Tal como no exemplo anterior, nenhum valor gerado pode ser ignorado pelo filtro, bem como nenhum valor filtrado pode se ignorado para escrita. Geração, filtro e impressão devem respeitar ordem (vide o exemplo 3 do material de golang). Não use os objetos do pacote java.util.concurrent

Link do repositório

8ª questão

Sobre semáforos, responda as questões abaixo (considere que ***wait*** reduz o valor do semáforo e ***post*** incrementa)

```
wait(S)
<região crítica>
post(S)
<região não-crítica>
```

a) com qual valor o semáforo S precisa ser criado para proteger o acesso à região crítica com exclusão mútua?

Resposta: o valor que semáforo precisa ser criado para proteger a região crítica é 1, pois assim a execução do down será feita, impedindo o acesso a região crítica por outros processos.

b) Se o semáforo for inicializado com o valor 13, quantas threads podem acessar concorrentemente a região crítica?

```
Processo 1 executa:   Processo 2 executa:
while(true) {         while(true) {
    wait(S)               wait(Q)
    a();                      b();
    post(Q);                  post(S)
```

Resposta: 13 threads podem acessar concorrentemente.

c) o que acontece com o código assim de os semáforos são inicializados com: i) 1; ii) 1 e 0; iii) 0

```
Processo 1 executa:   Processo 2 executa:
while(true) {         while(true) {
        wait(Q);              wait(S);
        wait(S);              wait(Q);
        a();                  b();
        post(S);              post(Q);
        post(Q);              post(S)
```

Resposta:

i)
O código será executado em uma espécie de loop alternado entre os processos. O processo 1 tomando a CPU, o semáforo Q e, posteriormente o semáforo S, executarão um wait(), depois uma chamada a a() é feita e incrementados os valores do semáforo S e Q, respectivamente, pelo post(). Em seguida, o processo 2 passa a tomar a CPU, pelo fim do tempo de uso da CPU do processo 1, e faz sua execução, retornando posteriormente para o processo 1 e assim por diante.
ii)
Dependendo do escalonamento feito podemos ter a ocorrência de um deadlock. Pro caso do processo 1 ser executado primeiro o deadlock acontece, já que o valor do semáforo S fazem os dois processos ficarem em espera. Para o processo 2 tomando a CPU e executando primeiro, uma chamada a b() é feita e os valores dos semáforos são incrementados posteriormente.
iii)
Tem-se a ocorrência de um deadlock, pois com os dois semáforos iniciados em 0 tanto o processo 1 quanto o processo 2 entrarão em modo de espera, não podendo serem "acordados" por nenhuma outra chamada, impossibilitando a continuação da execução.

d) o que acontece com o trecho de código acima se os semáforos Q e S são inicializados com 1?

Resposta: A execução do código se dá de forma convencional (fluxo normal de execução), porém quando um dos processos perde a CPU entre a execução de dois wait() pode haver a ocorrência de um deadlock.