Programação Concorrente – Atividade de Fechamento: Sistema concorrente de log em arquivo

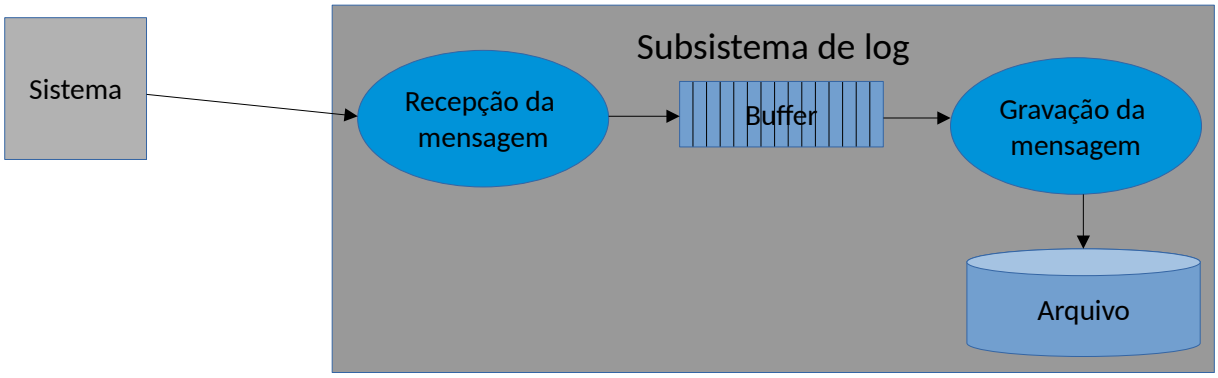
Leitura complementar:

- https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/package-summary.html

O objetivo da aula de hoje é o desenvolvimento de um componente em Java para fazer log de um sistema qualquer em arquivo, utilizando os assuntos finais da unidade. A maioria dos grandes sistemas implementados hoje possuem mecanismos de log para registro de eventos relevantes. Entretanto, se a implementação do serviço de log for feita de uma maneira simplista, o desempenho geral do sistema pode ser afetado. Por exemplo, imagine um servidor Web, onde cada requisição feita por um cliente deve gerar um registro correspondente em um arquivo contendo as informações relevantes de tal requisição (log). Neste exemplo, se a gravação do arquivo for feita no contexto do atendimento do cliente (mesma *thread*), o *overhead* de gravação será percebido pelo cliente, pois operações de gravação em arquivo costumam ser lentas. Neste contexto, seria interessante separar a geração da mensagem de log, da sua efetiva gravação em arquivo, que é o que será feito neste roteiro.

A arquitetura do componente

Como arquitetura do componente de log, considere o diagrama abaixo:



Considerando o diagrama anterior, o subsistema de log recebe a mensagem, armazena em um buffer, o qual é descarregado para arquivo através da gravação da mensagem. Neste diagrama, pode-se perceber que o *buffer* representa exatamente o problema do produtor consumidor visto em aula e solucionado utilizando monitores, *locks* e semáforos. Todavia, agora veremos estratégias de mais alto nível para resolver este problema.

Produtor consumidor com componentes existentes do Java

public interface **BlockingQueue<E>** extends Queue<E>

É uma fila que oferece suporte a operações que esperam que a fila se torne não vazia ao recuperar um elemento e que o espaço fique disponível na fila ao armazenar um novo elemento.

Os métodos de BlockingQueue existem em quatro formas, com diferentes maneiras de lidar com operações que não podem ser satisfeitas imediatamente, mas podem ser satisfeitas em algum ponto no futuro: **um lança uma exceção**, o segundo r**etorna um valor especial** (nulo ou falso, dependendo do operação), o terceiro **bloqueia o thread atual indefinidamente até que a operação possa ser bem-sucedida**, e o quarto **bloqueia apenas por um determinado limite de tempo máximo** antes de desistir. Esses métodos estão resumidos na seguinte tabela:

	Lança uma exceção	Valor especial	bloqueia	Tempo máximo
Inserir	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remover	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examinar	`element()`	`peek()`	*Não aplicável*	*Não aplicável*

Uma BlockingQueue não aceita elementos nulos. As implementações lançam NullPointerException nas tentativas de adicionar, colocar ou oferecer um valor nulo. Um nulo é usado como um valor sentinela para indicar falha nas operações de pesquisa.

Uma BlockingQueue pode ser limitada pela capacidade. A qualquer momento, ele pode ter uma capacidade restante além da qual nenhum elemento adicional pode ser colocado sem bloqueio. Um BlockingQueue sem qualquer restrição de capacidade intrínseca sempre relata uma capacidade restante de Integer.MAX_VALUE. Existem diversas implementações para a interface BlockingQueue, entre elas:

- public class **ArrayBlockingQueue<E>** extends AbstractQueue<E> implements BlockingQueue<E>: Uma fila limitada armazenada em array. Esta fila ordena os elementos na ordem FIFO (first-in-first-out). A cabeça da fila é aquele elemento que está na fila há mais tempo. A cauda da fila é aquele elemento que está na fila há menos tempo. Novos elementos são inseridos no final da fila e as operações de recuperação da fila obtêm elementos no início da fila. Este é um clássico exemplo "buffer limitado", no qual um array de tamanho fixo contém elementos inseridos pelos produtores e extraídos pelos consumidores. Uma vez criada, a capacidade não pode ser alterada. As tentativas de colocar um elemento em uma fila cheia resultarão no bloqueio da operação; tentativas de obter um elemento de uma fila vazia serão bloqueadas de forma semelhante. O construtor desta classe possui assinatura `ArrayBlockingQueue(int capacity)`, que permite especificar a capacidade da fila.

- public class **LinkedBlockingQueue<E>** extends AbstractQueue<E> implements BlockingQueue<E>: Uma fila opcionalmente limitada baseada em nós encadeados. Esta fila ordena os elementos em ordem FIFO (first-in-first-out). A cabeça da fila é aquele elemento que está na fila há mais tempo. A cauda da fila é aquele elemento que está na fila há menos tempo. Novos elementos são inseridos no final da fila e as operações de recuperação da fila obtêm elementos no início da fila. As filas encadeadas geralmente têm um rendimento mais alto do que as filas baseadas em array. O argumento opcional do construtor de limite de capacidade serve como uma forma de evitar a expansão excessiva da fila. A capacidade, se não especificada, é igual a Integer.MAX_VALUE. Os nós encadeados são criados dinamicamente a cada inserção, a menos que isso coloque a fila acima da capacidade de armazenamento do computador.

- public class **SynchronousQueue<E>** extends AbstractQueue<E> implements BlockingQueue<E> Uma fila na qual cada operação de inserção deve esperar por uma operação de remoção correspondente por outra thread e vice-versa. Uma fila síncrona não tem capacidade interna (nem mesmo 1 único elemento cabe). Não se pode examinar uma fila síncrona porque um elemento só está presente quando você tenta removê-lo; você não pode inserir um elemento (usando qualquer método) a menos que outra thread esteja tentando removê-lo; você não pode iterar, pois não há nada para iterar. A cabeça da fila é o elemento que a primeira thread de inserção da fila está tentando adicionar à fila; se não houver tal thread na fila, nenhum elemento estará disponível para remoção e poll () retornará nulo. As filas síncronas são semelhantes aos canais de *Rendez Vous* usados na linguagem Ada. Eles são adequados para padrões de transferência, nos quais um objeto em execução em uma thread deve sincronizar-se com um objeto em execução em outra thread para entregar a ele alguma informação, evento ou tarefa.

Atividade de Implementação

Considerando o problema do log apresentado anteriormente, o buffer de armazenamento atua como ponto de contato entre o produtor e o consumidor, podendo claramente se beneficiar da utilização da interface BlockingQueue (assim como a instância do problema do produtor consumidor vista em aula). Como no exemplo apresentado sobre o servidor Web, não se deseja ter operações bloqueantes no atendimento a um cliente (pelo tempo que isto pode gerar), o que poderia ser solucionado pela utilização de uma fila com capacidade ilimitada, o que será atingido no exercício prático a seguir.

Considere a classe Logger parcialmente apresentada abaixo:

```
public class Logger {

    private static Logger instance = null;
    private final static String logFileName = "logmessages.txt";
    // incluir campos necessarios

    // singleton
    public static synchronized Logger getInstance(){

    private Logger(){

    public void putMessage(String message){

}
```

O que você deverá implementar nesta classe (fornecida juntamente com este roteiro):

(1) A classe deverá possuir um buffer compartilhado (sugestão: LinkedBlockingQueue).

(2) A classe deverá possuir uma thread para gravação em arquivo (consumidor).

(3) Para inserir uma mensagem no log, deve-se utilizar o método putMessage (produtor), lembrando de utilizar o serviço bloqueante da interface BloquingQueue (put(e)).

(4) Cada mensagem, ao ser retirada do buffer (método bloqueante take(e)) e gravada em arquivo, deverá conter o numero (contador) e a hora do evento. Alternativamente, a thread pode retirar todos elementos atuais do buffer (operação não bloqueante poll() da interface Queue) e dormir por 200 milissegundos, repetindo o processo ao acordar (gravação em lote).

(5) Inclua o que for necessário (métodos e atributos).

A classe que simula a geração de mensagens de log também é fornecida com este roteiro. Enviar a Implementação.