UFCG UASC/CEEI Disciplina: Programação Concorrente - Prova 1

Aluno: Matrícula:
Turma:

1 - Considere um sistema que atende dois tipos de clientes: internos e externos. Nesse sistema, chegam muito mais requisições de clientes externos do que internos. Para o projetista do sistema, é importante manter um balanço entre requisições de ambos os tipos, executando uma determinada função (**handle**), numa proporção 1:4. Ou seja, para cada quatro requisições externas (threads) executando a função **handle**, precisamos ter uma requisição (thread) interna executando a função **handle.** Assuma que threads são criadas (ou seja, não se preocupe com a criação das threads) para executar cada requisição que chega no sistema. Requisições externas executam a função **externalRoute** enquanto as internas executam **internalRoute.** Internamente, chamam **handle** (a região crítica). Modifique as funções **externalRoute** e **internalRoute** para garantir o comportamento esperado. A função handle não deve ser modificada. Você pode criar semáforos e qualquer outra variável como variáveis globais. Se deseja, inicie essas variáveis junto da declaração. Se precisar crie funções utilitárias que podem ser chamadas pelas funções **externalRoute** e **internalRoute.**

```
func void externalRoute(Request req) {
    handle()
}

func void internalRoute(Request req) {
    handle()
}
```

2 - Considere a implementação da estrutura de dados BufferedQueue. O objetivo desta estrutura é servir como um buffer de tamanho fixo (representado por capacity) onde threads produtoras enfileiram (chamando o método queue) itens a serem consumidos por threads consumidoras (chamando o método dequeue). Um objeto dessa estrutura de dados pode ser compartilhado entre múltiplas threads produtoras e consumidoras. Implemente o controle de concorrência (usando semáforos) para esse objeto de forma a garantir a proteção de possíveis regiões críticas, que uma thread consumidora aguarde a chegada de um novo item quando o buffer estiver vazio e que uma thread produtora aguarde o consumo de algum item caso o buffer esteja cheio. Segue implementação:

```
class BufferedQueue {
 int capacity;
 private final Queue<T> queue;

 //cria um BufferedQueue
new BufferedQueue(int capacity) {
 this.capacity = capacity;
 this.queue = new LinkedList<>();
}

//remove um item do buffer
T dequeue() {
 //This pool method returns the first element of this list, or null if this list is empty.
```

```
 T item = queue.poll();
 return item;
}

//adiciona um novo item no buffer
void enqueue(T item){
 //This add method appends the specified element to the end of this list.
 queue.add(item);
}
```

3 - Um cache de memória é um componente de hardware ou software que armazena dados temporariamente para acelerar leituras de dados que são acessados frequentemente. Em um sistema multi-threaded, múltiplas threads podem acessar o cache de memória simultaneamente. No entanto, a depender do tipo de operação a ser realizada pela thread no cache, alguns desafios de concorrência precisam ser considerados. Seguem os requisitos a serem atendidos:

- Leitura Simultânea: Várias threads podem ler do cache ao mesmo tempo sem problemas, desde que não modifiquem os dados;
- Escrita Exclusiva: Quando uma thread precisa atualizar ou escrever no cache, ele deve garantir que nenhuma outra operação (leitura ou escrita) esteja ocorrendo simultaneamente para evitar dados inconsistentes.
- Priorização da Escrita: Quando uma thread vai escrever no cache e já existe uma ou mais threads lendo o cache, a thread que vai realizar a escrita, além de executar de forma exclusiva, deve ser priorizada em relação à novas threads de leitura que cheguem depois. Ou seja, as threads de leitura que já tenham chegado podem terminar de executar mas novas threads de leitura que cheguem depois não devem prosseguir.

Considerando os requisitos acima, e que uma nova thread é criada no sistema sempre que necessário (ou seja, não se preocupe com a criação das threads), utilize semáforos e implemente as funções abaixo:

```
//impl. o controle de conc. para operações de leitura no cache.
data read()
//impl. o controle de conc. para operações de escrita no cache.
void write(data d)
//inicia variáveis globais e semáforos compartilhados entre as múltiplas threads.
main()
```

Na sua implementação, considere que fará um wrap para as funções **read** e **write.** Ou seja, implementará seguindo o padrão abaixo, inserindo as chamadas para os semáforos na função **wrap_read** e **wrap_write** (sem modificar as funções read e write).

```
func data wrap_read() {
            return read()
}
```

API Semáforos
Semaphore (int initialValue) wait() / signal()

p.s Das abstrações de concorrência vista em sala, você só pode usar semáforos. Qualquer abstração mais complexa (p.ex barreira etc) precisa ser implementada para ser usada.