UFCG UASC/CEEI

Disciplina: Programação Concorrente
Professor: Thiago Emmanuel Pereira

Prova 1

Q1 - Considere a API abaixo. A função **gateway** deve criar **nthreads** threads diferentes. O código executado por cada thread deve ser o da função request**.** A função request deve sortear um número aleatório **n** e dormir **n** segundos. Após criar as threads, a função **gateway** deve esperar que até **wait_nthreads** terminem. Após a espera, a função gateway deve retornar a soma dos valores **n** sorteados nas funções request.

```
int gateway(int nthreads, int wait_nthreads)
void request()

//ao chamar a função, uma thread bloqueará por nseconds segundos
void sleep(int nseconds)

//a função random retorna um número aleatório positivo
int random()

//cria uma thread que executará a função f quando for escalonada. retorna o id da
//thread. Assuma que no máximo podem existir 1024 threads (ids entre 0 e 1023)
int create_thread(func() f)
```

Q2 - Alguns sistemas críticos controlam tanto o número máximo de requisições que podem receber, quanto a quantidade das requisições que, uma vez recebidas, podem ser atendidas. Considere que um sistema desse tipo, ao receber uma requisição, cria uma uma nova thread que executa a função run. Além disso, o sistema tem uma única thread para atender as requisições, que executa a função handle em loop infinito.

```
void run();
void poison();

void encrypt()

void handle() {
      for(;;) {
            generateKey();
      }
}
```

As threads que executam run precisam verificar se o sistema já está atendendo o número máximo de **K** outras threads. Caso já esteja, a thread deve chamar a função **poison** que fará com que a thread termine (e, portanto, o número máximo de requisições no sistema não passe de **K**), ou seja, a chamada para **poison** não retorna. Caso não exista nenhuma thread de requisição, a thread do sistema (que atende as requisições) deve bloquear. Caso a thread do sistema esteja bloqueada, uma nova

requisição que tenha chegado para executar **run** deve desbloquear a thread do sistema. A thread do sistema executa **generateKey** para atender uma requisição. Por sua vez, a thread de requisição executa **encrypt**. A execução de **generateKey** deve ser concorrente com somente uma execução de **encrypt**.

Notem que a função **encrypt** executa o trabalho útil da thread de requisição. Ou seja, após terminar a execução de **encrypt**, a thread pode encerrar a execução do método **run** (obviamente, não é obrigatório que **encrypt** seja a última linha do código) normalmente (ou seja, não é necessário executar poison para esse caso).

Considerando as restrições acima, escreva/modifique as funções **run** e **handle** acima para coordenar a execução das threads.

Crie uma função main para inicializar os semáforos criados.

Q3 - Implemente um HashMap que seja seguro para uso concorrente. Assuma que você só precisa implementar três funções desse Map:

1) o construtor, **new HashMap()**;
2) **put(int key, int value);** e
3) **boolean containsKey(int key)**

Assuma que internamente, o Map usará uma LinkedList. Você também precisa implementar a List e ela precisa também ser segura para uso concorrente. Assuma que você só precisa implementar três funções da List (mas, assuma também que deve existir um método remove(int value) que você não precisa implementar):

1) o construtor, **new LinkedList()**;
2) **add(int value);** e
3) **boolean contain(int value)**

Lembre que internamente, um HashMap é organizado em buckets. Um bucket reúne todos os **values** os quais as **keys** relacionadas tiveram o mesmo hashcode. Em sua implementação, cada bucket deve usar uma LinkedList. Então, todos os **values** de um mesmo bucket estão na mesma LinkedList. Assuma também que a quantidade de buckets é definida a priori (e, é igual a 1024). É importante que seu código seja eficiente, então cuidado para proteger somente a região crítica (evite proteger código além da região crítica).