LINGUAGEM DE PROGRAMAÇÃO II

Centro de Informática – Universidade Federal da Paraíba – Campus I

Professor: Carlos Eduardo Batista - Semestre: 2018.2

TRABALHO PRÁTICO: *LOCKS* E BARREIRAS

Entrega: 15/04/2019 – 23h59

Envio por e-mail (bidu @ ci.ufpb.br) - arquivo de entrega deve anexar *todos* os códigos fonte em C/C++ dentro de um diretório nomeado "SUA_MATRICULA-LP1-ProvaPratica1" o qual deve ser comprimido em um arquivo ZIP ("SUA_MATRICULA-LP2-ProvaPratica1.zip"). O arquivo ZIP deve conter obrigatoriamente um arquivo de texto chamado README.txt contemplando **todas** as instruções de compilação e execução, além de qualquer observação que se fizer necessária para correção.

Leitura e escrita de arquivos com concorrência

O trabalho consiste na implementação de um programa em C/C++ que gere, a partir de um conjunto de N arquivos de entrada (requisições), N arquivos de saída (respostas), utilizando programação concorrente e a lógica detalhada a seguir.

O programa deverá conter em seu diretório raiz: um diretório chamado ***files***, o qual conterá arquivos sortidos (imagens JPEG e arquivos TXT, por exemplo); um diretório chamado ***reqs***, o qual conterá os arquivos de texto contendo as requisições; um diretório chamado ***answ***, que deverá armazenar os arquivos de texto contendo as respostas para as requisições.

Ao ser iniciado, o programa **deve buscar na pasta *reqs* os arquivos de requisição** (texto com a extensão *.req*), requisições estas que tratam da solicitação pelo conteúdo dos arquivos na pasta *files*. **Cada arquivo de requisição (.req) deverá ser tratado (lido e processado) em *threads* individuais**. O conteúdo dos arquivos de requisição deve seguir o formato descrito abaixo:

```
GET <nome-do-arquivo.extensão>
```

Para cada requisição, um arquivo de resposta (extensão .ans) deverá ser criado na pasta *answ*. A resposta da requisição deverá ser o conteúdo do arquivo solicitado ou, em caso de requisição malformada ou por arquivo inexistente, uma mensagem de erro adequada. Após tratar todas as requisições (e gerar todos os arquivos de resposta adequados) o programa **deverá exibir uma mensagem de sucesso informando a quantidade de bytes processada por todas as threads** (quantidade de bytes calculada a partir somatório do tamanho dos arquivos de resposta gerado por cada *thread*) e finalizar sua execução.

Para formatos de entrada e saída, considere o exemplo de execução descrito a seguir.

Em uma implementação correta, o programa é executado considerando o conteúdo das pastas *files* e *reqs* descrito abaixo (pasta *answ* deve iniciar vazia):

```
files/
        example.txt
        photo.jpg
```

```
reqs/
        req1.req
        req2.req
        req3.req
        req4.req
```

O conteúdo dos arquivos de requisições é:

Req1	Req2	Req3	Req4
GET photo.jpg	GET example.txt	GET index.html	GHEET sssss.sss

Portanto, após a execução do programa temos a pasta *answ* contendo os seguintes arquivos:

```
answ/
        ans-req1.ans
        ans-req2.ans
        ans-req3.ans
        ans-req4.ans
```

Temos então o conteúdo dos arquivos de resposta:

Ans-Req1	Ans-Req2	Ans-Req3	Ans-Req4
Conteúdo de photo.jpg	Conteúdo de example.txt	ERROR FILE index.html NOT FOUND	ERROR BAD COMMAND

A execução no terminal exibiria:

```
$ ./programa-lp2

Processando 4 requisições

[Thread 1] Processando req1.req

...

[Thread 4] Processando req4.req

Respostas criadas com sucesso. Total gerado em bytes: 329408234 bytes

$
```

Locks devem ser utilizados para a sincronização do acesso às seções críticas, barreiras devem sincronizar as múltiplas threads para o término do programa. A implementação deverá ser feita em C/C++ utilizando a biblioteca Pthreads ou as threads STD (C++11), utilizando as funções que estas oferecem para o tratamento de *locks* e barreiras (não é necessária a re-implementação). A implementação deverá atender às quatro propriedades de uma solução para o problema da seção crítica: exclusão mútua, ausência de *deadlock*, ausência de atraso desnecessário e entrada eventual. A saída do seu programa deve ser bem planejada e o código **comentando** (principalmente quando do uso de *locks* e barreiras), de forma a mostrar o que está acontecendo a cada momento.