



1º. Exame Escolar

Disciplina: Infra-estrutura de Software

Este exercício deve ser feito de forma individual ou em dupla, devendo ser apresentado e entregue na aula do dia **09/06/2016** no laboratório **G1**.

O projeto valerá de 0 a 10, e será levada em consideração a organização dos códigos de cada solução. Escreva códigos legíveis e com comentários sobre cada decisão importante feita nos algoritmos. As questões deverão ser implementadas em *threads* e utilizando o sistema operacional Linux. Ademais, caso uma questão necessite de arquivos de entrada, os mesmos deverão ser disponibilizados. O não cumprimento das regras acarretará em perda de pontos na nota final.

Tarefas:

1. [3,0 pontos] Uma distribuidora de artigos esportivos precisa de um programa que faça um levantamento do número de flechas em bom estado no estoque, e o número de flechas danificadas (que precisam ser consertadas e possuem um custo em relação aos reparos). O seu programa deve receber um número N de arquivos, um número $T \leq N$ de threads, um número F de tipos de flechas e um vetor P de tamanho F , onde $P[i] = \text{custo do conserto de uma unidade do } i\text{-ésimo tipo de flecha}$. Para o propósito do programa, o vetor P pode ser inicializado estaticamente. Ademais, seu programa deverá ter o vetor `custo_ruim` de dimensão F , o qual representa a **custo de total de consertar as unidades i lidas em cada arquivo**, ex. `custo_ruim[i] = 5 * P[i]`. Uma variável `contador_bom` deve ser utilizada para contar a quantidade flechas (independente do tipo) em bom estado.

Cada arquivo terá 2 números por linha, $\langle f, q \rangle$ tal que f é o tipo da flecha (entre 0 e $F-1$) e q é a quantidade desse tipo de flecha. Caso $q < 0$, você precisará calcular o preço para consertar as $|q|$ flechas, caso contrário, apenas incremente a quantidade de flechas em bom estado. Cada thread deverá pegar um arquivo. Quando uma thread concluir a leitura de um arquivo, e houver um arquivo ainda não lido, a thread deverá ler algum arquivo pendente. No final, imprima o total de flechas boas e para cada tipo de flecha, imprima o preço do conserto delas.

Assumindo o conhecimento prévio da quantidade de threads e arquivos, pode-se definir no início do programa quais arquivos a serem tratados por cada thread. Uma outra alternativa é ler os arquivos sob demanda, a partir do momento que uma thread termina a leitura de um arquivo, pega qualquer outro não lido dinamicamente.

ATENÇÃO: deve-se garantir a exclusão mútua ao alterar o vetor que guardará o custo do conserto de cada flecha. Porém, você deverá assumir uma implementação refinada. Uma implementação refinada garante a exclusão mútua separada para cada posição do vetor. Mais especificamente, enquanto um preço está sendo contabilizado para um tipo de flecha x e modificando o vetor na respectiva posição, uma outra thread pode

modificar o vetor em uma posição y que representa outra flecha. Ou seja, se o vetor de flechas possui tamanho 10, haverá um outro vetor de 10 mutex, um para cada posição do vetor de flechas. Ao ler um arquivo e detectar um conserto para a flecha y , a thread trava o mutex relativo à posição y , calcula o preço, e destrava o mutex na posição y . Obviamente, se mais de uma thread quiser modificar a mesma posição do vetor de flechas simultaneamente, somente 1 terá acesso, e as outras estarão bloqueadas. O mutex garantirá a exclusão mútua na posição.

A variável contador_bom é uma região crítica, e, assim, deve-se garantir também uma exclusão mútua específica para alterar esta variável.

Exemplo:

T = 1 // nº de threads
 N = 1 // nº de arquivos
 F = 3 // nº de flechas
 P[3] = {2, 3, 7} // preço de cada flecha

Arquivo de entrada [1.in](#):

```
2 10
1 -2
0 3
2 3
```

Saída:

16 flechas em bom estado!

Custo de consertar as flechas de tipo 0: R\$ 0,00

Custo de consertar as flechas de tipo 1: R\$ 6,00

Custo de consertar as flechas de tipo 2: R\$ 0,00

2. [3,0 pontos] O método de Jacobi é uma técnica representativa para solucionar sistemas de equações lineares (SEL). Um sistema de equações lineares possui o seguinte formato: $\mathbf{Ax} = \mathbf{b}$, no qual

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Ex:

$$A = \begin{bmatrix} 2 & 1 \\ 5 & 7 \end{bmatrix}, \quad b = \begin{bmatrix} 11 \\ 13 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$2x_1 + x_2 = 11$$

$$5x_1 + 7x_2 = 13$$

O método de Jacobi assume uma solução inicial para as incógnitas (x_i) e o resultado é refinado durante P iterações, usando o algoritmo abaixo:

```
while(k < P)
begin
```

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \dots, n.$$

```
    k = k + 1;
end
```

Por exemplo, assumindo o SEL apresentado anteriormente, $P=10$, $n=2$, e $x_1^{(0)}=1$ e $x_2^{(0)}=1$:

```
while(k < 10)
begin
    x1(k+1) = 1/2 * (11 - x2(k))
    x2(k+1) = 1/7 * (13 - 5x1(k))
    k = k+1;
end
```

Exemplo de execução

k=0

$$\begin{aligned} x_1^{(1)} &= 1/2 * (11 - x_2^{(0)}) = 1/2 * (11-1) = 5 \\ x_2^{(1)} &= 1/7 * (13 - 5x_1^{(0)}) = 1/7 * (13-5 * 1) = 1.1428 \end{aligned}$$

k=1

$$\begin{aligned} x_1^{(2)} &= 1/2 * (11 - 1.1428) \\ x_2^{(2)} &= 1/7 * (13 - 5 * 5) \end{aligned}$$

...

Nesta questão, o objetivo é quebrar a execução sequencial em threads, na qual o valor de cada incógnita x_i pode ser calculado de forma concorrente em relação às demais incógnitas (Ex: $x_1^{(k+1)}$ pode ser calculada ao mesmo tempo que $x_2^{(k+1)}$). A quantidade de threads a serem criadas vai depender de um parâmetro N passado pelo usuário durante a execução do programa, e N deverá ser equivalente à quantidade de processadores (ou núcleos) que a máquina possuir. No início do programa, as N threads deverão ser criadas, I incógnitas igualmente associadas para thread, e nenhuma *thread* poderá ser instanciada durante a execução do algoritmo. Dependendo do número N de threads, alguma thread poderá ficar com menos incógnitas associadas a ela.

Para facilitar a construção do programa e a entrada de dados, as matrizes não precisam ser lidas do teclado, ou seja, podem ser inicializadas diretamente dentro do programa (ex: inicialização estática de vetores). Ademais, **os valores iniciais de $x_i^{(0)}$ deverão ser iguais a 1**, e adote mecanismo (ex: *barriers*) para sincronizar as threads depois de cada iteração. Faça a experimentação executando o programa em uma máquina com 4 processadores/núcleos, demonstrando a melhoria da execução do programa com 1, 2 e 4 threads.

ATENÇÃO: apesar de $x_1^{(k+1)}$ poder ser calculada ao mesmo tempo que $x_2^{(k+1)}$, $x_i^{(k+2)}$ só poderão ser calculadas quando todas as incógnitas $x_i^{(k+1)}$ forem calculadas. Barriers são uma excelente ferramenta para essa questão.

3. [3,0 pontos] Em Java existem implementações de coleções (ex.: LinkedList, Set) que não apenas são seguras para o uso concorrente, como são especialmente projetadas para suportar tal uso. Uma fila bloqueante (**Blocking Queue**) é uma fila limitada de estrutura FIFO (First-In-First-Out) que bloqueia uma *thread* ao tentar adicionar um elemento em uma fila cheia ou retirar de uma fila vazia. Utilizando as estruturas de dados definidas abaixo e a biblioteca *PThreads*, crie um programa em C do tipo produtor/consumidor implementando uma fila bloqueante de inteiros (int) com procedimentos semelhantes aos da fila bloqueantes em Java.

```
typedef struct elem{
    int value;
    struct elem *prox;
}Elem;

typedef struct blockingQueue{
    unsigned bufferSize, bufferStatus;
    Elem *head, *last;
}BlockingQueue;
```

Na estrutura BlockingQueue acima, “head” aponta para o primeiro elemento da fila e “last” para o último. “bufferSize” armazena o tamanho máximo que a fila pode ter e “bufferStatus” armazena o tamanho atual (Número de elementos) da fila. Já na estrutura Elem, “value” armazena o valor de um elemento de um nó e “prox” aponta para o próximo nó (representando uma fila encadeada simples). Implemente ainda as funções que gerem as filas bloqueantes:

```
BlockingQueue* newBlockingQueue(unsigned inBufferSize);
void putBlockingQueue(BlockingQueue* Q, int newValue);
int takeBlockingQueue(BlockingQueue* Q);
```

- **newBlockingQueue:** cria uma nova fila Bloqueante do tamanho do valor passado.
- **putBlockingQueue:** insere um elemento no final da fila bloqueante Q, bloqueando a *thread* que está inserindo, caso a fila esteja cheia.
- **takeBlockingQueue:** retira o primeiro elemento da fila bloqueante Q, bloqueando a *thread* que está retirando, caso a fila esteja vazia.

Assim como em uma questão do tipo produtor/consumidor, haverá *threads* consumidoras e threads produtoras. As P *threads* produtoras e as C *threads* consumidoras deverão rodar em loop infinito, sem que haja *deadlock*.. Como as *threads* estarão produzindo e consumindo de uma fila bloqueante, sempre que uma thread produtora tentar produzir e a fila estiver cheia, ela deverá imprimir uma mensagem na tela informando que a fila está cheia e dormir até que alguma thread consumidora tenha consumido e, portanto, liberado espaço na fila. O mesmo vale para as *threads* consumidoras se a fila estiver vazia, elas deverão imprimir uma mensagem na tela informando que a fila está vazia e dormir até que alguma *thread* produtora tenha produzido.

Utilize variáveis condicionais para fazer a *thread* dormir (suspender sua execução temporariamente). Os valores de P, C e B (número de *Blocking Queues*) poderão ser inicializados estaticamente.

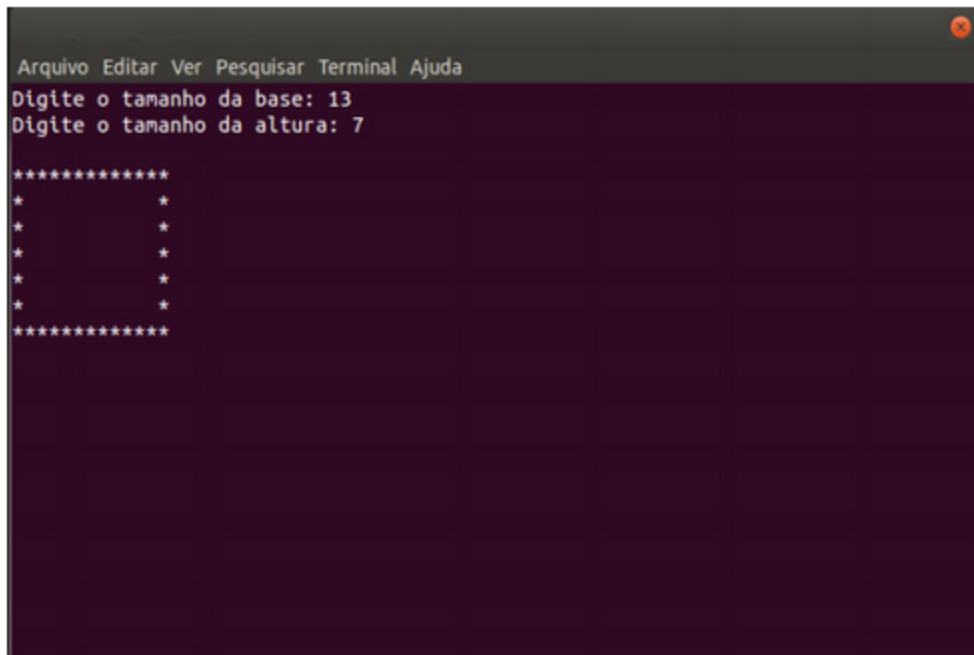
Dica: *Espera ocupada/ociosa é proibida. Ademais, você deverá garantir a exclusão mútua e a comunicação entre threads.*

Tarefa adicional: Assembly – use o *assembler* Nasm (www.nasm.us)

4. **[1,0 ponto]** Retângulo de Estrelas (modo de operação **protegido** – 32 ou 64 bits)

Crie um programa que receba como entrada dois números W e H (pedindo do usuário cada número separadamente). O programa deve imprimir na tela um retângulo feito de asteriscos (*) de altura H e largura W.

Exemplo:



```
Arquivo Editar Ver Pesquisar Terminal Ajuda
Digite o tamanho da base: 13
Digite o tamanho da altura: 7

*****
*               *
*               *
*               *
*               *
*               *
*               *
*****
```

Boa Sorte!