Projeto de Threads 2015.1

O projeto valerá de 0 a 10, e será levado 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 pthreads e utilizando o sistema operacional Linux. Ademais, caso uma questão necessite de arquivos, a equipe deverá disponibilizar arquivos exemplos de entrada. O não cumprimento das regras acarretará em perdas de pontos na nota final.

1. Em computação, busca por força bruta ou busca exaustiva é uma técnica de solução de problemas bastante trivial, porém muito geral que consiste em enumerar todos os possíveis candidatos da solução e testar cada candidato para saber se ele satisfaz os critérios definidos. Implemente um programa que tente descobrir uma senha de exatamente 10 caracteres Todos os caracteres serão numéricos: '0' até '9'. Ele deverá verificar sistematicamente todas as possíveis senhas até que a correta seja encontrada. Para isso, utilize *T threads,* na qual cada deverá testar uma faixa de senhas diferentes, de forma que uma senha de teste só possa ser gerada/testada por uma única thread. O programa deverá ser encerrado quando uma thread descobrir a senha. Nesse momento, deverá ser impressa na tela, a senha e a identificação/número da thread que a encontrou. Todas as outras senhas testadas não deverão ser impressas na tela. A senha a ser quebrada será uma string inicializada estaticamente, assim como o número *T de threads*. Todas as senhas geradas deverão ser strings numéricas.

OBS.: *Selecione arbitrariamente algumas senhas, e para cada uma faça comparações entre os tempos de execução para T=1 e para T=4 ou T=8. Adicionalmente, todas threads deverão ser criadas no início do programa e, durante a execução, nenhuma outra thread poderá ser executada.*

2. Você foi convocado para paralelizar o trabalho de um grupo de matemáticos, no qual cada grupo resolverá problemas de máximo divisor comum (MDC). Sua tarefa é criar um programa usando *pthreads* que gerencie o trabalho da seguinte forma:

- Ler a quantidade de grupos
- Ler a quantidade de integrantes por grupo
- Ler pares de números para cada integrante de cada grupo,os quais terão seus MDC calculados
- Um “turno” é representado pelo cálculo do MDC dos números de um integrante de cada equipe em ordem (ex: turno 1: os primeiros matemáticos das equipes 1 e 2; turno 2: os segundos matemáticos das equipes 1 e 2; ...)
- Os grupos podem resolver seus problemas simultaneamente, porém uma equipe deve esperar que outros grupos terminem o seu “turno” de MDC, somar os valores de MDC de todos os grupos, e então o “turno” seguinte será calculado.A soma do MDC de cada turno deve ser exibido na tela, para, em seguida, o próximo turno de MDC iniciar.

- Cada grupo deverá ser representado por um *thread*

Entrada:
2 //Quantidade de Equipes
2 //Quantidade de Membros por Equipe
15 10 //números para o primeiro matemático da equipe 1
20 30 //números para o segundo matemático da equipe 1
30 45 //números para o primeiro matemático da equipe 2
60 24 //números para o segundo matemático da equipe 2

Saída:
20//Saída para o "primeiro turno" de MDC(**5** + **15**)
22//Saída para o "segundo turno" de MDC(**10** + **12**)

Entrada:
1 //Quantidade de Equipes
2 //Quantidade de Membros por Equipe
21 9 //números para o primeiro "matemático da equipe 1
130 200 //números para o segundo "matemático da equipe 1

Saída:
3//Saída para o "primeiro turno" de MDC
10//Saída para o "segundo turno" de MDC

Dica: Barreira é uma técnica viável para esse problema. *Adicionalmente, todas threads deverão ser criadas no início do programa (após a leitura dos parâmetros) e, durante a execução, nenhuma outra thread poderá ser executada.*

3. Em um sistema de controle de falhas, deseja-se fazer o levantamento da quantidade de defeitos de diferentes equipamentos adquiridos pelos clientes de uma empresa. O registro de cada produto vendido encontra-se em diferentes arquivos, os quais foram obtidos dos clientes. Faça um programa que receba um número ***N*** de arquivos, um número ***T <= N*** de *threads* utilizadas para fazer a contagem, e um número ***P*** de produtos. Em seguida, o programa deverá abrir os ***N*** arquivos nomeados ***"x.in"*** no qual ***1 <= x <= N***. Cada arquivo terá 1 produto por linha que será um número ***y | 0 <= y <= P*** em que 1 significa o produto 1 e assim sucessivamente. 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. Ao final imprima na tela o total de produtos lidos, e o percentual de cada tipo de produto vendido.

Assumindo o conhecimento prévido da quantidade de threads e arquivos, pode-se definir no início do programa quais arquivos a serem tratados por cada thread. Uma outra alternativa ler

os arquivos sob demanda. a partir do momento que uma thread termina a leitura de um arquivo, pega qualquer outro não lido dinamicamente.

Ademais, deve-se garantir a exclusão múltua ao alterar o array que guardará a quantidade de cada produto. ***Porém, você deverá assumir uma implementação refinada.*** *Uma implementação refinada garante a exclusão mútua separada para cada posição do array. Mais especificamente, enquanto um produto está sendo contabilizado para um candidato* ***x*** *e modificando o array na respectiva posição, uma outra thread pode modificar o array em uma posição* ***y*** *que representação outro produto. Ou seja, se o array de produtos possui tamanho 10, haverá um outro array de 10 mutex, um para cada posição do vetor de produtos. Ao ler um arquivo e detectar um venda para o produto y, a thread trava o mutex relativo à posição y, incrementa a quantidade desse tipo de produto, e destrava o mutex na posição y. Obviamente, se mais de uma thread quiser modificar a mesma posição do array de produtos simultaneamente, somente 1 terá acesso, e as outras estarão bloqueadas. O mutex garantirá a exclusão mútua na posição.*

4. Em Java existem implementações de coleções (ex.: LinkedList, Set) que não apenas são seguras para o uso concorrente, mas 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 sizeBuffer, statusBuffer;
    Elem *head, *last;
}BlockingQueue;
```

Na estrutura BlockingQueue acima, “head” aponta para o primeiro elemento da fila e “last” para o último. “sizeBuffer” armazena o tamanho máximo que a fila pode ter e “statusBuffer” 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 inSizeBuffer);
```

```
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). O valores de P, C e B poderão ser inicializados estaticamente.

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