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 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. Você deverá criar um programa usando pthreads, no qual threads deverão decrementar um contador global de 5.000.000 até 0. A thread que alcançar este valor deverá imprimir que o valor foi alcançado e todas as threads deverão finalizar a execução.

2. 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 em *pthreads* 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 uma 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.*

3.Em uma escola deseja-se analisar os desempenhos dos alunos de várias salas no último teste, e a nota dos alunos de uma sala encontra-se em um arquivo cujo o nome possui o seguinte formato: sala_.txt onde no lugar de "_" encontra-se o número da sala, que pode ser um número natural de 1 a n (onde n é a quantidade de arquivos a serem lidos.). Os dados nos arquivos estão no seguinte formato:

Aluno1 10

Aluno2 9.5

Aluno3 3.5

…

Para analisar o desempenho de uma sala, você deverá calcular as seguintes estatísticas: Média, Moda e Mediana, e desvio padrão. Os dados serão guardados em um ***array***, no qual cada posição representa os dados de uma sala. Ao final do cálculo de todas as salas, o programa deverá exibir os resultados obtidos

Sala X

Media: “valor da média”

Moda: “valor da moda”

Mediana: “valor da mediana”

Desvio Padrão: “valor do desvio padrão”

Os arquivos de entrada sala_.txt devem ser acessados de forma simultânea, e cada arquivo será acessado por uma única thread. Em outras palavras, duas threads não podem acessar o mesmo arquivo. O número de threads e de arquivos será informados inicialmente pelo usuário (o número de threads e o de arquivos podem ser diferentes).

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 as estatísticas de cada sala. ***Porém, você deverá assumir uma implementação refinada.*** *Uma implementação refinada garante o acesso separado para cada posição do array. Mais especificamente, enquanto uma sala* ***x*** *está sendo contabilizada e modificando o array na respectiva posição, uma outra thread pode modificar o array em uma posição* ***x*** *que representa outra sala.*

4. Um sistema gerenciamento de banco de dados (SGBD) comumente precisa lidar com várias operações de leituras e escritas concorrentes. Neste contexto, podemos classificar as threads como leitoras e escritoras. Assuma que enquanto o banco de dados está sendo atualizado devido a uma operação de escrita (uma escritora), as threads leitoras precisam ser proibidas em realizar leitura no banco de dados. Isso é necessário para evitar que uma leitora interrompa uma modificação em progresso ou leia um dado inconsistente ou inválido.

Você deverá implementar um programa usando pthreads, considerando **N** threads leitoras e **M** threads escritoras. A base de dados compartilhada (região crítica) deverá ser um array, e threads escritoras deverão continuamente (em um laço infinito) escrever no array em qualquer posição. Similarmente, as threads leitoras deverão ler dados (de forma contínua) de qualquer posição do array. As seguintes restrições deverão ser implementadas:

1. As threads leitoras podem simultaneamente acessar a região crítica (array). Ou seja, uma thread leitora não bloqueia outra thread leitora;
2. Threads escritoras precisam ter acesso exclusivo à região crítica. Ou seja, a manipulação deve ser feita usando exclusão mútua. Ao entrar na região crítica, uma thread escritora deverá bloquear todas as outras threads escritoras e threads leitoras que desejarem acessar o recurso compartilhado.

Dica: *Você deverá usar mutex e variáveis de condição.*

5. Para facilitar e gerenciar os recursos de um sistema computacional com múltiplos processadores (ou núcleos), você deverá desenvolver uma **API** para tratar requisições de chamadas de funções em threads diferentes. A **API** deverá possuir:

- Uma constante ***N*** que representa a quantidade de processadores ou núcleos do sistema computacional. Consequentemente, ***N*** representará a quantidade máximas de threads em execução;

- Um ***buffer*** que representará uma fila das execuções pendentes de funções;;

- Função **agendarExecucao.** Terá como parâmetros a função a ser executada e os parâmetros desta função em uma *struct*. Para facilitar a explicação, a função a ser executada será chamada de ***funexec***. Assuma que ***funexec*** possui o mesmo formato daquelas para criação de uma thread: um único parâmetro. Isso facilitará a implementação, e o struct deverá ser passado como argumento para ***funexec*** durante a criação da *thread*. A função **agendarExecucao** é não bloqueante, no sentido que o usuário ao chamar esta funcionalidade, a requisição será colocada no ***buffer***, e um ***id*** será passado para o usuário. O ***id*** será utilizado para pegar o resultado após a execução de ***funexec*** e pode ser um número sequencial;

- Thread **despachante.** Esta deverá pegar as requisições do ***buffer***, e gerenciar a execução de ***N*** *threads* responsáveis em executar as funções ***funexecs*****.** Se não tiver requisição no buffer, a *thread* **despachante** dorme. Pelo menos um item no ***buffer,*** faz com que o despachante acorde e coloque a ***funexec*** pra executar. Se por um acaso ***N*** *threads* estejam executando e existem requisições no buffer, somente quando uma thread concluir a execução, uma nova ***funexec*** será executada em uma nova thread. Quando ***funexec*** concluir a execução, seu resultado deverá ser salvo em uma área temporária de armazenamento (ex: um buffer de resultados). O resultado de uma ***funexec*** deverá estar associada ao ***id*** retornado pela função **agendarExecucao. Atenção: esta *thread* é interna da API e escondida do usuário.**

- Função **pegarResultadoExecucao.** Terá como parâmetro o ***id*** retornado pela função **agendarExecucao.** Caso a execução de ***funexec*** não tenha sido concluída ou executada, o usuário ficará bloqueado até que a execução seja concluída. Caso a

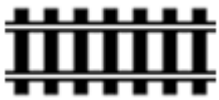
execução já tenha terminado, será retornado o resultado da função. Dependendo da velocidade da execução, em muitos casos, os resultados já estarão na área temporária.

A implementação não poderá ter espera ocupada, e os valores a serem retornados pelas funções ***funexec*** podem ser todas do mesmo tipo (ex: números inteiros ou algum outro tipo simples ou composto definido pela equipe). ***funexec*** é um nome utilizado para facilitar a explicação, e diferentes nomes poderão ser utilizados para definir as funções que serão executadas de forma concorrente.
Você deverá utilizar variáveis de condição para evitar a espera ocupada. Lembre-se que essas variáveis precisam ser utilizadas em conjunto com mutexes. Mutexes deverão ser utilizados de forma refinada, no sentido que um recurso não deverá travar outro recurso independente.

6. Você deverá implementar um sistema computacional de controle de ferrovias usando C e *pThreads*. A ferrovia é composta por 5 interseções, as quais só permitem **2 trens simultaneamente**. Todavia, um trem passando em uma interseção não deve afetar (bloquear) o andamento dos outros trens em outras interseções. Uma interseção é representada por uma variável inteira. Um trem passando pela interseção (com menos de 2 trens) deverá modificar a variável contador da interseção indicando a quantidade de trens nesta, e esperar 500 milissegundos para indicar o término de sua passagem. Ao concluir a passagem na interseção, deverá decrementar o respectivo contador em uma unidade. O trem, que liberar uma interseção, somente deverá notificar algum trem aguardando a liberação desta interseção.

Por exemplo: Trem 1 e Trem 2 estão na interseção 5, e Trem 3 está aguardando a liberação desta interseção (depois de ter passado pela interseção 4). Trem 1, ao sair da interseção 5, disponibilizará um trilho na interseção, mas só deverá notificar sua saída para o Trem 3 (pois é o único trem aguardando a interseção 5). Os trens deverão ser implementados usando threads, as quais precisam acessar as interseções na sequência 1,2,3,4,5. Ao concluir o percurso, cada trem começará a trafegar novamente a partir do início (ou seja, a partir da interseção 1). Assuma a existência de 10 trens.

1 -2 -3 - 4 - 5



==========>

7. Um dos algoritmos mais conhecidos na computação é o Merge Sort, o qual é um algoritmo de ordenação por comparação do tipo dividir-para-conquistar.
https://pt.wikipedia.org/wiki/Merge_sort

```
sort(l, r){
        printf("Ordenando [%d, %d]\n", l, r);
        if(l == r) return;
        m = (l+r)/2
        sort(l, m)
        sort(m+1, r)
        merge(l, m, r)
        printf("Ordenado [%d, %d]\n", l, r);
}
```

Uma coisa a se notar é que enquanto o problema do lado esquerdo não for resolvido, o algoritmo **não** começará a resolver o problema do lado direito.

Como o problema da esquerda e o da direita operam em intervalos disjuntos, uma boa otimização seria resolvê-los em paralelo.

Implemente esta otimização usando pthreads, imprimido o vetor final ordenado.

8. Crie um programa em pthreads que encontre o N-ésimo número primo natural. Uma técnica simples para determinar se um certo número natural é primo é a “divisão por tentativa”. Basta dividí-lo por todos os números menores (Exceto pelo número 1) ou iguais à sua raiz quadrada. Se nenhuma dessas “tentativas” resultar em uma divisão exata (Divisão cujo resto é igual a 0) o número é considerado primo. Caso contrário ele é considerado composto. O número 1 não é considerado um número primo e nem composto.

O usuário deverá informar pelo teclado o número ***T*** de threads a serem criadas e o número ***N***. As ***T*** threads só poderão ser criadas no começo do programa e só poderão ser encerradas depois que o número primo ***N*** já tenha sido determinado. Cada thread deverá fazer várias divisões por tentativas em números diferentes. Assim que uma thread terminar de testar as divisões em um número, ela imediatamente deverá testar em outro.

O número procurado deverá ser impresso no console da tela.

Observação: Nessa questão, a sincronização entre as threads é algo fundamental. Um programa que não esteja perfeitamente sincronizado poderá determinar não o N-ésimo número primo, mas sim um número primo próximo a ele.

Dica: O único número primo par é o número 2 e todos os outros são múltiplos dele. Para efeitos práticos, então, evite utilizar esses números tanto para testá-los quanto para testar outros números.

Ex1.: Entrada: Numero T: 4
Numero N: 34156

Saída: N-esimo numero primo: 404021

Ex2.: **Entrada:** Numero T: 8
Numero N: 89336

Saída: N-esimo numero primo: 1150489

Será necessário exclusões mútuas para 2 regiões críticas. Uma região crítica indicará qual o próximo número natural a ser testado. Uma outra região crítica deverá armazenar a ordem do número primo testado (n-ésimo testado até o momento) e o número primo. A ordem basta incrementar (pois, basicamente, irá representar um simples contador). Todavia, deve-se ter cuidado ao alterar o número primo, pois um número maior já pode ter sido testado.