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.
Atenção: Usar somente pthreads e Linux.

1. Desenvolva um programa que conte a ocorrência de uma palavra específica em ***X*** arquivos de texto. O programa deve usar threads, onde cada thread é responsável por buscar a palavra em um arquivo diferente. Use mutexes para atualizar um contador global de forma segura. O programa deve imprimir o total de ocorrências da palavra em todos os arquivos.

2. Crie um programa em C que utilize o algoritmo de ordenação bubble sort de maneira concorrente. O programa deve dividir um array em N partes, e cada thread deve ordenar uma parte. Após todas as threads completarem a ordenação de suas respectivas partes, uma thread final deve mesclar todos os segmentos para formar o array ordenado completo. Utilize barriers para sincronizar a conclusão das ordenações parciais antes de começar a mesclagem.

3. Implemente um simulador de operações bancárias usando threads. O programa deve ter **N** threads representando clientes e uma thread representando o banco. Os clientes devem realizar operações como depósito, saque e consulta de saldo. O programa deve garantir que as operações sejam atômicas em uma mesma conta para evitar condições de corrida.

4. O sudoku é jogado em um tabuleiro de 9x9, dividido em nove subgrades de 3x3. O objetivo é preencher o tabuleiro de modo que cada linha, cada coluna e cada subgrade contenha todos os números de 1 a 9 sem repetições. Faça um programa usando pthreads que recebe uma matriz 9x9 e confere, usando múltiplas threads, se é uma solução válida de sudoku.

5. O método de Jacobi é uma técnica representativa para solucionar sistemas de equações lineares (SEL). Um sistema de equações lineares possui o seguinte formato : $A\mathbf{x} = \mathrm{b}$, no qual

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

Ex:
$2x_1 + x_2 = 11$
$5x_1 + 7x_2 = 13$

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

O método 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
```

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

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

Por exemplo, assumindo o SEL apresentado anteriormente, P=10, e $x1^{(0)}$=1 e $x2^{(0)}$=1:

while(k < 10)
begin
$x_1^{(k+1)}$ =1/2 * (11 – $x_2^{(k)}$)
$x_2^{(k+1)}$ = 1/7 * (13 – 5$x_1^{(k)}$)
k = k+1;
end

Exemplo de execução

k=0
$x_1^{(1)}$ =1/2 * (11 – $x_2^{(0)}$) = 1/2 *(11-1) = 5
$x_2^{(1)}$ = 1/7 * (13 – 5$x_1^{(0)}$) = 1/7 *(13-5 * 1) = 1.1428

k=1
$x_1^{(2)}$ =1/2 * (11 –1.1428)
$x_2^{(2)}$ = 1/7 * (13 – 5 * 5)

…

Nesta questão, o objetivo é quebrar a execução seqüencial 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 assoicadas à 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, demostrando a melhoria da execução do programa com 1, 2 e 4 threads.

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

6. Uma versão brasileira do Linux pretende implementar um novo algoritmo de escalonamento para *threads* em CPUs *multicore* (múltiplos núcleos). Para isso, deve seguir os requisitos abaixo:

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

- ***lista_pronto*** que representará uma fila das execuções pendentes das threads;

- Uma thread **escalonador.** Esta deverá pegar as *threads* da **lista_pronto**, e gerenciar a execução nos ***N*** *núcleos*. Assuma que o código das threads são representadas por uma função qualquer que termina (ex: não tem laço infinito). Se não houver *thread* para ser executada *na* **lista_pronto**, a *thread* **escalonador** dorme. Pelo menos uma thread na **lista_pronto*,*** faz com que o **escalonador** acorde e coloque a *n*ova **thread** pra executar. Se por um acaso ***N*** *threads* estejam executando e existem *threads* na **lista_pronto,**, somente quando uma thread concluir a execução, uma nova thread será executada.
- **Atenção: a *thread* escalonador é interna do sistema operacional e escondida do usuário.**

A implementação não poderá ter espera ocupada. A estrutura e funcionamento ficarão a critério da equipe, desde que siga os requisitos acima. Por exemplo, pode-se criar uma função **agendar** para adicionar uma *thread* a **lista_pronto** e acordar o escalonador quando necessário.
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.