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 *n* threads deverão incrementar um contador global até o número 1.000.000. A thread que alcançar este valor deverá imprimir que o valor foi alcançado e todas as threads deverão finalizar a execução.

2. Em algumas estações de trem, existem telas que informam os trens que estão chegando e partindo. Estas telas são atualizadas sempre que uma nova informação situação dos trens está disponível. Você deve implementar um programa usando *threads* que atualize esta tela de acordo com as informações lidas de arquivos. Uma tela com 7 linhas possui o seguinte formato:

```
AQT123 Istambul     17:45
XYZ001 Nice         17:50
ABC789 Moscou       18:20
FFF305 Estocolmo    18:35
QRS111 Madri        18:45
DEF321 Berlim       19:00
GHI456 St. Petsbu.  19:15
```

Cada linha deve ser atualizada de maneira independente. Assim, é preciso garantir a exclusão mútua **por linha.** Por exemplo, enquanto uma thread estiver atualizando um trem na linha 3 da tabela, outra thread pode atualizar a linha 5. Haverá ***N*** de arquivos e um número **1 < T <= *N*** de *threads* utilizadas para atualizar a tela. A tela possuirá **L linhas**. As informações contidas em um arquivo de entrada tem o seguinte formato:

```
3 // Número da linha da tabela a ser atualizada
ABC123 Paris 19:00 // Nova informação a ser escrita
7
QWE497 Lisboa 18:50
1
GHD056 Frankfurt 20:00
```

Logo, se as alterações acima fossem aplicadas na tela do começo da questão teríamos:

```
AQT123 Frankfurt    20:00
XYZ001 Nice         17:50
ABC789 Paris        19:00
FFF305 Estocolmo    18:35
QRS111 Madri        18:45
DEF321 Berlim       19:00
GHI456 Lisboa       18:50
```

Você deverá usar o padrão ANSI para deixar a tela colorida e a linha não mudar de posição na tela. Procure no google o uso de printf com ANSI em C (no sistema operacional Linux). Para que seja possível visualizar as alterações, cada linha deverá aguardar 2 segundos antes de ser modificada novamente.

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 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 mútua ao alterar uma linha da tela. ***Porém, você deverá assumir uma implementação refinada.*** *Uma implementação refinada garante a exclusão mútua separada para cada linha da tela. Mais especificamente, enquanto uma linha* ***x*** *está sendo alterada por uma thread, uma outra thread pode modificar a linha* ***y****. Ou seja, se a tela possui 7 linhas, haverá um array de 7 mutex, um para cada linha. Ao ler um arquivo e detectar uma alteração para linha y, a thread trava o mutex relativo à posição y, faz a modificação, e destrava o mutex na posição y. Obviamente, se mais de uma thread quiser modificar a mesma linha simultaneamente, somente 1 terá acesso, e as outras estarão bloqueadas. O mutex garantirá a exclusão mútua na linha.*

3. Considere 2 strings **s1** and **s2**. Crie um programa em C usando pthreads para encontrar a quantidade de substrings em **s1** que são iguais a **s2**. Por exemplo, suponha que quantidade_substring(s1, s2) implementa a funcionalidade, então quantidade_substring(“abcdab”, “ab”) = 2, quantidade_substring(“aaa”, “a”) = 3, quantidade_substring(“abac”, “bc”) = 0. Assuma o tamanho de **s1** e **s2** seja **n1** e **n2**, respectivamente, and **p** threads são adotadas. Assuma também que **n1 mod p = 0** (resto da divisão), e **n2 < n1/p** . Adicionalmente, considere que a string s1 é uniformemente particionada para que **p** threads pesquisem concorrentemente por substrings. Depois que uma thread termina a busca e obtenha um quantidade de substrings, este contador deverá ser incrementado a uma variável global que terá o total de todas substrings s2 em s1. Quando todas as threads terminarem, o resultado deverá ser exibido.

4. 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.

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 : $A\mathbf{x} = b$, 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. 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 (**BlockingQueue**) 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 int 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 int SizeBuffer);
void putBlockingQueue(BlockingQueue*Q,intnewValue);
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*** (tamanho do buffer) poderão ser inicializados estaticamente.

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

7. Você deverá implementar um programa que converte imagens colorida em tons de cinza utilizando pthreads (para acelerar a conversão). As imagens deverão adotar o modelo de cores RGB: Red, Green e Blue. Este é um modelo aditivo, no qual as cores primárias vermelho, verde e azul são combinadas para produzir uma cor. Desta forma, em uma imagem do tipo bitmap (matricial), cada pixel possui 3 valores. O formato textual PPM (Portable Pixel Map) do tipo P3 será adotado e, abaixo, segue um exemplo:

```
P3
3 2
255
```

```
255   0   0  # red
  0 255   0  # green
  0   0 255  # blue
255 255   0  # yellow
255 255 255  # white
  0   0   0  # black
```

Na 1a linha, o arquivo possui o número mágico identificando o formato. Em seguida, as dimensões são informadas e, na 3a linha, o valor máximo possível para cada cor no modelo é definido. Nas linhas seguintes, o arquivo define os valores das cores para cada pixel. Para simplificar, assuma que cada linha do arquivo (a partir da 4a linha) tenha os valores das cores para um pixel, obedecendo o preenchimento por linha da imagem. Considerando o exemplo acima, as 3 primeiras linhas do arquivo são da 1a linha da imagem, e as próximas 3 linhas do arquivo são da 2a linha da imagem. Ademais, considere que o arquivo não terá comentários (ex: # black)

Para fazer a conversão para tons de cinza (C), adote a seguinte fórmula: C = R*0.30 + G*0.59 + B*0.11. Exemplo: se um pixel possui o valor RGB <R=100, G=200, B=100>, o respectivo valor em tons de cinza é 159 = 100*0.30 + 200*0.59 + 100*0.11.

Seu programa deverá ler um arquivo PPM e, em seguida, a conversão é realizada usando múltiplas threads. No final, a imagem convertida é salva em um arquivo distinto. No arquivo final, coloque os valores R,G e B iguais ao tom de cinza calculado para cada pixel.

Dica: *A leitura e escrita de arquivo não deve usar múltiplas threads, mas somente a conversão da imagem que estará sendo representada por uma matriz. Cada pixel (elemento da matriz) pode ser convertido independente dos outros pixels da imagem, ou seja, isolando a conversão de um pixel, não há condição de disputa/corrida.*