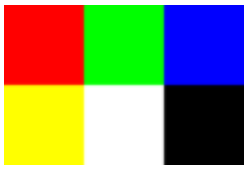
O projeto valerá de 0 a 10 e será levada em consideração a organização dos códigos de cada solução. Escreva códigos legíveis e com comentários sobre cada decisão importante feita nos algoritmos. As questões deverão ser implementadas em pthreads e utilizando o sistema operacional Linux. Ademais, caso 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 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. 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)

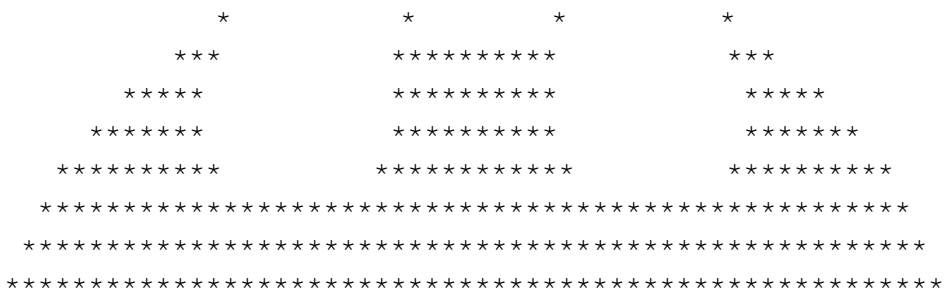
Para fazer a conversão para tons de cinza (C), adote a seguinte fórmula: C = R*0.30 + G*0.59 + B*0.11. Exemplo: se um pixel possui o valor RGB <R=100, G=200, B=100>, o respectivo valor em tons de cinza é 159 = 100*0.30 + 200*0.59 + 100*0.11.

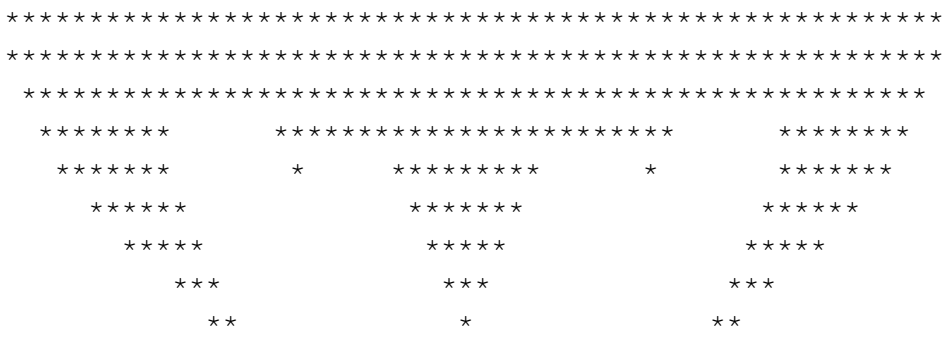
Seu programa deverá ler um arquivo PPM e, em seguida, a conversão é realizada usando múltiplas threads. No final, a imagem convertida é salva em um arquivo distinto. No arquivo final, coloque os valores R,G e B iguais ao tom de cinza calculado para cada pixel.

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.*

3. Você será o projetista responsável pela elaboração de um renderizador para um *game engine*. Um renderizador é basicamente um processo que gera um output visual a partir de um modelo. Assim, você deverá implementar em pthreads um programa que seguirá os seguintes passos:

-O programa deve ter um vetor bidimensional 150x40 do tipo básico char que será chamado de *display*. Nestes, as saídas visuais serão geradas;
-Os modelos serão diversos arquivos .txt que terão o seguinte formato:
*Coordenadas x y (coordenada do canto superior esquerdo do modelo, será representada por dois números inteiros);
*Dimensões do modelo x y (todos os modelos serão retângulos , e suas dimensões serão representadas por dois números inteiros);
*As próximas N linhas serão o modelo em si, que deve ser lido como um vetor bidimensional do tipo char.
Ex:
20 13
56 17





-Assuma **N** threads que acessarão **X** arquivos

Cada modelo é independente um do outro, entretanto apenas uma thread pode acessar o *display* por vez. Após todos um modelo ser carregado por completo, o vetor deve ser impresso no terminal.
Link para os arquivos (assets): assets

4. Sudoku é um jogo simples, no qual cada quadrado (3x3) deve ser preenchido com números de 1 a 9. Entretanto, o mesmo número não pode ser usado duas vezes. O somatório de todos elementos em uma linha ou coluna deve ser igual a 9. Você deverá criar uma aplicação usando pthreads a fim de verificar se um sudoku foi resolvido corretamente. Você deverá assumir um tabela com nove quadrados, similar ao exemplo da figura abaixo

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Para verificar a solução deseja-se que cada linha, coluna e quadrado sejam verificados simultaneamente através do uso de **N** threads. Caso alguma thread verifique que aquela solução não seja válida, deve-se enviar uma mensagem para todas as threads encerrarem a execução e deve-se imprimir “INVALIDA” na tela. Caso todas as threads encerrem a execução

sem invalidar a solução, isso significa que que aquele sudoku é válido e deve-se imprimir “VALIDA” na tela.

EXEMPLOS:

INPUT:

7 4 8 5 3 1 9 2 6

3 9 6 7 8 2 4 1 5

1 5 2 4 9 6 8 3 7

2 8 3 1 5 4 7 6 9

6 7 9 3 2 8 5 4 1

5 1 4 6 7 9 2 8 3

4 2 1 9 6 7 3 5 8

9 6 5 8 4 3 1 7 2

8 3 7 2 1 5 6 9 4

OUTPUT:

VALIDA

INPUT:

6 8 1 5 9 3 2 7 4

3 7 2 4 1 8 6 5 9

4 9 5 7 6 2 3 8 1

7 6 9 8 4 1 5 2 3

1 3 4 2 7 5 9 6 8

5 2 8 9 3 6 4 1 7

9 1 7 6 2 4 8 3 5

2 5 3 1 8 9 7 4 7

8 4 6 3 5 7 1 7 2

OUTPUT:

INVALIDA

5. 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.*

6. 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.

7. Utilize a API criada na questão x para implementar um sistema de máquinas de caixa eletrônico. Assuma **C** máquinas clientes que farão somente débitos em contas correntes. A forma de implementação será livre, desde que use a API implementada. Ademais, assuma que não haverá condição de disputa para uma mesma conta corrente

8. 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.*

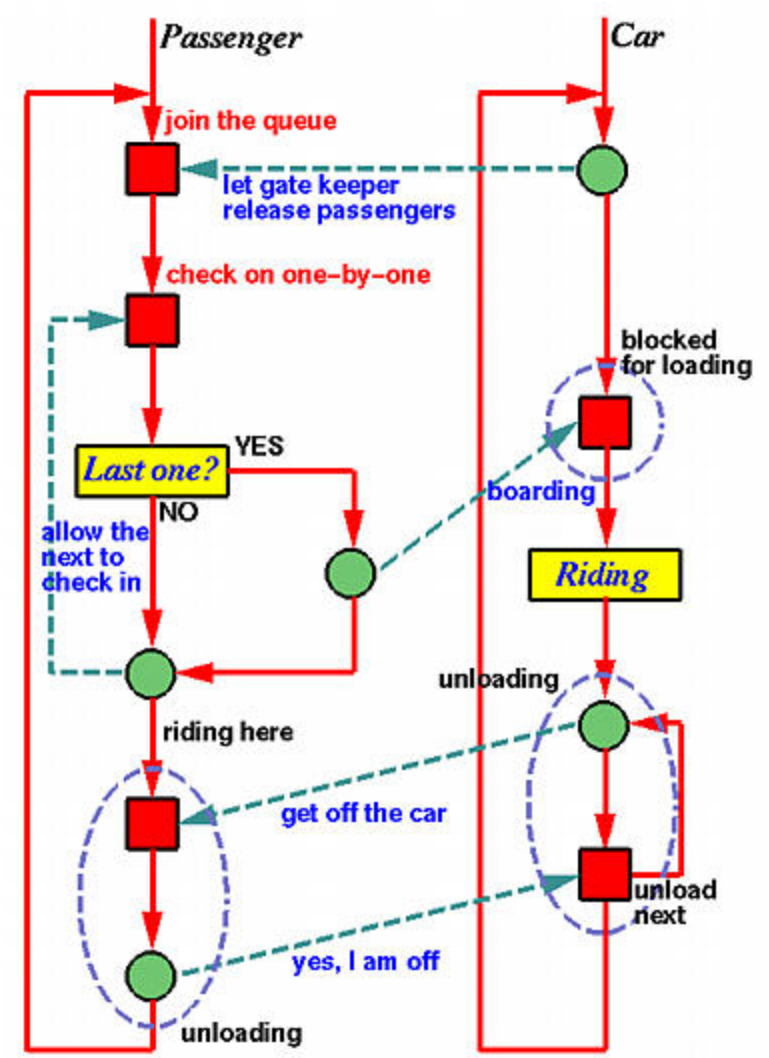
9 Um exemplo muito comum de controle de processo é o Problema da Montanha Russa. Nesse problema, existem 3 tipos de threads:

- a montanha russa;
- os passageiros;
- o(s) carrinho(s).

Para o nosso problema, vamos supor que temos **20** passageiros ao todo e **1** carrinho. Os passageiros precisam esperar na fila até que consigam o carrinho para andar na montanha russa. O carrinho suporta no máximo **10** passageiros por viagem, e ele só sai da estação quando estiver cheio. Depois do passeio, os passageiros, um pouco enjoados, passeiam pelo parque antes de retornarem à fila para andar novamente na montanha russa. Porém, por motivos de segurança, o carrinho só pode fazer **10** voltas por dia.

Supondo que o carro e cada passageiro sejam representados por uma thread diferente, escreva um programa, usando pthreads, que simule o sistema descrito. Após as 10 voltas, o programa deve ser encerrado

Importante: Nenhum passageiro pula pra fora/pra dentro do carrinho em movimento e não é possível pular a vez na fila de espera.


Passenger
Car
join the queue
let gate keeper
release passengers
check on one-by-one
blocked
for loading
Last one?
YES
NO
boarding
allow the
next to
check in
Riding
unloading
riding here
get off the car
unload
next
yes, I am off
unloading