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 de 0 até 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. Considerando um *array* como entrada e um número *N*, verifique se ele está ordenado (não decrescente), usando N *threads*.

Sugestão: *particionar o array em N partes, e cada thread verifica se aquele trecho está ordenado. Por fim, cada thread checa se o primeiro e último elemento do seu está ordenado com seu próximo*.

3. Usando *pthreads*, implemente um programa para encontrar uma saída de um labirinto,

Um labirinto é representado por uma matriz de 0s e 1s, um ponto de entrada e um de saída. Seu programa deverá achar um caminho (se houver) da entrada até a saída que não passe pelas paredes. Em outras palavras, não pode ter posições com 1 no caminho, apenas 0..

(animações para uma thread: find a path out, find shortest path out)

As threads devem realizar a busca evitando passar por posições que já foram visitadas (por qualquer thread).

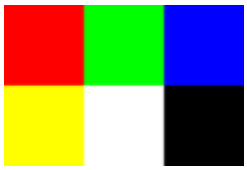
Dicas para Solução:

- Use DFS/BFS considerando cada casa do labirinto (posição da matriz) como um node de um grafo e duas casas adjacentes como dois nodes ligados por uma aresta.
- Use uma matriz de mutex para as threads não procurarem ao mesmo tempo na mesma casa/posição.

4. 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.*

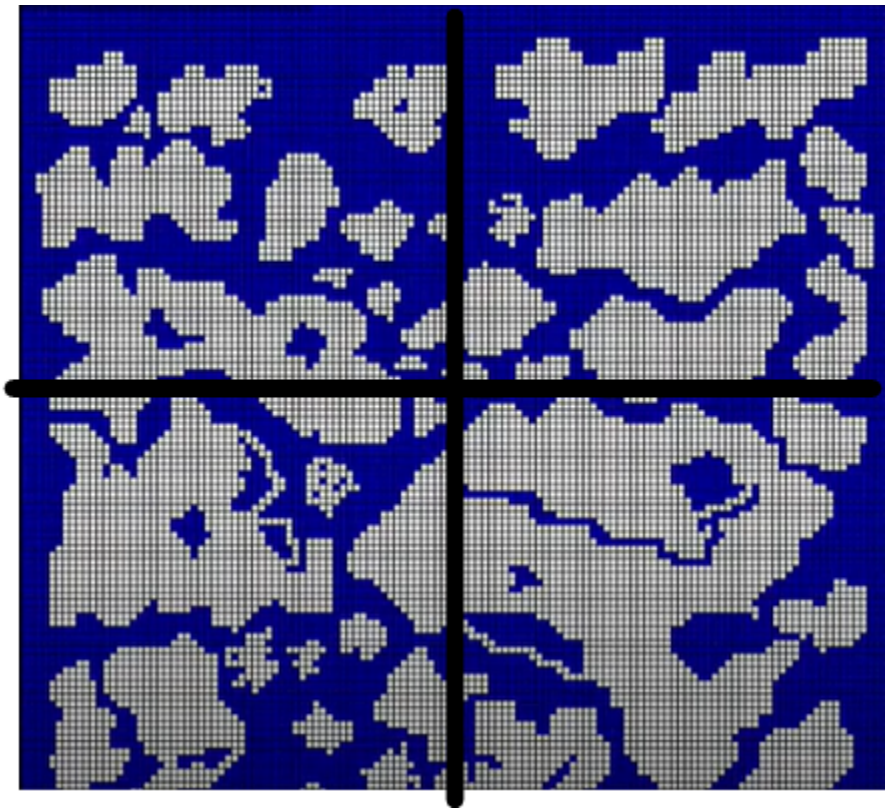
5 - A marinha recebeu um mapa em formato de matriz com 0 e 1, de tal forma que:
0: água
1: terra

Sua equipe deverá implementar um programa em *pthreads* para identificar a quantidade de ilhas no mapa. De forma geral, se duas posições de terra são adjacentes na vertical, na horizontal, ou na diagonal, elas são da mesma ilha.

Uma forma de implementação é utilizando a estrutura *Disjoint Sets* (veja capítulo 21 do livro Algoritmos: Teoria e Prática de Cormen et al). Percorrendo a matriz, toda vez que encontrar uma entrada de terra (1), faça *Union* da posição com todas as posições de terra adjacentes. Ao final da execução do algoritmo, conta-se a quantidade de conjuntos disjuntos. Este é o número de ilhas.

O algoritmo deve receber uma matriz como entrada e um inteiro **N** que representa o número de *threads* a serem criadas e utilizadas para resolver o problema..

Dica: (I) divida as threads por regiões disjuntas do mapa, de tal forma que uma *thread* nunca olhe a região da outra. No final, faça o *Union das* interseções dos territórios das threads, ou seja, analisando as fronteiras (linha preta na figura) e a existência de território.. Outra opção é não se importar com a possibilidade das *threads* acessarem uma mesma região. Entretanto, o programa deverá controlar a condição de disputa na criação dos conjuntos disjuntos.



6. 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.*

7. 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.

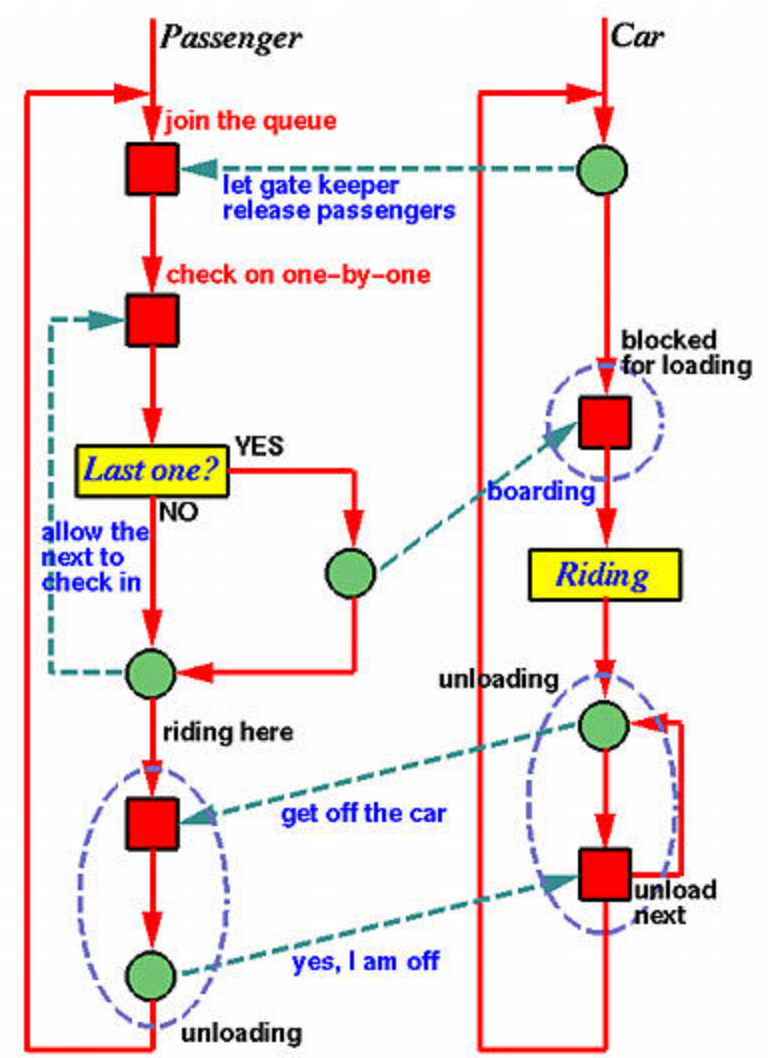
8. Um exemplo muito comum de controle de processo é o Problema da Montanha Russa. Nesse problema, existem 3 tipos de threads:

- 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.



9. Implemente um programa em pthreads que encontre todos os número primos menores que um natural N, usando o crivo de Eratóstenes (versão simplificada):

1. Crie um array de bool com tamanho **N** com todos os elementos sendo *true* (ou 1)
2. Ignore a posição 0 e 1 do array (não são considerados primos) : array[0] = array[1] = false (ou 0);
3. Comece com o primeiro primo: 2;
4. Elimine todos os múltiplos do primo escolhido que estão no array: array[múltiplo] = False;
5. Pegue o primeiro número não eliminado que é maior que o anterior (ex:3) . Ele também é primo. Em seguida. volte para o passo anterior;
6. se não existir esse próximo primo termine o algoritmo

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 todos os números primos já tenham sido encontrados (imprima eles no terminal). Cada thread deverá fazer a eliminação de múltiplos de primos diferentes (as threads não devem pegar o mesmo primo para eliminar em nenhum momento). Assim que uma thread terminar de eliminar os múltiplos de um número, ela imediatamente deverá testar um outro número.

Ex: Entrada: Número T: 3
Número N: 11

Saída: 2, 3, 5, 7

OBS: caso haja problema com duas threads eliminando um número simultaneamente (ex: thread 2 elimina o 6 e thread 3 elimina o 6 ao mesmo tempo), implemente um mecanismo baseado em mutex que garanta exclusão mútua na hora de escolher o próximo número. Ademais, cuidado com a condição de disputa ao array de booleanos..