O projeto valerá de 0 a 10, e será levado 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) Nos aeroportos, existem telas que informam os voos que estão chegando e partindo. Estas telas são atualizadas sempre que uma nova informação situação dos voos 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 Buenos Aires 17:45
XYZ001 Tóquio       17:50
ABC789 Moscou       18:20
FFF305 Estocolmo    18:35
QRS111 Madrid       18:45
DEF321 Nairóbi      19:00
GHI456 Quito        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 voo na linha 3 da tabela, outra thread pode atualizar a linha 5.Haverá número ***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 Recife 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 Tóquio       17:50
ABC789 Recife       19:00
FFF305 Estocolmo    18:35
QRS111 Madrid       18:45
DEF321 Nairóbi      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.

Devido à proximidade do 1o EE, você poderá implementar a tela usando uma matriz de caracteres com L linhas que pode ser inicializada no próprio código. Após uma

atualização de linha, imprima a nova tela de voos. Para entrega final do projeto, deverá modificar o código para usar o padrão ANSI especificado no parágrafo anterior

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 múltua 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.*

2) 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

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

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
```

```
        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
```

$x_1^{(k+1)} = 1/2 * (11 - x_2^{(k)})$

$x_2^{(k+1)} = 1/7 * (13 - 5x_1^{(k)})$

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

Exemplo de execução

k=0

$x_1^{(1)} = 1/2 * (11 - x_2^{(0)}) = 1/2 *(11-1) = 5$

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

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.*

3) Implemente famoso jogo “pedra-papel-tesoura” usando threads. O programa deverá ter **T** threads, no qual cada um representará um jogador. Como entrada, o usuário deverá informar quantidade de jogadores **T** e a quantidade de rodadas **N**. A saída consiste em apresentar uma mensagem mostrando o placar final. O jogo será automático, havendo somente a interação com o usuário no início.

Barriers são uma excelente ferramenta para essa questão. Caso precise, use a função rand da biblioteca stdlib para gerar números aleatórios.

4) Você deverá implementar um sistema computacional de controle de uma frota táxis aéreos autônomos usando C e *pThreads*. O sistema é responsável por controlar 5 estações de embarque, as quais só permitem **2 aerotáxis simultaneamente**. Todavia, um aerotáxi parando em uma estação **não deve** afetar (bloquear) a parada dos outros táxis em outras estações. Uma estação de embarque é representada por uma variável inteira. Um aerotáxi parando em uma estação (com menos de 2 táxis no momento) deverá modificar a **variável contador da estação** indicando a quantidade de táxis nesta, e esperar **500 milissegundos** para indicar o término do embarque. Ao concluir o embarque na estação, deverá decrementar o respectivo contador em uma unidade. O aerotáxi, que liberar uma estação, somente deverá notificar algum aerotáxi que esteja aguardando a liberação desta estação.
Por exemplo: Aerotáxi 1 e Aerotáxi 2 estão na estação 5, e Aerotáxi 3 está aguardando a liberação desta estação (pois acabou de passar pela estação 4). Aerotáxi 1, ao sair da estação 5, disponibilizará uma vaga na estação, mas só deverá notificar sua saída para o Aerotáxi 3 (pois é o único aguardando a estação 5). Os táxis deverão ser implementados usando threads, as quais precisam acessar as estações na sequência 1,2,3,4,5. Ao concluir o percurso, cada aerotáxi começará a trafegar novamente a partir do início (ou seja, a partir da estação 1). Assuma a existência de 10 aerotáxis autônomos.

A imagem serve como inspiração.

5) Muito satisfeitos com seu trabalho, a empresa de táxis aéreos solicitou que você desenvolva mais um sistema computacional utilizando C e pthreads. Você deve controlar o embarque de passageiros. Em uma estação, há 3 filas onde os passageiros aguardam para embarcar, em 2 aerotáxis esperando de cada vez.
Cada passageiro que vai saindo da fila é direcionado ao aerotáxi que estiver com mais vagas disponíveis naquele momento. Cada fila suporta N pessoas, sendo N > 10, e um aerotáxi pode levar 4 passageiros. Cada passageiro demora 200ms para embarcar, e quando o táxi está preenchido, ele parte. Após 500ms, outro aerotáxi, vazio, vem para a estação.

Obs.:

- Cada fila é representada por uma thread
- A capacidade de um aerotáxi é 4 passageiros.
- Se necessário, pode definir outras threads, como por exemplo, uma thread de controle
- As filas podem esvaziar e os aerotáxis não estarem com a capacidade máxima preenchida. Neste caso, os táxis devem partir e o programa finalizar.
- Não importa a ordem em que os passageiros saem das filas.

6) Usando C e Pthreads, implemente um sistema que simula o roteamento de pacotes através de uma rede de computadores. O sistema é composto por vários clientes (computadores que enviam pacotes de dados), servidores (computadores que recebem pacotes) e um único roteador, responsável por encaminhar os pacotes de um cliente para um servidor.
Um cliente se comunica com o roteador através de um buffer compartilhado com todos os outros clientes. O roteador, por sua vez, se comunica com os servidores de maneira semelhante, **com a diferença que cada servidor tem seu próprio buffer**. **O roteador comporta até cinco pacotes de clientes em seu buffer e cada servidor acomoda dez pacotes.**
Um pacote de um cliente é representado por uma estrutura de dados contendo o seu número de identificação e o número de identificação do servidor que deseja contatar. O número do servidor a ser chamado é determinado aletoriamente (**sugestão: use a função rand da biblioteca stdlib**). Desta maneira, ao enviar um pacote, o cliente coloca-o no buffer e o roteador faz o repasse ao servidor de destino, armazenando o pacote recebido no buffer do servidor.
Ao terminar de fazer todas suas solicitações, um cliente deve aguardar todos os outros acabarem. Quando todos acabarem, o roteador deve ser notificado e informar aos servidores para que parem de esperar pacotes nas suas filas e terminem de processar os que ainda existem.

Defina uma constante C para a quantidade de clientes, R para a quantidade de requisições por cliente e S para a de servidores. A solução deverá ser genérica para qualquer quantidade.