O projeto valerá de 0 a 10 e será levada em consideração a organização dos códigos de ecada 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. Uma rede de supermercados precisa de um novo sistema para contabilizar as vendas de produtos, nos quais as vendas são armazenadas em arquivos textos.. Faça um programa que receba um número ***N*** de arquivos. um número ***T <= N*** de *threads* que deverão ser utilizadas para fazer a contagem, e um número ***P*** de de produtos . Em seguida, o programa deverá abrir os ***N*** arquivos nomeados ***"x.in"***, em quel ***1 <= x <= N***.. Cada arquivo terá 1 produto por linha que será um número ***y | 0 <= y <= P*** ; Cada *thread* deverá pegar um arquivo. Quando uma *thread* concluir a leitura de um arquivo, e houver um arquivo ainda não lido, a *thread* deverá ler algum arquivo pendente. Ao final, imprima na tela. o total de cada produto,.

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 é a leitura dos 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 o array que guardará a quantidade de cada produto. Uma implementação mais refinada garante a exclusão mútua separada para cada posição do array. Mais especificamente, enquanto um produto **x** está sendo contabilizado e a thread modificando o array na respectiva posição, uma outra thread pode modificar o array em uma posição **y** que representa outro produto. Ou seja, se o array de produtos possui tamanho 10, haverá um outro array de 10 mutex, um para cada posição do vetor de produtos. Ao ler um arquivo e detectar o produto y, a thread trava o mutex relativo à posição y, incrementa a quantidade de produtos, e destrava o mutex na posição y. Obviamente, se mais de uma thread quiser modificar a mesma posição do array de produtos simultaneamente, somente 1 terá acesso, e as outras estarão bloqueadas. O mutex garantirá a exclusão mútua na posição.*

2. QuickSort é um algoritmo de ordenação que utiliza a abordagem dividir para conquistar.
https://pt.wikipedia.org/wiki/Quicksort., Animação: Multithread Quicksort

```
Qsort( l , r ):
        if( l  <  r ):
                pivot_index = partition( l , r )
                Qsort( l , pivot_index - 1 )
                Qsort( pivot_index+1 , r )
```

Perceba que, enquanto o lado esquerdo não for resolvido, o algoritmo não começará a resolver o problema do lado direito.
Como o problema da esquerda e o da direita operam em intervalos disjuntos, uma melhoria seria criar uma *thread* para cada lado. Considerando a abordagem recursiva apresentada, para cada chamada de Qsort, uma thread deverá ser criada.
Implemente o algoritmo usando *pthreads,* recebendo um vetor de inteiros colm entrada e exibindo o resultado final após a conclusão do algoritmo.

3.. 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.*

4. DFS é um dos algoritmos mais conhecidos de busca em grafo https://en.wikipedia.org/wiki/Depth-first_search, animação: DFS in a tree DFS animation

Neste método busca,, uma versão com múltiplas *threads* poderia visitar caminhos diferentes simultâneamento em um mesmo grafo

Considerando um grafo G e um inteiro *N* de threads, implemente uma versão DFS com *pthreads*
Quando uma *thread* visitar um nó, imprima no terminal: “thread id visitou node_id\n”,
em que node_id é o id do *node* e id é o id da *thread*.

5. 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 (**Blocking Queue**) 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 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 inSizeBuffer);
void putBlockingQueue(BlockingQueue* Q, int newValue);
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 poderão ser inicializados estaticamente.

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

6. OMP (Open MP) é uma API para C muito usada para concorrente, permitindo implementações sucintas, mas poderosas. Um simples `#pragma omp parallel` antes de chaves faz com que o respectivo escopo seja repetidos **N** vezes por **N** threads (**N** definido por variavel de ambiente). Adicionalmente, um `#pragma omp parallel for` antes de um *for* faz com que esse laço seja executado concorrentemente, de tal forma que cada iteração só ocorra uma vez, mas que todas ocorram até o final do for em qualquer ordem. Suas implementações são dependentes da máquina, compilador, sistema operacional e outros componentes. Às vezes, OMP é até implementado usando Pthreads. **Sua missão é implementar uma função em C que simule um `#pragma omp parallel for` em Pthread.**

OMP é uma API "inteligente", pois não cria threads desnecessárias, nem destrói e cria outras threads sem motivo. Dentre suas variáveis de ambiente, há uma variavel chamada OMP_NUM_THREADS que será aqui emulada por uma macro de mesmo nome. Usualmente, essa variável tem a mesma quantidade de núcleos que o sistema possui, a fim de usar completamente os recursos da máquina, mas sem gerar *overheads* desnecessários.

Quando um *for* aparece no código, OMP entrega a cada uma das **N** threads uma parcela do trabalho daquele *for*, e espera que todas as **N** threads acabem seu trabalho. Como exemplo temos :

```
#pragma omp for parallel
for(int i = 0 ; i < 10 ; i++ )
{
     CODIGO...
}
```

Se o número de threads a ser criada é 4, OMP distribuiria o trabalho possivelmente assim:

Thread 0: `for( int i = 0; i < 3  ; i++ ) CODIGO...`
Thread 1: `for( int i = 3; i < 6  ; i++ ) CODIGO...`
Thread 2: `for( int i = 6; i < 8  ; i++ ) CODIGO...`

Thread 3: `for( int i = 8; i < 10 ; i++ ) CODIGO...`
Espera as threads acabarem.

Repare que nenhum trabalho é realizado mais de uma vez, e que nenhuma *thread* pegou mais trabalho que a outra de forma considerável. Adicionalmente, nenhum trabalho não solicitado não foi distribuído (como uma implementação errada poderia fazer as Threads 2 e 3 terem também 3 iterações, o que transformaria o for entre 0 e 12! Um absurdo).

No final OMP esperaria que todas as 4 threads acabassem para continuar.

A forma de divisão de trabalho pela OMP não é tão rigido. Sabemos que `#omp for parallel` possui uma cláusula chamada `schedule` (definida por padrão quando omitida) que recebe dois argumentos: o tipo de escalonamento das threads e o tamanho do *chunk_size*. Os tipos de escalonamento são: *static, dynamic, guided, runtime,auto*. O *chunk_size* é um valor entre 1 e o número total de iterações.

Em outras palavras, o *chunk_size* é o quanto as iterações devem estar juntas. Por exemplo, com *chunk_size* igual a 3, as iterações são passadas de 3 em 3 para as *threads*, respeitando o final das iterações. No primeiro exemplo desta questão, o *chunk_size* era igual a 3, mesmo assim as ultimas threads pegaram 2 iterações cada (o mais próximo de 3 possível).

Segue agora as definições dos schedules:

static : Antes de todas as threads rodarem, já é definido quais iterações cada thread vai executar, e não há nenhuma comunicação entre a thread pai e as filhas até elas terminarem. As iterações são passadas em *round-robi*n a cada *chunk_size*. Exemplo : Um *for* entre 0 e 100(exclusivo) com *chunk_size* igual a 10 e 4 threads seria dividido (deterministicamente):

Thread 0: 0-9 , 40-49, 80-84
Thread 1: 10-19, 50-59, 85-89
Thread 2: 20-29, 60-69, 90-94
Thread 3: 30-39, 70-79, 95-99

dynamic: As threads são inicializadas sem iterações definidas e conforme vão necessitando elas pedem à thread pai mais iterações. A thread pai deve passar tantas iterações quanto sejam definidas em *chunk_size* na ordem original desde que não ultrapasse o límite definido no *for*. Caso não haja mais trabalho a thread deve ser fechada. Reparem que condições de corrida são implícitas nesse modelo. Exemplo: Um *for* entre 0 e 15(exclusivo) com *chunk_size* igual a 2 e 4 threads será executado possivelmente assim:

Thread 0 a 3 são criadas.
Thread 0 pede iterações: Recebe iterações 0-1
Thread 2 pede iterações: Rebece iterações 2-3
Thread 3 pede iterações: Recebe iterações 4-5
Thread 1 pede iterações: Rebece iterações 6-7
Thread 3 pede iterações: Rebece iterações 8-9
Thread 3 pede iterações: Rebece iterações 10-11
Thread 0 pede iterações: Recebe iterações 12-13
Thread 2 pede iterações: Rebece iterações 14
Thread 1 pede iterações: É fechada.
Thread 2 pede iterações: É fechada.
Thread 3 pede iterações: É fechada.
Thread 0 pede iterações: É fechada.

guided: É o mesmo que *dynamic,* mas com tamanho de iterações passadas igualmente para cada *thread*, diminuindo a cada etapa e nunca menor que o *chunk_size* passado como parâmetro. A cada momento que novas iterações sejam executadas pelas threads, o número de iterações a serem repassadas deve ser: (iterações restantes)/(número de threads), mas respeitando o mínimo relativo ao *chunk_size*. Exemplo: Um *for* entre 0 e 15(exclusivo) com chunk_size igual a 2 e 4 threads será executado possivelmente assim:

Thread 0 a 3 são criadas.
Thread 0 pede iterações: Recebe iterações 0-3 (teto de 15/4)
Thread 2 pede iterações: Rebece iterações 4-6 (teto de 11/4)
Thread 3 pede iterações: Recebe iterações 7-8 (teto de 8/4)
Thread 1 pede iterações: Rebece iterações 9-10 (teto de 6/4<chunk_size = 2)
Thread 3 pede iterações: Rebece iterações 11-12 (teto de 4/4<chunk_size = 2)
Thread 3 pede iterações: Rebece iterações 13-14 (teto de 2/4<chunk_size = 2)
Thread 1 pede iterações: É fechada.
Thread 2 pede iterações: É fechada.
Thread 3 pede iterações: É fechada.
Thread 0 pede iterações: É fechada.

runtime: Em tempo de execução OMP vê em suas variaveis de ambiente quais das 3 outras politicas de escalonamento ele deve executar neste *for*. **Não precisa ser implementado nessa questão.**

auto: O mesmo que runtime, mas definido pelo compilador. **Igualmente não precisa ser implementado nessa questão.**

Implemente a função em C com a seguinte assinatura para simular a API OMP:

```
void omp_for( int inicio , int passo , int final , int schedule
, int chunk_size , void (*f)(int) );
```

A qual simularia um :

```
#pragma omp for
for(int i = inicio ; i < final ; i += passo )
{
     f(i);
}
```

Repare que `f` é um ponteiro de função e `schedule` deve receber um valor entre vários tipos enumerados ou macros para cada uma das 3 politicas de escalonamento.