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. 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

$x_1^{(k+1)} = 1/2 * (11 - x_2^{(k)})$
$x_2^{(k+1)} = 1/7 * (13 - 5x_1^{(k)})$
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. O bitmap é uma estrutura de dados utilizada em muitas áreas da computação. Na computação gráfica, por exemplo, pode representar uma imagem, no qual 1 seria um pixel preto e 0 seria um pixel branco. Entretanto, ele pode ser representado de diversas formas, o que torna necessário termos um modo eficiente de transformar em diferentes representações.
A primeira representação é simplesmente uma array bidimensional composto de 0s e 1s, e a segunda é gerada a partir de uma decomposição dessa matriz.

Primeiro, o bitmap inteiro é considerado. Se todos os bits forem 1, então 1 é a saída. Se todos forem 0, então 0 é a saída. Caso contrário, a saída deve ser em um formato D e o bitmap deve ser dividido em até 4 partes. Cada uma das partes deve ser processada da mesma maneira que o bitmap original.

Os 4 quadrantes são processados na seguinte ordem: superior esquerdo, superior direito, inferior esquerdo, inferior direito. Quando um bitmap possui um número par de linhas e também de colunas, então todos os quadrantes possuirão as mesmas dimensões.
Quando o número de colunas é ímpar, os quadrantes da esquerda devem ter uma coluna a mais do que os quadrantes da direita. Quando o número de linhas é ímpar, os quadrantes superiores devem possuir uma linha a mais do que os quadrantes inferiores. Seu objetivo é ler um bitmap no formato de matriz e transformá-lo na segunda representação usando N threads.

EXEMPLOS:

Entrada:
1 1 //A primeira linha possui a quantidade de linhas colunas
1

Saída:
1

Entrada:
2 2
00
00

Saída:
0

Entrada:
2 2
01
00

Saída:
D0100

Entrada:
3 4
0010
0001

1011

Saída:
D0D1001D101

Entrada:
4 4
0101
0101
0101
0101

Saída:
DD0101D0101D0101D0101

4**.** Com a aproximação das eleições, um estado começou a usar um novo sistema eletrônico de contabilização de votos para prefeito. Após o período de eleição, os votos de cada candidato estão em um arquivo e precisam agora ser contabilizados. Faça um programa que receba um número **N** de arquivos, um **número T <= N** de threads utilizadas para fazer a contagem e um número **C** de candidatos a prefeito. Em seguida, o programa deverá abrir os **N** arquivos nomeados "x.in" no qual **1 <= x <= N**. Cada arquivo terá 1 voto por linha que será um número **y | 0 <= y <= C,** no qual 0 significa voto em branco, 1 significa voto ao candidato 1 e assim sucessivamente. 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 votos, a porcentagem de votos recebidos por cada candidato e o número do candidato vencedor (que será o candidato com mais votos não importando a porcentagem).

Assumindo que não se tem conhecimento prévio da quantidade de threads e arquivos (o usuário os dará como entrada), as threads devem ler os arquivos sob demanda. A partir do momento que uma thread termina a leitura de um arquivo pega qualquer outro (ainda não lido) dinamicamente.

Ademais, deve-se garantir a exclusão múltua ao alterar o array que guardará a quantidade de votos de cada candidato. ***Porém, você deverá assumir uma implementação refinada.*** *Uma implementação refinada garante a exclusão mútua separada para cada posição do array. Mais especificamente, enquanto um voto está sendo contabilizado para um candidato* ***x*** *e modificando o array na respectiva posição, uma outra thread pode modificar o array em uma posição* ***y*** *que representação outro candidato. Ou seja, se o array de produtos possui tamanho*

10, haverá um outro array de 10 mutex, um para cada posição do vetor de candidatos. Ao ler um arquivo e detectar um voto para o candidato y, a thread trava o mutex relativo à posição y, incrementa a quantidade de votos deste candidato, e destrava o mutex na posição y. Obviamente, se mais de uma thread quiser modificar a mesma posição do array de candidatos simultaneamente, somente 1 terá acesso, e as outras estarão bloqueadas. O mutex garantirá a exclusão mútua na posição.

5. Um dos algoritmos mais conhecidos na computação é o Merge Sort, o qual é um algoritmo de ordenação por comparação do tipo dividir-para-conquistar.
https://pt.wikipedia.org/wiki/Merge_sort

```
sort(l, r){
        printf("Ordenando [%d, %d]\n", l, r);
        if(l == r) return;
        m = (l+r)/2
        sort(l, m)
        sort(m+1, r)
        merge(l, m, r)
        printf("Ordenado [%d, %d]\n", l, r);
}
```

Uma coisa a se notar é que enquanto o problema do 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 boa otimização seria resolvê-los em paralelo.

Implemente esta otimização usando pthreads, imprimido o vetor final ordenado.

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.*