

TRABALHO PRÁTICO II

Disciplina: Arquitetura e Organização de Computadores

Professor: Saulo Henrique Cabral Silva

Alunos: Eduardo Octávio de Paula e Rodolfo Oliveira Miranda

Programação Paralela

1 Introdução

A programação paralela é uma técnica que permite a execução simultânea de múltiplas tarefas, visando aumentar a eficiência e desempenho de aplicações, especialmente em sistemas com múltiplos processadores ou núcleos. Em Java, uma das maneiras mais simples e tradicionais de implementar a programação paralela é através da classe *"Thread"*.

1.1 Classe *"Thread"*

Em Java, uma *"Thread"* representa uma unidade independente de execução dentro de um programa. Cada uma possui seu próprio conjunto de instruções e pode ser executada de forma simultânea com outras threads. Isso permite que um programa execute várias operações ao mesmo tempo, melhorando a utilização dos recursos do sistema e reduzindo o tempo de execução das tarefas.

```
4 usages   Rodolfo Miranda
public class ImageCorrector extends Thread{

    5 usages
    private final int[][] originalImage;
    3 usages
    private final int[][] correctedImage;
    2 usages
    private final int startX, endX, startY, endY;
```

Exemplo de Classe que estende de *"Thread"*.

1.2 Desafio Proposto

Para este trabalho, foi proposto a implementação de programação paralela com intuito de aprendizado sobre Paralelismo utilizando a classe "*Thread*" anteriormente apresenta.

Para isso, foi nos dado duas atividades, sendo elas: A **correção de imagens** com pixels defeituosos e a **quebra de senhas** de arquivos compactados.

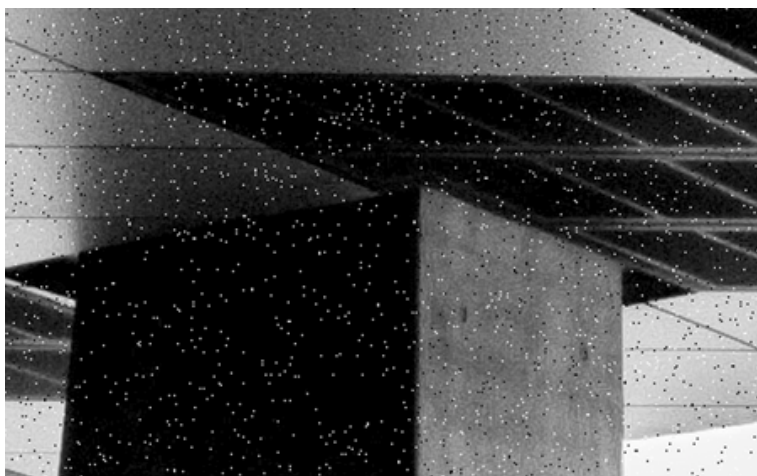
2 Correção de Imagens

No campo da tecnologia de imagem digital, a qualidade e precisão das imagens são cruciais para diversas aplicações, incluindo fotografia, medicina, segurança e visão computacional. Com a crescente demanda por imagens de alta qualidade, técnicas de correção de imagem têm se tornado cada vez mais importantes para aprimorar e restaurar imagens degradadas.

Este projeto tem como objetivo a implementação de algoritmos que utilizam de técnicas para redução de ruídos para corrigir pixels defeituosos em imagens utilizando o conceito de programação paralela.

2.1. "O Problema"

Foi nos dados imagens em escala de cinza com pixels defeituosos, isso significa que eles podem possuir valor igual a zero (totalmente pretos) ou 255 (totalmente brancos).



Exemplo de pixels defeituosos.

Nosso objetivo é verificar quais pixels precisam ser corrigidos e gerar uma nova imagem sem ruídos.

2.2 Desenvolvimento

Para o desenvolvimento deste projeto, foram disponibilizadas antecipadamente, pelo professor, partes do código necessárias para concluir a tarefa proposta.

Os códigos disponibilizado foram:

Função que “lê” os pixels da imagem e os guarda em uma matriz de inteiros.

```
public class PixelsReader {
    1 usage  👤 Rodolfo Miranda
    public static int[][] readPixels(File file) {

        BufferedImage bufferedImage;
        try {
            bufferedImage = ImageIO.read(file);
            int width = bufferedImage.getWidth( observer: null);
            int height = bufferedImage.getHeight( observer: null);

            int[][] pixels = new int[width][height];

            for (int i = 0; i < width; i++) {
                for (int j = 0; j < height; j++) {
                    copyGreyScale(pixels, i, j, bufferedImage);
                }
            }

            return pixels;
        } catch (IOException ex) {
            System.err.println("Erro no caminho indicado pela imagem");
        }
        return null;
    }
    1 usage  👤 Rodolfo Miranda
    private static void copyGreyScale(int[][] pixel, int i, int j, BufferedImage bufferedImage) {
        float red = new Color(bufferedImage.getRGB(i, j)).getRed();
        float green = new Color(bufferedImage.getRGB(i, j)).getGreen();
        float blue = new Color(bufferedImage.getRGB(i, j)).getBlue();
        int greyScale = (int) (red + green + blue) / 3;
        pixel[i][j] = greyScale;
    }
}
```

Função que grava uma nova imagem com a matriz de pixels corrigidos.

```
public class PixelsRecorder {  
  
    1 usage  👤 Rodolfo Miranda  
    public static void recordPixels(String pathToRecord, int pixels[][]) {  
  
        pathToRecord = pathToRecord  
            .replace(target: ".png", replacement: "_modificado.png")  
            .replace(target: ".jpg", replacement: "_modificado.jpg");  
  
        int width = pixels.length;  
        int height = pixels[0].length;  
  
        BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_BYTE_GRAY);  
  
        byte[] bytesPixels = new byte[width * height];  
        for (int x = 0; x < width; x++) {  
            for (int y = 0; y < height; y++) {  
                bytesPixels[y * (width) + x] = (byte) pixels[x][y];  
            }  
        }  
  
        image.getRaster().setDataElements(x: 0, y: 0, width, height, bytesPixels);  
  
        File ImageFile = new File(pathToRecord);  
        try {  
            ImageIO.write(image, formatName: "png", ImageFile);  
            System.out.println("Nova Imagem dispon. em: " + pathToRecord);  
        } catch (IOException e) {  
            System.err.println("Erro no caminho indicado pela imagem");  
        }  
    }  
}
```

2.2.1 Iniciando threads

Após os pixels da imagem defeituosa serem convertidos em uma matriz de inteiros é chamado o método que irá dividi-la para as threads disponíveis.

```
public static int[][] correctPixels(int[][] originalImage) throws InterruptedException {  
  
    //Pegando os valores das linhas e colunas da matriz e gerando uma nova para guardar os pixels corrigidos  
    int row = originalImage.length;  
    int column = originalImage[0].length;  
    int[][] correctedImage = new int[row][column];  
  
    //Pegando a quantidade de threads disponíveis  
    int numberOfThreads = Runtime.getRuntime().availableProcessors();  
  
    //Criando uma lista de threads  
    List<ImageCorrector> threads = new ArrayList<>(numberOfThreads);  
  
    //Dividindo o número de linhas para cada thread  
    int chunkHeight = row / numberOfThreads;  
  
    for (int i = 0; i < numberOfThreads; i++) {  
        //Definindo os pontos de início e fim para cada thread  
        int startX = i * chunkHeight;  
        int endX = (i == numberOfThreads - 1) ? row : (i + 1) * chunkHeight;  
  
        //Instanciando uma nova thread, adicionando-a na lista de Threads e a iniciando  
        ImageCorrector thread = new ImageCorrector(originalImage, correctedImage, startX, endX, startY: 0, column);  
        threads.add(thread);  
        thread.start();  
    }  
  
    //Esperando todas as threads terminarem de corrigir os pixels  
    for (ImageCorrector thread : threads) {  
        thread.join();  
    }  
}
```

Primeiro é criado uma nova matriz com tamanho igual ao da matriz a ser corrigida.

```
int row = originalImage.length;  
int column = originalImage[0].length;  
int[][] correctedImage = new int[row][column];
```

Após isso, é verificado quantas núcleos do CPU estão disponíveis para uso. E também é instanciado uma “List” para encadear as threads que serão instanciadas.

```
//Pegando a quantidade de threads disponíveis
int numberOfThreads = Runtime.getRuntime().availableProcessors();

//Criando uma lista de threads
List<ImageCorrector> threads = new ArrayList<>(numberOfThreads);
```

Para utilizar do processo de paralelização das threads, é definido o tamanho das “chunks” da matriz a ser corrigida para atribui-las às threads. O tamanho da chunk é o resultado da fração entre a quantidade de linhas pelo número de núcleos disponíveis.

```
//Dividindo o número de linhas para cada thread
int chunkHeight = row / numberOfThreads;
```

Logo após, é utilizado um “for”, iniciado em zero, que irá instanciar as threads. Antes disso, é definido os pontos de início e fim que cada thread irá percorrer da matriz.

O ponto de início (“startX”) será igual a multiplicação entre o valor do contador, que refere à um thread, e o tamanho da chunk.

Exemplo: Caso o valor do contador seja zero (indicando que é a primeira thread a ser instanciada) e o tamanho da chunk seja 1.000, o resultado dessa multiplicação será zero. Logo, a primeira thread irá começar a percorrer pela linha zero.

Para definir o ponto de parada da thread, é feita uma comparação. Caso o valor do contador seja igual ao número de threads disponíveis menos um, significa que esta é a ultima thread, logo “endX” será igual ao número de linhas. Caso contrário, será igual à multiplicação entre soma do contador mais um e o tamanho da chunk.

```
for (int counter = 0; counter < numberOfThreads; counter++) {
    //Definindo os pontos de início e fim para cada thread
    int startX = counter * chunkHeight;
    int endX = (counter == numberOfThreads - 1)
        ? row
        : (counter + 1) * chunkHeight;
```

Em seguida, é instanciado a thread referente ao número do contador. No construtor da thread tem como parâmetros a matriz a ser corrigida, a nova matriz onde os pixels corrigidos serão armazenados, os pontos de início e fim das linhas que a thread irá percorrer e os pontos de início e fim das colunas, que possuem mesmo valor para todas as thread, sendo eles zero e o máximo de colunas da matriz, respectivamente.

Depois a thread é adicionada na lista de threads e é iniciada.

```
//Instanciando uma nova thread, adicionando-a na Lista de Threads e a iniciando
ImageCorrector thread = new ImageCorrector(originalImage, correctedImage,
    startX, endX, startY: 0, column);
threads.add(thread);
thread.start();
}
```

Quando todas as threads são iniciadas, é aguardado a finalização dos processos de cada uma para que a matriz com os valores corrigidos seja retornada.

```
//Esperando todas as threads terminarem de corrigir os pixels
for (ImageCorrector thread : threads) {
    thread.join();
}

//Retornando a matriz corrigida
return correctedImage;
}
```

Quando a thread é iniciada, o método sobrescrito “run()” é acionado. Ele possui dois “for’s” que percorrem os pixels da matriz a ser corrigida. Dentro deste loop, possui uma comparação com o valor de cada pixel. Caso este seja igual à zero (preto) ou 255 (branco), o método de corrigir pixel é acionado e o valor retornado é copiado para a nova matriz na posição do pixel defeituoso. Caso contrário, o valor é apenas copiado para a nova matriz.

```
@Override
public void run() {
    //Os pontos de início e fim foram definidos quando a thread foi instanciada
    for (int i = startX; i < endX; i++) {
        for (int j = startY; j < endY; j++) {
            //Verificando se o pixel possui valor 0 (preto) ou branco (255) para que seja corrigido
            //Caso verdadeiro, ele será corrigido
            //Caso falso, ele irá copiar o valor da matriz original
            if (originalImage[i][j] == 255 || originalImage[i][j] == 0) {
                correctedImage[i][j] = CorrectPixels.correctPixel(originalImage, i, j);
            } else {
                correctedImage[i][j] = originalImage[i][j];
            }
        }
    }
}
```


2.2.2 Corrigindo a Imagem

Quando o método de corrigir pixel é acionado é feito os seguintes passos.

Primeiro, criamos uma variável para armazenar a soma dos valores dos pixels que estão em volta daquele que queremos corrigir. Também criamos uma variável para armazenar a quantidade de pixels que percorremos e outra para armazenar a quantidade de pixels pretos percorridos.

```
public static int correctPixel(int[][] originalImage, int i, int j) {  
    //Soma dos valores dos pixels em volta do pixel a se corrigir  
    int valuePixelsSum = 0;  
    //Número de pixels em volta do pixel a se corrigir  
    int pixelsNumb = 0;  
    //Número de pixels pretos em volta do pixel a se corrigir  
    int blackPixelsNumb = 0;
```

Em seguida, é percorrida uma área 3x3, sendo o pixel central aquele que queremos corrigir, utilizando dois “for’s”.

Dentro do loop, existe uma comparação que verifica se os contadores não são iguais à valores inacessíveis da matriz original. Isso significa que, caso o pixel a ser corrigir for um pixel de borda, o loop não tentará acessar pixels inexistentes.

```
//Percorrendo os pixels em volta do pixel a se corrigir  
for (int i1 = -1; i1 <= 1; i1++) {  
    for (int j1 = -1; j1 <= 1; j1++) {  
        //Verificando se o valor dos contadores não são negativos,  
        // para que não tente acessar valores inalcançáveis  
        if (j1 + j >= 0  
            && j1 + j < originalImage[i].length  
            && i1 + i >= 0 && i1 + i < originalImage.length) {
```


Caso a posição do pixel seja acessível, é verificado se ele tem posição igual ao pixel que queremos corrigir, e se verdadeiro, ele é simplesmente ignorado, caso falso os seguintes passos são feitos:

1. É verificado se o valor deste pixel é zero (preto), se sim, é somado mais um na quantidade de pixels pretos;
2. Em seguida, é somado o valor dele a soma dos valores dos pixels;
3. Por fim, é somado mais um na quantidade de pixels percorridos;

```
//Caso a posição do pixel percorrido seja igual a posição do pixel a
// se corrigir é passado para o próximo pixel
if (i1 + i == i && j1 + j == j) continue;

//Caso o pixel percorrido tenha valor 0 (preto) é somado 1
// ao número de pixels pretos
if (originalImage[i1 + i][j1 + j] == 0) blackPixelsNumb++;

//Somando o valor do pixel percorrido à soma dos
// valores dos pixels percorridos
valuePixelsSum += originalImage[i1 + i][j1 + j];

//Somando 1 ao número de pixels percorridos
pixelsNumb++;
```

No final do loop, é verificado se a quantidade de pixels pretos são maiores do que a quantidade de pixels não pretos. Caso verdadeiro, isso significa que a área em questão é escura e por isso, é retornado zero. Caso contrário, é feito a média dos valores dos pixels, usando a soma desses valores e a quantidade de pixels percorridos, retornando, assim, o valor corrigido referente ao pixel defeituoso.

```
//Caso o número de pixels pretos seja maior que a quantidade de pixels percorridos é retornado 0
if (blackPixelsNumb > (pixelsNumb - blackPixelsNumb)) {
    return 0;
}

//Caso contrário, é feito a média dos valores dos pixels percorridos
return valuePixelsSum / pixelsNumb;
```

2.2.3 Finalizando Correção

Depois de todos os pixels serem percorridos e e todas as threads finalizarem suas operações, a matriz de pixels corrigidos gerada é convertida em um novo arquivo no formato de imagem e salvo no mesmo diretório da imagem original.

Os procedimentos anteriormente apresentados serão realizados para todas as imagens presentes no diretório indicado.

Imagem antes da correção



Imagem após correção



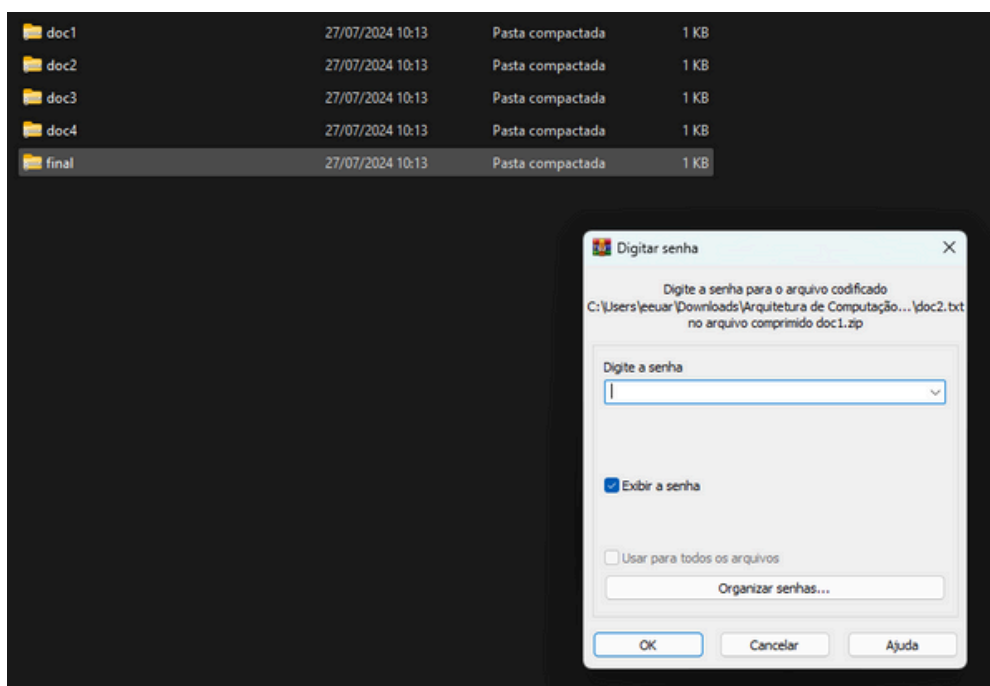
3 Quebra de Senhas

Senhas são uma das formas mais comuns de autenticação usadas para proteger informações digitais. Porém, não é a forma mais segura de proteger um dado. A quebra de senhas é uma ameaça constante, como métodos de ataques de força bruta ou ataques de hackers.

Este projeto tem como objetivo a implementação de algoritmos que irão quebrar a senha de arquivos que estão comprimidos utilizando o conceito de programação paralela.

3.1. “O Problema”

Foi nos dado cinco arquivos compactados com senha. Dentre estes arquivos, a senha do nomeado “final” foi dividida e escondida dentro dos demais arquivos. Nosso objetivo é desenvolver um algoritmo que quebre estes quatro arquivos, a fim de encontrar as partes da senha final, podendo assim, acessar o último arquivo.



Arquivos a serem quebrados.

3.2 Desenvolvimento

Para o desenvolvimento deste projeto, foram disponibilizadas antecipadamente, pelo professor, partes do código necessárias para concluir a tarefa proposta.

Os códigos disponibilizado foram:

Função que define a senha e tenta descompactar o arquivo.

```
private boolean testPassword(String password) {
    try {
        zipFile.setPassword(password.toCharArray());

        List<FileHeader> fileHeaderList = zipFile.getFileHeaders();

        for (FileHeader header : fileHeaderList) {
            zipFile.extractFile(header, Main.path);
            System.out.println("Encontramos a senha e o arquivo: " + password);
            return true;
        }
    } catch (net.lingala.zip4j.exception.ZipException ex) {
        return false;
    }

    return false;
}
```

Observação: algumas alterações foram feitas, mas a lógica é a mesma.

3.2.1 Definindo os arquivos

Inicialmente, é necessário definir quais arquivos deverão ser quebrados. Estes arquivos serão definidos em fila, isso significa que todas as threads deverão tentar quebrar os arquivos em sequência.

```
public static void main(String[] args) throws InterruptedException, IOException {

    //Senha do arquivo final
    String finalPassword = "";

    for (int i = 1; i <= 4; i++){

        //Localiza o arquivo a ser descompactado
        ZipFile file = new ZipFile(path + "doc" + i + ".zip");

        //Define o arquivo que as threads devem quebrar a senha
        FileBreaker.setZipFile(file);

        //Tenta quebrar a senha do arquivo
        BreakFile.breakFile();

        //Redefine se a senha já foi quebrada para 'false' para o próximo arquivo a ser quebrado
        FileBreaker.isPasswordBroken = false;
    }
}
```

3.2.2 Quebrando o arquivo

Depois que o arquivo a ser quebrado é definido, o método que instância as Threads é acionado.

```
public static void breakFile() throws InterruptedException {

    //Número de caracteres da tabela ASCII (33 - 126)
    int numberOfCharsAscii = 93;

    //Caracter inicial
    int startOfAscii = 33;

    //Caracter final
    int endOfAscii = 126;

    //Número de Threads disponíveis
    int numberOfThreads = Runtime.getRuntime().availableProcessors();

    //Número de caracteres para cada thread
    int numberOfCharsPerThread = numberOfCharsAscii / numberOfThreads;

    //Lista para armazenar as threads
    List<FileBreaker> threads = new ArrayList<>(numberOfThreads);

    //Criação e Iniciação das threads
    for (int counter = 0; counter < numberOfThreads; counter++){

        //O valor inicial de uma determinada thread
        int start = startOfAscii + (numberOfCharsPerThread * counter);

        //Valor final de uma determinada thread
        int end = (counter != numberOfThreads - 1) ? startOfAscii + (numberOfCharsPerThread * (counter + 1)) : endOfAscii;

        //Instancia uma thread
        FileBreaker thread = new FileBreaker(start, end);

        //Adiciona a nova thread na lista
        threads.add(thread);

        //Inicia a thread
        thread.start();
    }

    //Espera que todas as threads terminem suas tarefas
    for(FileBreaker thread : threads){
        thread.join();
    }
}
```

Primeiro é definido algumas variáveis que serão utilizadas no método. Estas são:

- A quantidade de caracteres da tabela ASCII que serão usados;
- O valor do caractere inicial da tabela ASCII;
- O valor do caractere final da tabela ASCII;
- A quantidade de núcleos disponíveis;
- A quantidade de caracteres que cada thread deverá utilizar;
- A lista na qual será armazenada as threads;

```
//Número de caracteres da tabela ASCII (33 - 126)
int numberOfCharsAscii = 93;

//Caracter inicial
int startOfAscii = 33;

//Caracter finalA
int endOfAscii = 126;

//Número de Threads disponíveis
int numberOfThreads = Runtime.getRuntime().availableProcessors();

//Número de caracteres para cada thread
int numberOfCharsPerThread = numberOfCharsAscii / numberOfThreads;

//Lista para armazenar as threads
List<FileBreaker> threads = new ArrayList<>(numberOfThreads);
```

Logo após é iniciado um loop que irá instanciar e iniciar as threads. Antes disso, é definido o intervalo de caracteres que a thread irá percorrer.

O início é definido pelo valor inicial da tabela ASCII somado da multiplicação entre a quantidade de caracteres por thread e o número do contador.

Para a definição do final, é verificado se a thread a ser instanciada é a ultima. Caso seja, o final será igual ao valor do caractere final da tabela ASCII. Caso contrário, será igual ao valor inicial da tabela ASCII somado pela multiplicação entre a quantidade de caracteres por thread e o número do contador acrescentado de um.

Por fim, a thread é instanciada e adicionada na lista, logo em seguida é iniciada.


```

//Criação e Iniciação das threads
for (int counter = 0; counter < numberOfThreads; counter++){

    //O valor inicial de uma determinada thread
    int start = startOfAscii + (numberOfCharsPerThread * counter);

    //Valor final de uma determinada thread
    int end = (counter != numberOfThreads - 1)
        ? startOfAscii + (numberOfCharsPerThread * (counter + 1))
        : endOfAscii;

    //Instancia uma thread
    FileBreaker thread = new FileBreaker(start, end);

    //Adiciona a nova thread na lista
    threads.add(thread);

    //Inicia a thread
    thread.start();
}

```

Quando a thread é iniciada, o método sobrescrito “run()” é acionado. Nele, primeiro, é verificado se a senha ainda não foi encontrada. Caso tenha sido, a thread é “finalizada”. Caso contrário o é a classe que irá gerar e testar as senhas é acionada.

A senha pode variar de 1 a 3 caracteres, por tanto, é gerado, primeiro, senhas com um caractere, depois com dois caracteres, e por fim, três caracteres.

```

@Override
public void run() {

    //Caso a senha já tenha sido encontrada a thread é "finalizada"
    if (isPasswordBroken) return;

    //Instancia a classe que irá gerar e testar as senhas
    PasswordGenerator passwordGenerator = new PasswordGenerator(zipFile);

    //Primeiro testa uma senha com um caracter, depois com dois caracteres e por fim, com três caracteres
    for (int i = 1; i <= 3; i++) {

        //Testa novas senhas enquanto não encontrar a senha
        if (passwordGenerator.generatePassword(i, this.start, this.end)) {
            isPasswordBroken = true;
            return;
        }
    }
}

```


3.2.3 Gerando as senhas

Para gerar as senhas, foi utilizado loops que dependem da quantidade de caracteres necessários para saber se precisam ou não gerar mais caracteres. Isso significa que, se for necessário gerar apenas um, apenas um será gerado, caso contrário será gerado dois ou três.

```
public boolean generatePassword(int numbChar, int start, int end) {  
  
    String password = "";  
  
    //Gera uma senha dependendo do número de caracteres necessários  
    for (int i = start; i <= end; i++) {  
  
        //Verifica se a senha já foi encontrada  
        if(FileBreaker.isPasswordBroken) return false;  
  
        //Se a quantidade de caracteres necessários forem 1, é retornado apenas um caracter, se não é gerado mais caracteres  
        if (numbChar == 1) {  
            password = String.valueOf((char) i);  
            if (testPassword(password)) return true;  
        } else {  
            //Valor do caracter inicial da tabela ASCII  
            int startAscii = 33;  
            //Valor do caracter final da tabela ASCII  
            int endAscii = 126;  
  
            for (int j = startAscii; j <= endAscii; j++) {  
  
                //Verifica se a senha já foi encontrada  
                if(FileBreaker.isPasswordBroken) return false;  
  
                //Se a quantidade de caracteres necessários forem 2, é retornado dois caracteres, se não é gerado mais um caracter  
                if (numbChar == 2) {  
                    password = String.valueOf((char) i) + (char) j;  
                    if (testPassword(password)) return true;  
                } else {  
                    for (int k = startAscii; k <= endAscii; k++) {  
  
                        //Verifica se a senha já foi encontrada  
                        if(FileBreaker.isPasswordBroken) return false;  
                        //Gera o último caracter necessários  
                        if (numbChar == 3) {  
                            password = String.valueOf((char) i) + (char) j + (char) k;  
                            if (testPassword(password)) return true;  
                        }  
                    }  
                }  
            }  
        }  
    }  
}  
return false;
```

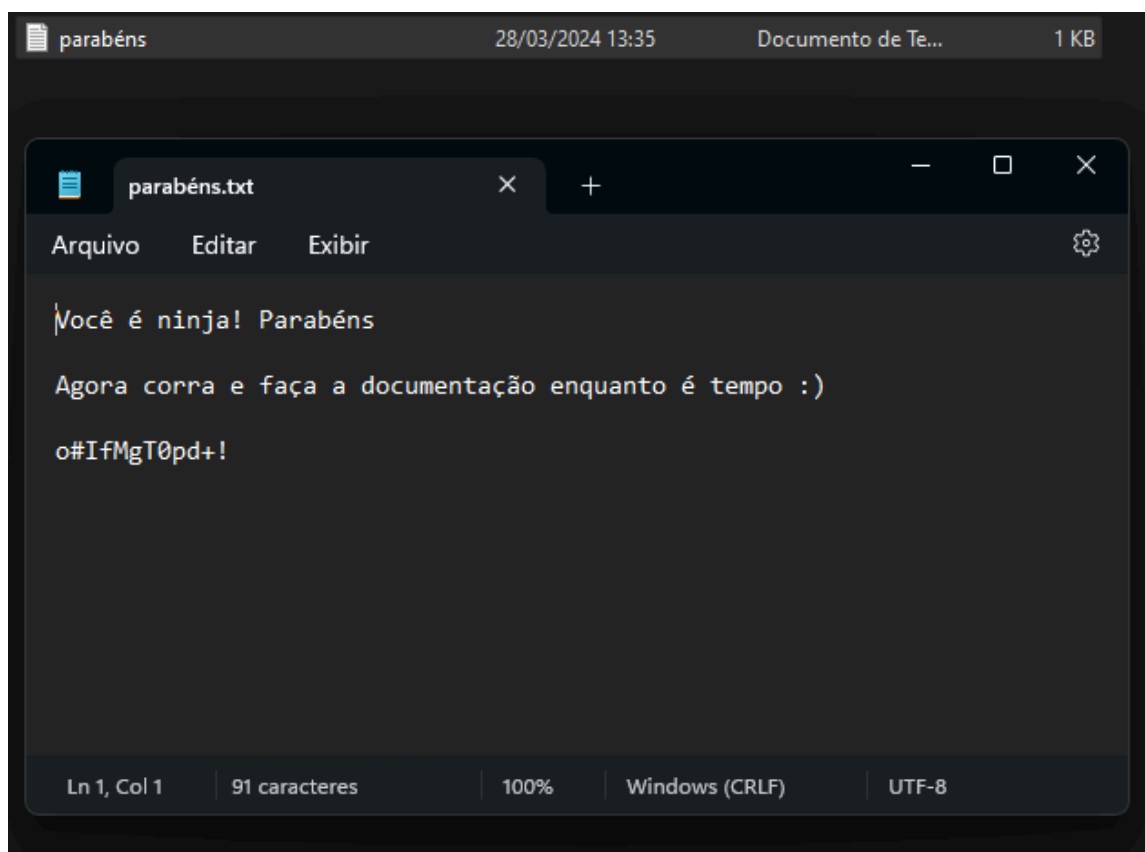
3.2.4 Testando a senha

Depois que a senha é gerada ela é testada pela função que o professor nos disponibilizou.

Caso for encontrada, o arquivo é descompactado no mesmo diretório, o loop de gerar senhas é finalizado e todas as threads passam para o próximo arquivo a ser quebrado.

3.2.5 Descobrindo a senha do arquivo final

Após todos os arquivos serem quebrados e descompactados, o conteúdo dentro deles é concatenado em uma String que é utilizada para descompactar o arquivo final, que contém o seguinte conteúdo:



The screenshot shows a text editor window titled 'parabéns' with a timestamp of '28/03/2024 13:35' and a file size of '1 KB'. The editor has a dark theme and a menu bar with 'Arquivo', 'Editar', and 'Exibir'. The main text area contains the following content:

```
Você é ninja! Parabéns  
Agora corra e faça a documentação enquanto é tempo :)  
o#IfMgT0pd+!
```

The status bar at the bottom indicates 'Ln 1, Col 1', '91 caracteres', '100%', 'Windows (CRLF)', and 'UTF-8'.

4 Conclusão

Em suma, este foi um trabalho difícil porém gratificante. Durante esse processo, enfrentamos desafios que nos proporcionou muito aprendizados.

A maior dificuldade que enfrentamos, foi a implementação da lógica de gerar a senha. Primeiro definimos que cada threads deveria gerar as senhas com o primeiro caractere dentro de um intervalo antes definido. Porém, os demais caracteres a serem gerados não poderiam ser apenas aqueles dentro deste intervalo e sim todos os caracteres da tabela.

Outra dificuldade, neste mesmo projeto, foi a aplicação do processo de paralelização, em que precisávamos de alguma forma indicar para as threads que a senha já foi encontrada e que elas deveriam passar para o próximo arquivo.

Esses desafios nos ajudaram a aprimorar nossas habilidades em programação paralela. A experiência foi valiosa e nos proporcionou uma visão mais profunda sobre a importância da sincronização e do gerenciamento eficiente de threads em projetos de alta complexidade.

5 Referências

O código presente nesta documentação estão disponíveis nos repositórios:

Disponível em: <<https://github.com/rodolfmirand/IFMG-SI/tree/main/Arquitetura%20e%20Organização%20de%20Computadores/CorrigindoImagens>>. Acesso em: 27 jul. 2024.

Disponível em: <<https://github.com/rodolfmirand/IFMG-SI/tree/main/Arquitetura%20e%20Organização%20de%20Computadores/QuebrandoSenhas>>. Acesso em: 27 jul. 2024.