Projeto II - Sistemas Operacionais (TT304)

Grupo: Atox

Integrantes:

- **Leonardo Pardo Davidiuk Gomes - 201253**
- **Henrique Lavieri Facio - 198806**

Objetivo do projeto

Desenvolver um programa que receba dados vindos de um arquivo texto, ordene-os e retorne um arquivo texto com os dados devidamentes organizados. Visando aplicar os conceitos passados no curso, o programa deve utilizar múltiplos threads para ordenar de maneira mais eficaz os dados recebidos durante a execução.

Para que seja possível a execução correta do programa, o usuário deve fornecer a quantidade de dados do arquivo de entrada, o número de threads que deseja utilizar (2, 4, 8, 16) e o nome do arquivo de saída.

Solução utilizada

Utilizamos os múltiplos threads para acelerar a execução do programa. Para alcançar tal objetivo transferimos os dados do arquivo texto de entrada para um vetor de números inteiros.

Após transferir os dados para o vetor, dinamicamente alocado, separamos o vetor em partes menores, de acordo com a quantidade de threads utilizados.

Para ordenar os dados utilizamos o algoritmo rápido de ordenação chamado HeapSort.

Após dividir em diversos vetores menores, executamos o heapsort em cada um deles e depois junta-se os diversos vetores num só e executamos novamente o algoritmo de ordenação escolhido para, finalmente, ordenar todo os dados recebidos do arquivo.

Explicando o código

```
void cria_heap(int *vet, int i, int f){
    int aux = vet[i];
    int j = i*2+1;
    while(j<=f){
        if (j<f){
            if(vet[j]<vet[j+1]){
                j=j+1;
            }
        }
    if (aux < vet[j]){
        vet[i] = vet[j];
        i = j;
        j = 2*i+1;
    }else{
        j=f+1;
    }
    }
    vet[i] = aux;
}

void heapsort (int *vet, int N){
    int i, aux;
    for (i=(N-1)/2; i >= 0; i--){
        cria_heap(vet, i, N-1);
    }
    for (i=N-1; i >= 0; i--){
        aux = vet[0];
        vet[0] = vet [i];
        vet [i] = aux;
        cria_heap(vet, 0, i-1);
    }
}
```

Figura 1 - Função cria_heap e heapsort

A figura 1 envolve as duas funções responsáveis por ordenar o vetor contendo os dados recebidos do arquivo indicado pelo usuário.

A função heapsort recebe como parâmetro um ponteiro para o vetor chamado **vet* e o tamanho do mesmo, ou seja, a quantidade de dados presentes no vetor, representado por N.

```
//Passa os dados do arquivo de origem para um vetor dinamicamente alocado
FILE * pFile;
char mystring [100];
int cont = 0;

pFile = fopen (nomeArqIni , "r");

while ( fgets (mystring , 100 , pFile) != NULL ){
    vetor[cont] = atoi(mystring);
    cont++;
}

fclose (pFile);
```

Figura 2 - Passagem dos dados do arquivo para um vetor

A figura 2 representa o trecho do código responsável pela passagem dos dados do arquivo de origem para um vetor dinamicamente alocado.

O trecho acima utiliza a função fgets para ler, linha por linha, o arquivo texto de entrada e, pela função atoi, transformar a string lida em um número inteiro e armazená-lo no vetor dinâmico.

```
    //Cria a quantidade de threads recebido do usuario
    for (y = 0; y < NUM_THREADS; y++) {
            argthread = malloc(sizeof(argumento));
            argthread->vet_aux
=malloc(sizeof(argumento)*argthread->intervalo);
            argthread->inicio = 0;
            argthread->intervalo = tamanhoVetor/NUM_THREADS;
            argthread->fim = argthread->inicio + argthread->intervalo;
            argthread->y = y;
            pthread_create (&threads[y], NULL, threadX, (void *)
argthread);  //(trhead criado,null,função,parametro)
            argthread->fim = argthread->fim + argthread->intervalo;

}
    //Impede o programa de terminar até finalizar todas as threads
        for (y = 0; y < NUM_THREADS; y++) {
            pthread_join (threads[y], NULL);
}
```

Figura 3 - Criação das threads

A figura acima cria a quantidade de threads indicada pelo usuário, sendo que o vetor auxiliar utilizado para dividir o vetor principal também é alocado dinamicamente. Além disso declara-se o início, fim, e um intervalo, relativos às threads.

Para que não haja erros de execução, utiliza-se o join das threads declaradas, para que o programa não termine sua execução antes de finalizar as threads inicializadas anteriormente.

Após esse passo, será chamada a função heapsort, que, agora, ordena os valores anteriormente pré-ordenados pelo algoritmo de HeapSort.

```
FILE *fp;
fp = fopen (nomeArqFinal, "w+");

for (p = 0; p < tamanhoVetor; p++){
fprintf(fp, "%d\n", vetor[p]);
}
fclose(fp);
      pthread_mutex_destroy (&shallnotpass);
```

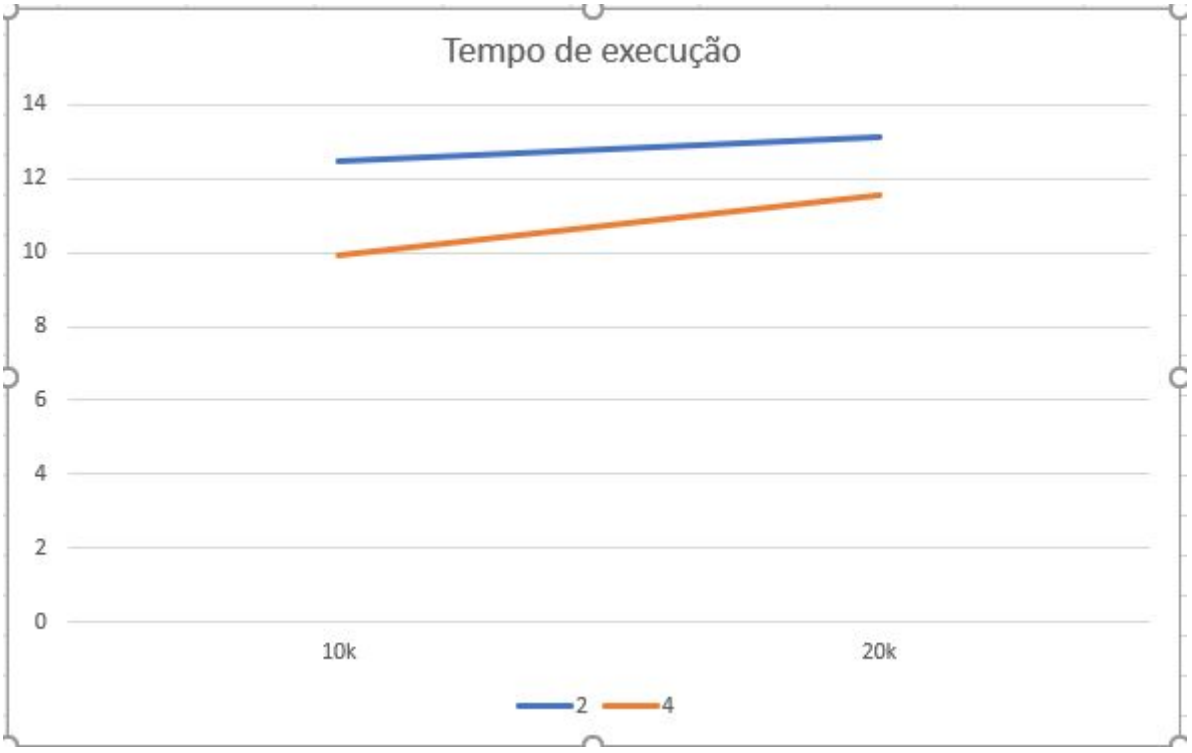
Figura 4 - Imprimir os valores ordenados num arquivo texto final

A figura 4 mostra o trecho responsável por imprimir num arquivo texto - de nome fornecido pelo usuário - os valores fornecidos pelo arquivo texto de entrada em ordem crescente, resultado do algoritmo de ordenação HeapSort.

Dificuldades

O grupo sofreu problemas com a alocação do vetor. Dessa forma o programa roda, pelo menos nos computadores em que testamos, com até 40000 dados, mais que isso o programa não gera arquivo final algum.

Não conseguimos reverter essa situação.Procuramos possíveis caminhos para a alocação dinâmica, mas, infelizmente, nada conclusivo, utilizamos também mais de uma forma de ler os dados do arquivo e passar para o vetor, mas nada mudou também.



azul = 2 threads laranja = 4 threads, ambos para N = 10000 e N = 20000.

Link para o repositório Git:

- https://github.com/henriq88/Trabson-SO