Relatório de Sistemas Operacionais

Leonardo Sanchez Apolinário Ra: 220173
Isaque Luiz da Costa Ra: 218315

Projeto 1

Descrição do problema:

Considere que há vários (n ≥ 1) arquivos, cada um deles com uma certa quantidade de números inteiros sem ordem estabelecida. O programa deverá ler os dados de todos esses n arquivos, organizá-los em ordem crescente e armazenar em um único arquivo resultante.

Toda essa operação – leitura dos arquivos, ordenação e concatenação dos dados – deve ocorrer utilizando 2, 4, 8 ou 16 threads, a critério do usuário. O programa deverá ser escrito para o sistema operacional Linux e obrigatoriamente utilizar a biblioteca POSIX Threads.

Código fonte disponível em:

https://github.com/XleonardoX/ProjetoSO2018/blob/master/multicat.c

Instruções para compilação:

gcc -pthread multicat.c -o multicat

Link para o vídeo:

https://www.youtube.com/watch?v=kwfjC4QbNTs

Descrição da solução do problema:

Basicamente para resolver o problema desse projeto decidimos separá-lo em partes. Essencialmente existem três funções a serem realizadas, São elas: Leitura dos arquivos de entrada, Ordenação dos números encontrados nos arquivos e salvar esses inteiros devidamente ordenados num arquivo de saída. Nós decidimos fazer essas funções separadamente de forma modularizada, assim pudemos desenvolver o programa por partes e realizando diversas verificações para certificar que tudo corria como esperado. Além disso, a modularização do programa nos permitiu controlar com maior precisão o uso de threads chamando-as apenas para as partes do programa que era necessário maior processamento e não para o programa como um todo.

Leitura dos arquivos de entrada:

Descrevendo melhor essas funções, começamos pela função que carrega os dados dos arquivos de entrada. Basicamente criamos uma estrutura global (multicat) que contem o um apontador para inteiros (int *v). A estrutura guardava também os nomes dos arquivos de entrada e de saída num vetor de strings (arquivos) que recebe os nomes com base na própria linha de comando do terminal (char *argv) e um inteiro auxiliar para a alocação do vetor (int top).

Tendo guardado os nomes dos arquivos de entrada chamamos a função que Lê esses arquivos (no programa a função chama-se “Multicat_integer”) e recebe como parâmetro um apontador para a estrutura. Criamos um laço de repetição para abrir para leitura todos os arquivos presentes no vetor de strings (com exceção do arquivo de saída). Primeiro abrimos para leitura para saber quantos números inteiros haviam no total em todos os arquivos de entrada e guardar essa quantidade numa variável da estrutura (int top) par poder alocar o tamanho correto do vetor (int *v) e guardar todos os números.

Depois usando a mesma lógica na estrutura de laços anteriormente citada, abrimos os arquivos novamente para leitura, porem dessa vez atribuindo os números diretamente ao vetor já alocado (os números são copiados para o vetor na ordem que são encontrados nos arquivos, portanto após essa função os números continuam desordenados, mas agora n vetor e não nos arquivos), e com isso encerramos essa função.

Ordenação:

Passando para a função de ordenação do vetor anteriormente construído, é nessa função que utilizamos as threads, dado que é a função que demanda mais processamento. Antes de chamar a função foi necessário saber quantas threads deveriam ser utilizadas, a critério do usuário. Isso foi resolvido, assim como nos arquivos, pegando esse argumento diretamente da linha de comando, na qual a quantidade de threads estava referida no segundo argumento passado, e com esse numero salvo criamos um vetor de threads para a função de ordenação. Dado o fato de que nesse programa apenas o processamento usando threads deve ser medido, nós começamos medir um pouco antes da chamada da função de ordenação e terminamos a medição imediatamente depois do final dessa função.

Como nessa função é utilizado múltiplas threads sua chamada de execução é diferente da usual, ela deve receber como parâmetro final um apontador para vazio (pthread_create(&threads[i],NULL,Ordena,(void *)x);) que no nosso caso é um apontador para a estrutura onde esta o vetor que queremos ordenar. Para criar as threads de ordenação foi desenvolvido um laço de repetição que criava exatamente a quantia de threads que o usuário passou na linha de comando, chamando a função para executar (com os cuidados necessários para bloquear as threads ao final da ordenação (pthread_join)).

Quanto ao algoritmo utilizado na função, optamos pelo clássico Bubble sort. Apesar de não ser tão rápido, sua escolha foi proposital justamente pelo fato desse algoritmo nos dar uma maior margem de medição, de modo a diferenciar melhor o tempo de processamento para diferentes quantidades de threads. Ao final dessa função temos um vetor ordenado e pronto pra ser salvo no arquivo de saída.

Salvar no arquivo de Saída:

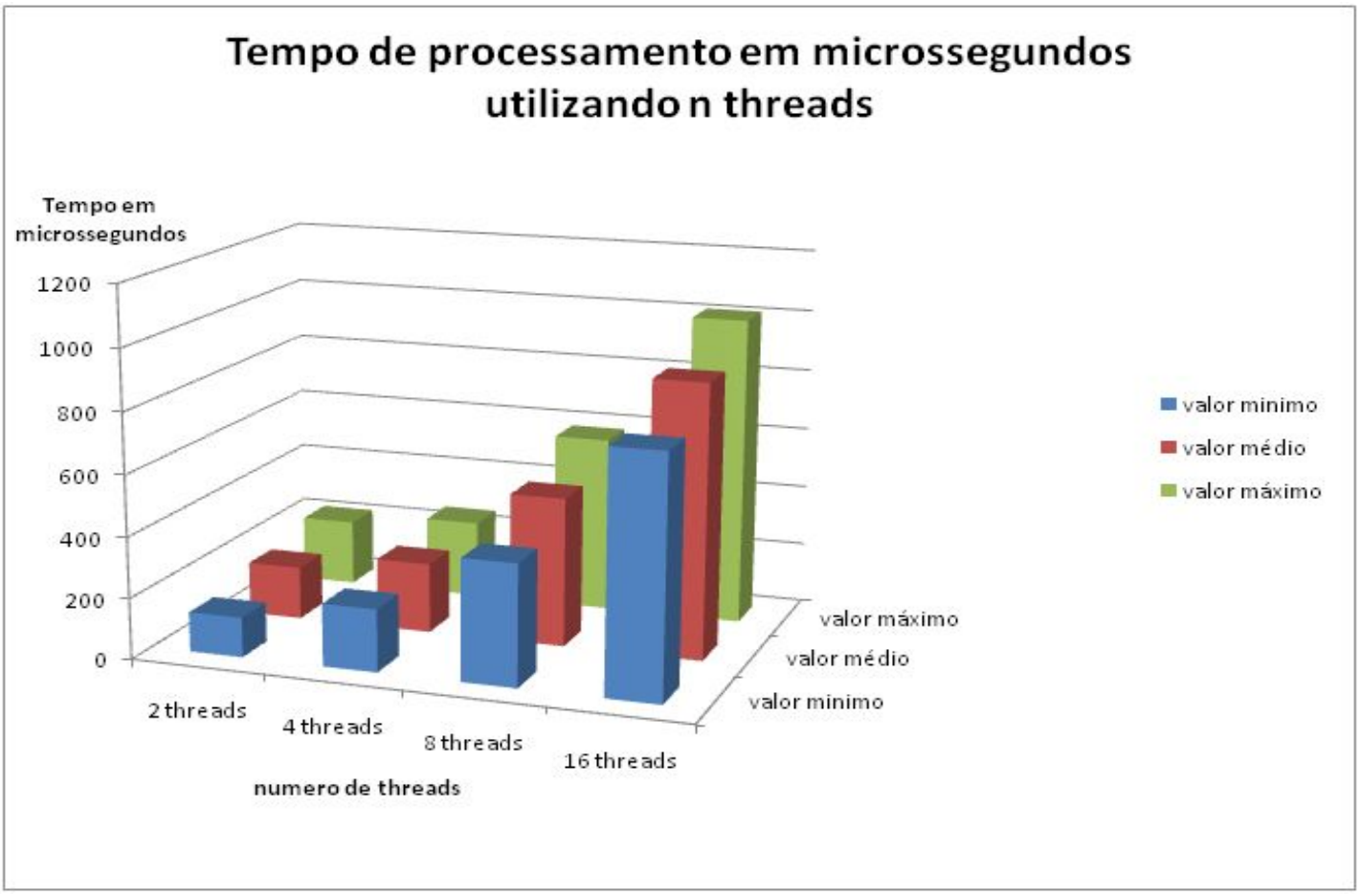
A função mais simples do programa é a de salvar no arquivo de saída. Ela basicamente abre o arquivo de saída, que está na última posição do vetor de strings da estrutura global, para escrita (w+) de modo que se o arquivo não existir ele é gerado automaticamente, e se ele existir o arquivo é limpo e escrito do início. Com o salvamento de todos os inteiros ordenados do vetor a função termina.

Com tudo feito é retornado ao final do programa o tempo de processamento utilizando n threads.

Observações:

Inicialmente usamos as threads para leitura dos arquivos porém nos foi aconselhado pelo monitor que esse uso era ineficiente, já que ocorreriam mais acessos ao disco e portanto mais tempo de processamento.

Gráficos e conclusões:



Concluímos através do gráfico que uma quantidade alta de threads torna o processo mais lento, uma vez que quanto mais threads um processo tiver maior a quantidade de trocas de contexto (promovidas pelo escalonador) entre essas threads e, portanto maior o tempo de processamento.

Para valores baixos com 2 ou 4 threads vemos que o tempo de processamento é relativamente pequeno e próximo. Porém quando subimos esse valor para 8 ou 16 threads o tempo de processamento sobe muito ao ponto de seu uso ser ineficiente já que os escalonador perde muito tempo fazendo as trocas de contexto aumentando o tempo de processamento.