UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES CIÊNCIAS E HUMANIDADES

ACH2044 - Sistemas Operacionais

Exercício programa 2
2024

Objetivos

Ordenação é um problema clássico em computação e torná-la mais eficiente é sempre um bom desafio. Neste exercício, você implementará um algoritmo de ordenação paralela de alto desempenho.

Há três objetivos principais para esta exercício programa:

- Familiarizar-se com os pthreads do Linux.
- Aprender a paralelizar um programa.
- Aferir o desempenho do programa.

Para implementar este exercício, você usará conceitos relacionados à concorrência, como criação de threads, exclusão mútua e sinalização (com variáveis de condição). Estes tópicos são tratados nas quatro primeiras aulas sobre concorrência.

Especificação

Você deve implementar um programa em linguagem C contido em um único arquivo com nome `psort<numero usp>.c` (por exemplo: `psort5894150.c`). O programa recebe três argumentos de linha de comando:

```
$ ./psort<nusp> <entrada> <saída> <threads>
```

O parâmetro `<entrada>` é o nome de um arquivo que contém os registros a serem ordenados. Cada registro tem o tamanho fixo de 100 bytes. Cada registro é composto por uma chave representada por um inteiro de 32 bits (4 bytes) seguidos por 96 bytes de dados. Os registros devem ser ordenados de acordo com as chaves apenas.

Note que será considerado um sistema Little Endian, ou seja, o número `int chave = 1` fica armazenado no arquivo como 0x01 0x00 0x00 0x00. A maioria dos processadores são Little Endian, então provavelmente você não precisará se preocupar com isso!

O programa deve ler todos os registros do arquivo de entrada, colocá-los na memória, classificá-los e, por fim, escrever o resultado da ordenação no arquivo indicado pelo parâmetro `<saída>`. Você deve forçar gravações no disco chamando `fsync()` ou `msync()` no arquivo de saída antes de terminar a execução do programa.

O parâmetro `<threads>` deve ser usado para determinar a quantidade máxima de threads adicionais que o programa usará. Se o parâmetro for igual a 0, o programa deve decidir quantas threads usar.

Você pode assumir que esta é uma ordenação de uma passada, ou seja, todos os dados cabem na memória. Você não precisa implementar uma ordenação de várias passadas.

Dicas

Seguem algumas dicas de coisas que você precisará pensar sobre e/ou resolver para implementar o exercício:

1. Como paralelizar o processo de ordenação? Este é o desafio central deste projeto. Pense no que pode ser feito em paralelo e no que deve ser feito em série por uma única thread e projete sua ordenação paralela conforme apropriado.
2. Um aspecto interessante (e completamente opcional para o EP) que uma implementação bastante eficiente vai levar em conta é o seguinte: o que acontece se uma thread acabar sendo mais lenta que as outras? Deve-se dar mais trabalho para as threads mais rápidas?
3. Como determinar quantas threads criar? No Linux, é possível usar funções como `get_nprocs()` e `get_nprocs_conf()` para determinar a quantidade de processadores disponíveis no sistema (e basear a quantidade de threads usadas no valor obtido). Leia as páginas do manual (com `man get_nprocs`) para saber mais detalhes.
4. Como executar cada parte do trabalho de forma eficiente? Embora a paralelização produza velocidade, a eficiência de cada thread na execução da ordenação também é de importância crítica.
5. Como acessar os arquivos de entrada/saída de forma eficiente? No Linux, há muitas maneiras de ler um arquivo, incluindo funções da biblioteca padrão C como o `fread()` e funções de acesso de baixo nível como o `read()`. Uma maneira particularmente eficiente é usar arquivos mapeados na memória, criados através da função `mmap()`. Ao mapear o arquivo de entrada no espaço de endereçamento do processo, você pode acessar os bytes do arquivo de entrada via ponteiro (como se fosse um vetor) de forma eficiente. Analogamente, a forma como você escreve o arquivo de saída e, talvez, como você intercala a escrita com a ordenação, pode fazer a ordenação rodar mais rápido.

Entrega

Você deverá entregar um relatório (nomeado como `<número usp>.pdf`, por exemplo, 5894150.pdf) contendo duas seções:

1) uma seção com as decisões de projeto que você tomou; e
2) uma seção com uma análise de desempenho do seu programa

A análise de desempenho deve relacionar o tempo de execução total do seu programa com o número de threads adicionais utilizadas (ou seja, variando o parâmetro `<threads>`). Varie a quantidade de threads adicionais de 1 até 8. Faça a análise para pelo menos dois

tamanhos de arquivo de entrada diferentes (um “grande” e um “pequeno”). Lembre-se de interpretar os resultados!

Documente o método de medição utilizado e não esqueça de apresentar resultados com significância estatística, isto é, para cada par (número de threads, tamanho do arquivo de entrada) faça a medição de tempo várias vezes e calcule a média e o intervalo de confiança. É recomendado automatizar a execução (por exemplo, com um script).

Além do relatório, deve ser entregue um único arquivo .c com nome `psort<numero usp>.c` (por exemplo, psort5894150.c).

Para os testes, seu código será ser compilado com as seguintes flags de compilação:

```
-Wall -Werror -pthread -std=c11 -O
```

A última flag diz para o compilador usar o otimizador. Você pode experimentar medir o tempo de execução do seu código com e sem -O e para ver o tamanho do impacto no desempenho.

Correção

Seu programa será primeiro testado quanto à corretude, ou seja, será verificado se ele é capaz de ordenar alguns arquivos de entrada corretamente. Serão testados arquivos de entrada já em ordem, arquivos em ordem reversa e arquivos com chaves aleatórias, cada um em três tamanhos diferentes (aproximadamente 1KB, 1MB e 100 MB).

Exemplos de arquivo de entrada e as respectivas saídas esperadas serão fornecidos. Também será fornecido o `ep_input_generator.c` para que você possa gerar outras entradas do tamanho que quiser.

Se o seu programa passar em todos os testes, seu código será testado quanto ao desempenho utilizando o comando `time` (e com prioridade aumentada com `nice` -19). Será avaliado o tempo total de execução para ordenar um arquivo grande com chaves aleatórias, passando-se `0` como número de threads.

O programa mais rápido será proclamado o **"ordenador mais rápido de todos da sala**[1]**"**!

Como de costume, os arquivos submetidos serão verificados por plágio, acarretando nota 0 se for constatado. É permitido aos alunos discutirem estratégias de resolução, porém a implementação deve ser feita individualmente[2].

[1] E talvez ganhe algo?
[2] Afinal, encontrar bugs em programas paralelos é parte do aprendizado :)