Trabalho Prático II - AEDS III
Índice Invertido

Ingrid Elisabeth Spangler 11/06/2017

Departamento de Ciência da Computação - UFMG

Resumo: *Esta documentação descreve um algoritmo de processamento de texto para criação de um índice invertido ordenado lexicograficamente em arquivos grandes demais para a RAM dos dispositivos em que eles se encontram. O algoritmo faz uso do quicksort externo e interno, e insertion sort, de acordo com a proposta do que foi ensinado na disciplina Algoritmos e Estruturas de Dados III de 2017/1.*

1. Introdução

Busca por palavras é um recurso muito útil para aplicativos de trocas de mensagens, e um índice invertido proporciona a maneira mais ágil de se realizar esta tarefa em textos particularmente grandes, para que esta seja possível é necessário um pré-processamento que consiste em mapear todas as palavras no formato **Palavra documento posição frequência**:

Palavra,1,30,10

E depois ordená-las lexicograficamente para que a busca funcione como uma consulta a um índice remissivo. Esta tarefa pode consumir uma porção considerável de RAM se realizada de uma vez só, principalmente em dispositivos mobile, como é o caso do problema resolvido por meio deste trabalho.

2. Solução

O programa requer uma estratégia que faz uso da memória secundária (disco) como auxiliar na criação do índice. Primeiramente, é feito o mapeamento das palavras, cada conversa é aberta e seus dados sao lidos palavra por palavra para uma struct Register e escrita em um arquivo binário, esta etapa não apresenta nenhuma necessidade desta otimização, pois é feito sequencialmente, sendo o verdadeiro problema a ordenação do arquivo com os índices criado. Cada registro a ser comparado ocupa 32 bytes de espaço interno, então apenas M/32 registros podem ser carregados para comparação por vez na etapa de ordenação, este valor

é computado em uma variável, três arquivos auxiliares são gerados no decorrer do procedimento: indexed, ordered e final. Usarei estes arquivos para delimitar as etapas:

- Indexação: As palavras são lidas dos arquivos de conversas de entrada uma por uma, convertidas em uma struct com as informações sobre seu documento de origem e sua posição, na ordem em que aparecem. Estes registros são escritos em blocos contíguos de 32 bytes em um arquivo binário chamado “indexed”.

- Ordenação: Os registros do arquivo binário são lidos para a memória do programa, M/32 registros por vez, de acordo com o método de leitura do quicksort externo.

- Contagem de frequência: Uma leitura sequencial dos registros é feita, e o número de palavras em cada conjunto de palavras iguais em documentos iguais no arquivo “Ordered” é calculado até que se encontre uma palavra diferente. Quando isso acontece, outra leitura é feita para substituir o campo frequência de cada palavra pela frequência computada e escrever no formato de saída. A saída deste procedimento é o arquivo em texto pleno “final”

3. Decisões de Implementação

Um modelo de arquivo binário foi utilizado para os índices auxiliares ao invés de texto pleno pela sua propriedade de regularidade de bytes, que proporciona praticidade da transferência de dados entre o arquivo e o programa em C.

Imagem: Bytes de um registro no arquivo binário

Os métodos de ordenação escolhidos para funcionar juntamente com o quicksort externo foram o quicksort interno para ordenar o primeiro buffer e o insertion sort para ordenar as demais iterações. Estas escolhas foram motivadas pela rapidez em cada situação. O quicksort é o método mais rápido para o caso médio, e o insertion sort é o método mais rápido para arquivos parcialmente ordenados, como é o caso do buffer após realizar apenas uma troca de registro.

O algoritmo da contagem de frequência passa por cada conjunto de palavras do arquivo duas vezes. A primeira para contagem e a segunda para atualização.

4. Análise de complexidade

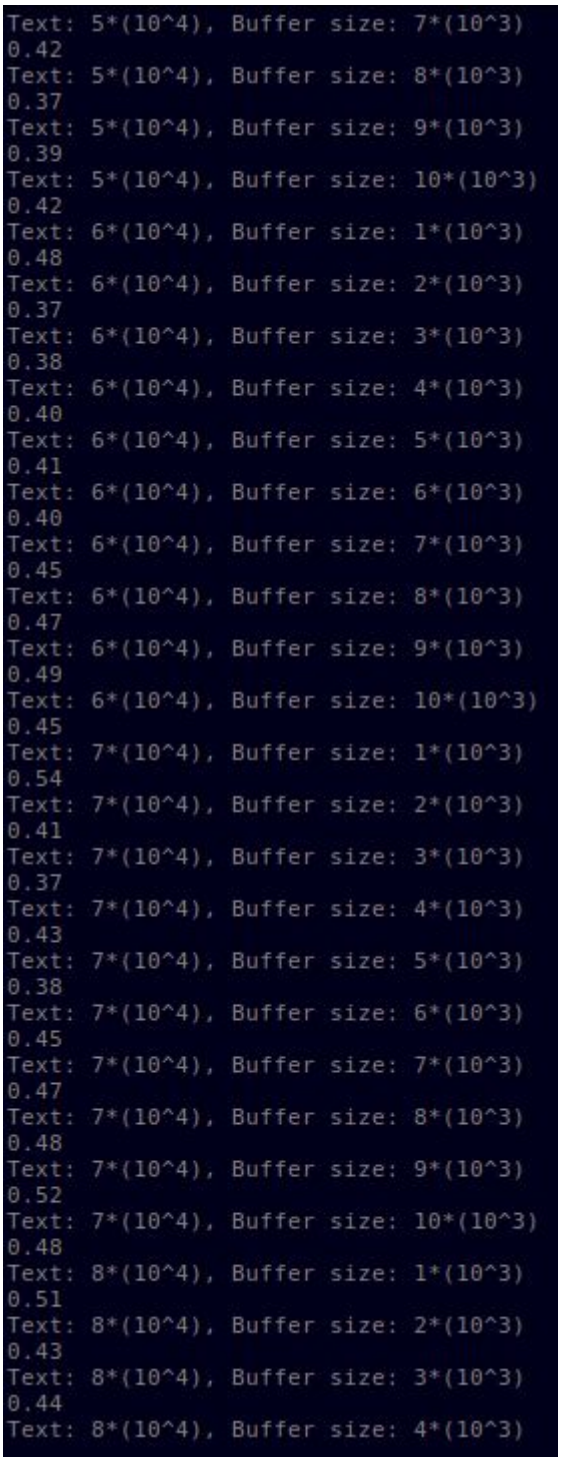
Análise das funções mais significativas do código:

Função	Tempo	Espaço
Add_frequency()	$O(n^2)$	O(1)
make_index()	O(n)	O(n)
PartialSort()	O(n)	O(n)
Sort()	O(n log n)	O(n)
QuickSortExt	O(n log n)	O(n)

5. Testes

Foram realizados testes com textos e memórias internas de tamanhos variados. Os testes foram executados 10 vezes cada um em um processador Intel Core i7 - 3537U 2.0GHz. Os resultados apresentados são uma amostra dos valores de um teste.

```
Text: 1*(10^4), Buffer size: 1*(10^3)
0.27
Text: 1*(10^4), Buffer size: 2*(10^3)
0.27
Text: 1*(10^4), Buffer size: 3*(10^3)
0.21
Text: 1*(10^4), Buffer size: 4*(10^3)
0.28
Text: 1*(10^4), Buffer size: 5*(10^3)
0.28
Text: 1*(10^4), Buffer size: 6*(10^3)
0.31
Text: 1*(10^4), Buffer size: 7*(10^3)
0.24
Text: 1*(10^4), Buffer size: 8*(10^3)
0.32
Text: 1*(10^4), Buffer size: 9*(10^3)
0.33
Text: 1*(10^4), Buffer size: 10*(10^3)
0.27
Text: 2*(10^4), Buffer size: 1*(10^3)
0.32
Text: 2*(10^4), Buffer size: 2*(10^3)
0.24
Text: 2*(10^4), Buffer size: 3*(10^3)
0.28
Text: 2*(10^4), Buffer size: 4*(10^3)
0.30
Text: 2*(10^4), Buffer size: 5*(10^3)
0.32
Text: 2*(10^4), Buffer size: 6*(10^3)
0.32
Text: 2*(10^4), Buffer size: 7*(10^3)
0.31
Text: 2*(10^4), Buffer size: 8*(10^3)
0.32
Text: 2*(10^4), Buffer size: 9*(10^3)
0.38
Text: 2*(10^4), Buffer size: 10*(10^3)
0.30
Text: 3*(10^4), Buffer size: 1*(10^3)
0.40
Text: 3*(10^4), Buffer size: 2*(10^3)
0.33
Text: 3*(10^4), Buffer size: 3*(10^3)
0.25
Text: 3*(10^4), Buffer size: 4*(10^3)
0.27
Text: 3*(10^4), Buffer size: 5*(10^3)
0.32
Text: 3*(10^4), Buffer size: 6*(10^3)
0.28
Text: 3*(10^4), Buffer size: 7*(10^3)
0.37
Text: 3*(10^4), Buffer size: 8*(10^3)
0.38
Text: 3*(10^4), Buffer size: 9*(10^3)
0.37
Text: 3*(10^4), Buffer size: 10*(10^3)
0.35
Text: 4*(10^4), Buffer size: 1*(10^3)
0.42
Text: 4*(10^4), Buffer size: 2*(10^3)
0.31
Text: 4*(10^4), Buffer size: 3*(10^3)
0.35
Text: 4*(10^4), Buffer size: 4*(10^3)
0.35
Text: 4*(10^4), Buffer size: 5*(10^3)
0.37
Text: 4*(10^4), Buffer size: 6*(10^3)
0.37
Text: 4*(10^4), Buffer size: 7*(10^3)
0.38
Text: 4*(10^4), Buffer size: 8*(10^3)
0.41
Text: 4*(10^4), Buffer size: 9*(10^3)
0.36
Text: 4*(10^4), Buffer size: 10*(10^3)
0.37
Text: 5*(10^4), Buffer size: 1*(10^3)
0.41
Text: 5*(10^4), Buffer size: 2*(10^3)
0.31
Text: 5*(10^4), Buffer size: 3*(10^3)
0.37
Text: 5*(10^4), Buffer size: 4*(10^3)
0.36
Text: 5*(10^4), Buffer size: 5*(10^3)
0.40
Text: 5*(10^4), Buffer size: 6*(10^3)
0.40
Text: 5*(10^4), Buffer size: 7*(10^3)
```



Os testes mostraram que o custo cresce proporcionalmente ao tamanho do texto em palavras porém apresentam comportamento decrescente à medida que o tamanho do buffer aumenta, devido ao menor número de passos de leitura e escrita que o quicksort externo precisa fazer quanto maior o buffer interno.

Após certo tamanho ótimo de buffer o custo cresce de volta. Este crescimento se deve ao crescimento da complexidade dos algoritmos de ordenação, quanto mais registros em uma página para ordenar, maior o custo.

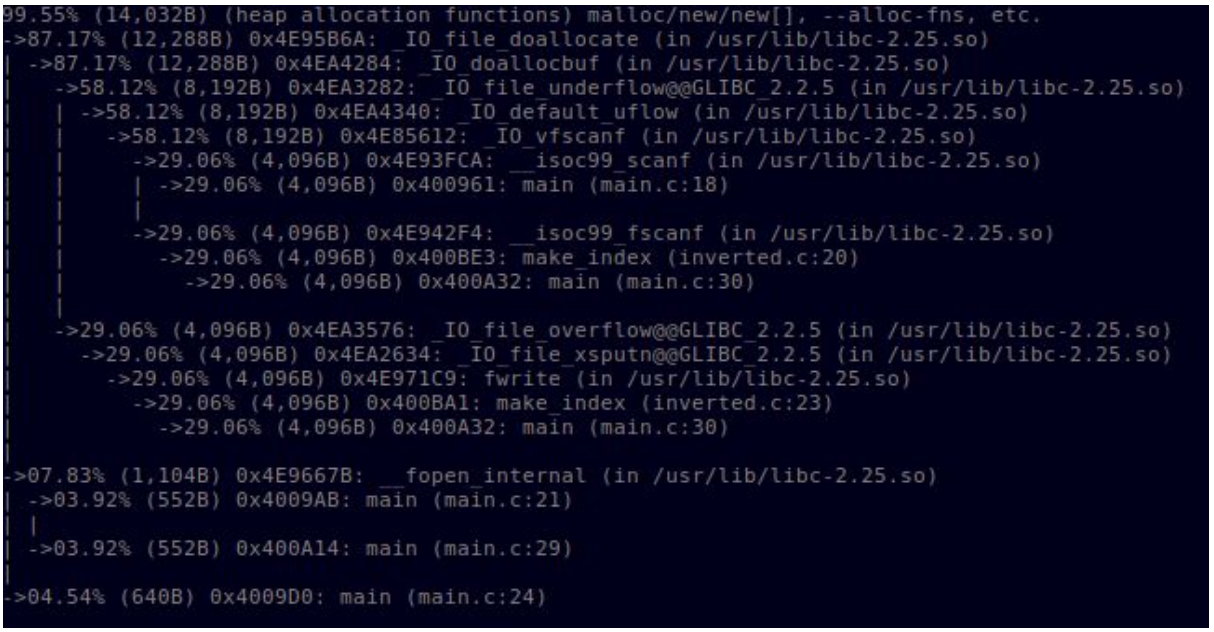


Imagem: análise de alocação de memória da ferramenta massif do valgrind

Para a análise de complexidade de espaço, foi utilizada a ferramenta massif. Como exemplo, que pode ser lido na captura de tela acima, para o teste toy nº 9, a função main aloca 640 bytes para a estrutura interna Registro (main.c:24), respeitando o input dado. As linhas acima (main.c:29 e main.c:21) são referentes à alocação de espaço para os arquivos, cujo tamanho é definido pelo tipo FILE* da biblioteca padrão. Os gastos das chamadas de alocação dependem das bibliotecas nativas de C GLIBC_2.2.5 e do sistema operacional e devem ser ignorados. A captura mostra a análise total de alocação no heap.

6. Conclusão

Este trabalho mostrou que, apesar de ordenação externa ser útil, ainda é bem custosa e difícil de ser implementada corretamente. Além disso, para uma boa otimização do processo, às vezes utilizar buffers menores do que a capacidade total da memória é a melhor opção dada a variação das complexidades combinadas dos dois algoritmos.