Trabalho Prático 1: Ordenação de Arquivos Grandes de Registros

Prof. Dr. Juliano Henrique Foleis

Introdução

Neste trabalho vamos explorar uma solução para ordenar arquivos grandes, onde cada elemento é um registro. Neste contexto nem sempre o arquivo pode ser carregado totalmente em memória primária (RAM), o que inviabiliza a utilização de alguns algoritmos de ordenação que estudamos. Note que não necessariamente "não poder ser carregado" quer dizer que o computador não tem RAM nem para carregar o arquivo. Por exemplo, é comum um servidor de banco de dados ter que atender muitas requisições ao mesmo tempo, cada uma necessitando de uma ordenação. Isto faz com que o programa tenha uma restrição na quantidade de memória usada para atender cada requisição.

Suponha que o arquivo que você quer ordenar é o X, um arquivo binário que consiste em uma sequência de registros do tipo ITEM_VENDA, definido a seguir:

```
typedef struct ITEM_VENDA{
    uint32_t id;
    uint32_t id_venda;
    uint32_t data;
    float desconto;
    char obs[1008];
}ITEM_VENDA;
```

Deseja-se ordenar o arquivo pelo campo *id*. Suponha também que você tem 100MB de RAM disponível para a ordenação. Como *sizeof(ITEM_VENDA)* = 1024, é possível armazenar 102400 registros na memória. Suponha que você consiga dividir o arquivo X em K pedaços $(P1, P2, \ldots, PK)$ de aproximadamente 100MB cada. Desta forma, cada um desses pedaços pode ser ordenado diretamente em memória principal, usando qualquer um dos métodos estudados. Como podemos "juntar" esses K pedaços em um arquivo final Y, ordenado?

Na semana 1 nós estudamos o algoritmo de ordenação por intercalação (MergeSort). Este algoritmo utiliza a operação de intercalação em 2 vias, que recebe como entrada 2 vetores ordenados, e retorna um vetor ordenado, contendo todos os elementos dos 2 vetores de entrada. No contexto da ordenação que estamos discutindo, podemos usar o mesmo raciocínio da intercalação de 2 vetores, mas desta vez usando k vetores. Este algoritmo de intercalação por K vias é bem conhecido, e é comumente chamado de k-way Merge. Essa estratégia é conhecida como Ordenação Externa com Intercalação em K-vias. A Figura 1 mostra um exemplo de ordenação com essa estratégia.

Existem muitos algoritmos de intercalação por k-vias. Um dos mais simples de implementar consiste na conversão direta do algoritmo Merge que discutimos na semana 1, buscando o menor elemento em k vetores, ao invés de apenas 2. Existe um último obstáculo para a implementação do k-way merge para gerar o arquivo Y. Como os K pedaços juntos possuem mais de 100MB (na realidade cada um possui aproximadamente 100MB, como discutido acima), não é possível que o algoritmo k-way merge receba os pedaços inteiros! Portanto uma estratégia para não estourar o limite de 100MB para guardar os registros em memória é manter apenas uma proporção dos menores elementos de cada um dos K pedaços em buffers de entrada. A Figura 2 mostra a idéia geral de como isso pode ser feito.

No exemplo, 9 buffers de entrada são alocados, cada um com 10MB de dados. Inicialmente, cada buffer contém os 10MB iniciais de cada pedaço P correspondente. Similar ao Merge estudado na semana 1, o menor

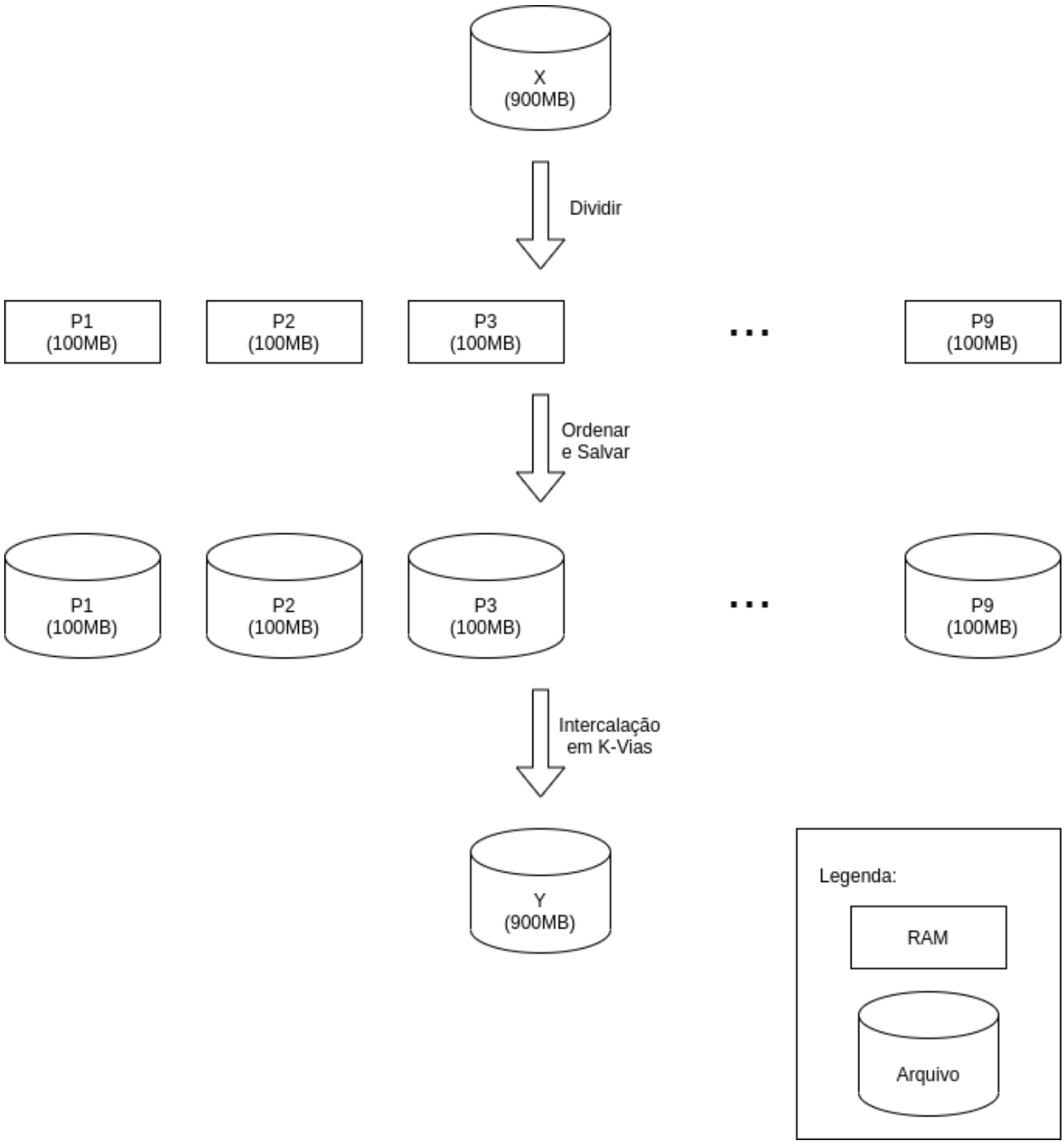

Figure 1: Ordenação Externa com Intercalação em K-vias

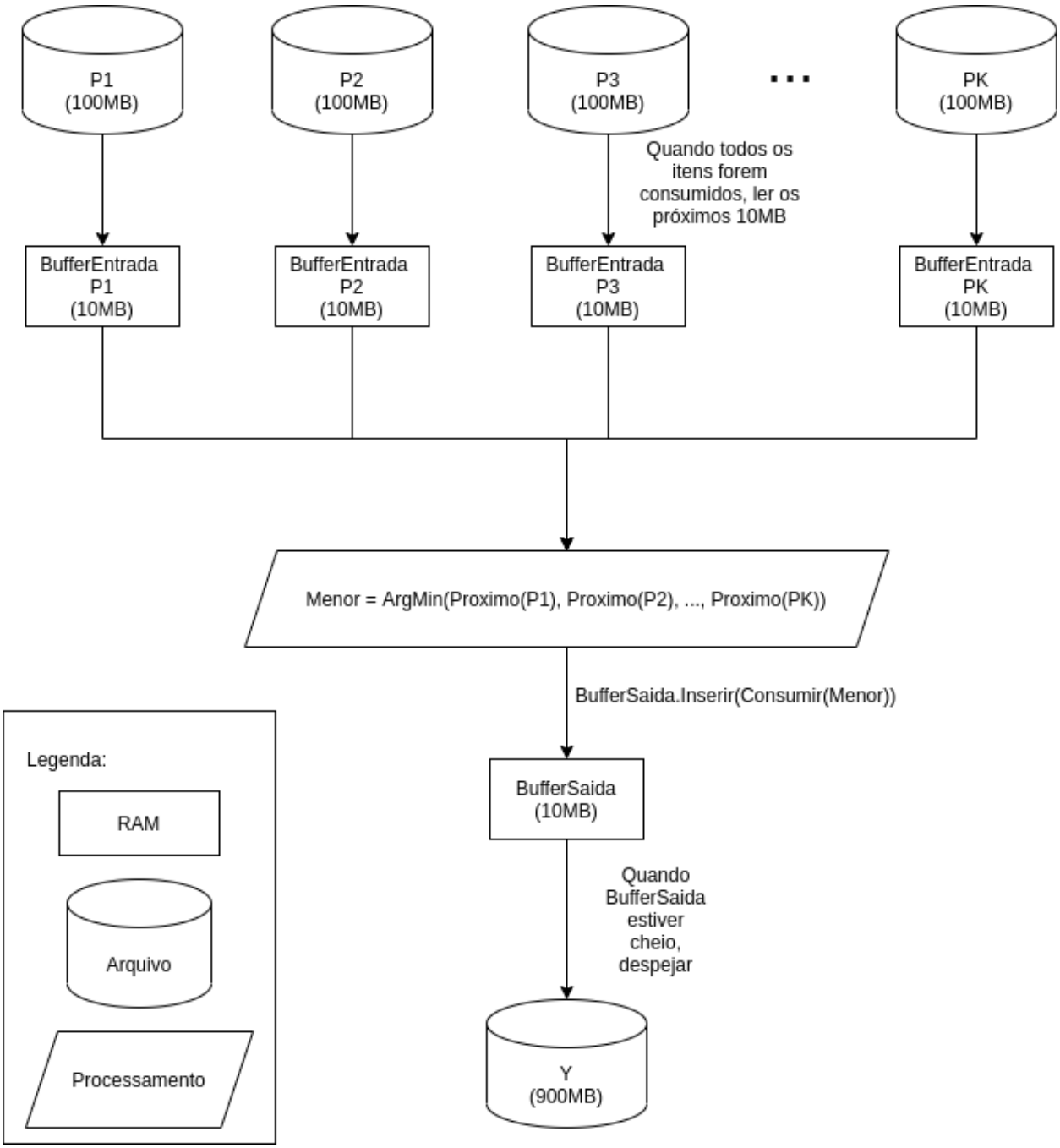

Figure 2: Intercalação em K-vias usando Buffers de Entrada e de Saída

elemento entre os K buffers é determinado comparando a chave de ordenação registro "atual" entre todos os K buffers. Assim que o último elemento de um buffer de entrada é "consumido" (ou seja, é o menor entre todos), os próximos 10MB de registros são lidos a partir do arquivo correspondente. A operação de intercalação em k-vias acaba quando todos os arquivos e buffers de entrada são esgotados.

A Figura 2 também mostra um buffer de saída. A idéia é que os registros vão sendo copiados para o buffer de saída, e são despejados (*dump*) no arquivo de saída *apenas* quando o buffer se torna cheio. Essa escrita no arquivo de saída "em lote", ou seja, de vários registros de uma vez, é mais eficiente do que a escrita de um único registro de cada vez. Isso acontece porque cada *fwrite* gera chamadas de sistema, o que interrompe a execução do programa. A idéia é fazer menos chamadas de sistema para minimizar o tempo que o programa fica interrompido. Além de ser uma abstração que facilita a implementação do algoritmo de intercalação em k-vias, os buffers de entrada também servem para minimizar a quantidade de acessos a disco, reduzindo o número de chamadas de sistema e, consequentemente, minimizando o tempo que o programa fica interrompido.

Especificação

Você deve implementar a rotina de ordenação de arquivos grandes de registros mostrada na Figura 1, com intercalação em K-vias usando buffers de entrada e buffers de saída mostrada na Figura 2. Os arquivos a serem ordenados são sequências de registros do tipo *ITEM_VENDA* discutido anteriormente.

Um programa para a geração procedural de um arquivo de entrada é dado. Estude o código para ver como que a geração dos dados é realizada.

A seguir apresento uma estratégia de implementação.

1. Implemente um TAD BufferEntrada. A idéia é que esse TAD encapsule a funcionalidade de ler os arquivos de entrada N registros de cada vez, e também retorne o próximo registro. Implemente as seguintes operações:

- **Criar**: aloca espaço para N registros, abre o arquivo de entrada, lê os primeiros N registros do arquivo de entrada e inicializa quaisquer variáveis de controle necessárias;
- **Proximo**: retorna o próximo registro do buffer, mas não o consome;
- **Consumir**: consome o próximo registro do buffer e o retorna. Neste caso "consumir" é apontar para o próximo elemento do buffer. Caso o registro atual seja o último do buffer, ler os próximos N registros do arquivo de entrada;
- **Vazio**: Retorna 1 se todos os registros do arquivo já foram consumidos pela função "Consumir". Caso contrário, retorna 0;
- **Destruir**: fechar o arquivo de entrada e liberar quaisquer variáveis alocadas dinâmicamente.

Fique a vontade para implementar mais operações caso seja necessário.

2. Implemente um TAD BufferSaida. A idéia é que esse TAD encapsule a funcionalidade de despejar o conteúdo de seu buffer interno em um arquivo de saída quando o buffer se torna cheio.

- **Criar**: aloca espaço para N registros e abre o arquivo de saída;
- **Inserir**: insere um registro no buffer de saída. Caso o buffer tenha N registros inseridos, despeje o conteúdo do buffer no arquivo de saída.
- **Despejar**: escrever o conteúdo do buffer de saída no arquivo de saída. Caso apenas $M < N$ elementos estejam inseridos no buffer, escreva apenas os M elementos no arquivo de saída. (Essa operação é comumente chamada de *dump*).
- **Destruir**: fechar o arquivo de saída e liberar quaisquer variáveis alocadas dinamicamente.

Fique a vontade para implementar mais operações caso seja necessário.

3. Implemente a intercalação em K-vias descrita acima e mostrada na Figura 2. Sua função deve receber como entrada os buffers de entrada e o buffer de saída já criados. Use os TADs sugeridos acima como buffers de entrada e saída.

4. Implemente a Ordenação Externa com Intercalação em K-vias descrita acima e mostrada na Figura 1. Sua função deve receber o nome do arquivo de entrada, o número máximo de bytes que podem ser usados

para armazenar registros durante toda a operação (B), o tamanho do buffer de saída em bytes (S) e o nome do arquivo de saída. Considere que B e S são múltiplos de 1024. Sua função pode criar arquivos temporários, mas não esqueça de apagar todos esses arquivos no final da função. Siga as orientações abaixo para dividir o arquivo de entrada em K pedaços e para determinar o número de registros em cada buffer de entrada.

Seja E o tamanho do arquivo de entrada, em bytes. Assim, sem exceder B, podemos dividir o arquivo de entrada em K pedaços, onde $K = \lceil \frac{E}{B} \rceil$, e $\lceil \rceil$ é a função "teto".

Como S bytes devem ser usados para o buffer de saída, restam $B - S$ bytes para os buffers de entrada. Como temos K pedaços e cada registro tem 1024 bytes, então cada buffer de entrada deve armazenar:

$$\left\lfloor \frac{\left(\frac{B-S}{K}\right)}{1024} \right\rfloor$$

registros.

Análise

Crie arquivos usando o código fornecido com *seed=42* com 256000, 512000, 921600 e 1572864 registros. Preencha as tabelas a seguir com o tempo de execução, em segundos, da ordenação dos arquivos.

		S		
		B/8	B/4	B/2
B	8388608 (8MB)			
	16777216 (16MB)			
	33554432 (32MB)			

Figure 3: Tempo De Execução Para Ordenar Arquivo com 256000 Registros

		S		
		B/8	B/4	B/2
B	16777216 (16MB)			
	33554432 (32MB)			
	67108864 (64MB)			

Figure 4: Tempo De Execução Para Ordenar Arquivo com 512000 Registros

Regras

- O trabalho pode ser feito (no máximo) em duplas.
- Se o trabalho for feito em duplas, ambos alunos tem que escrever parte do código. Em outras palavras, não é pra um aluno fazer o código e o outro fazer o relatório.
- O trabalho deve ser implementado em linguagem C, usando modularização e Makefile para compilar.
- Os TADs sugeridos devem ser implementados e utilizados, conforme descrito. Pode implementar funções adicionais caso seja necessário.

		S		
		B/8	B/4	B/2
B	67108864 (64MB)			
	134217728 (128MB)			
	268435456 (256MB)			

Figure 5: Tempo De Execução Para Ordenar Arquivo com 921600 Registros

		S		
		B/8	B/4	B/2
B	67108864 (64MB)			
	134217728 (128MB)			
	268435456 (256MB)			

Figure 6: Tempo De Execução Para Ordenar Arquivo com 1572864 Registros

- Trabalhos plagiados uns dos outros ou da internet serão totalmente desconsiderados, sem possibilidade de recuperação.
- Use a biblioteca padrão da linguagem C de entrada e saída para fazer as operações com arquivos (stdio.h).
- Comente o código nos pontos mais importantes. Não é necessário fazer textão :)

Entrega

- Um arquivo .zip (ou .tar.gz) contendo todo o código do trabalho e o pdf de um relatório de implementação e análise, especificado abaixo. Inclua também um arquivo alunos.txt, com os nomes dos alunos da equipe, e um LEIAME.txt com informações adicionais para compilação e execução.
- Um relatório de implementação e análise, com no máximo 10 páginas, descrevendo a implementação desenvolvida. Essa descrição deve indicar como que o código foi modularizado e quais foram os TADs adicionais implementados. Inclua como foi realizada a divisão do trabalho entre os membros da equipe. Inclua também as tabelas obtidas na seção Análise, e discuta os resultados. Reporte se você executou os testes em um SSD ou HDD.
- Prazo máximo: **02/12/2021 às 23:59**, via Moodle. Ambos integrantes da equipe precisam enviar o trabalho.

Bom Trabalho!