MAC-211 — Laboratório de Programação

Exercício Programa 2: JBase — Gerenciador de Buffers

Entrega: 23 de abril de 1999

1 Introdução

Este e os próximos exercícios-programa são etapas de um projeto considerável, cujo objetivo é construir um sistema de armazenamento de objetos com características bem diferentes do sistema de arquivos do Unix:

- **Registros**. Enquanto um arquivo Unix é uma seqüência de bytes sem nenhuma estrutura adicional, os arquivos do nosso sistema serão organizados em registros de dados. Um arquivo conterá uma ou mais coleções de registros. Cada registro corresponderá a um objeto armazenado em disco. Exemplificando: um objeto do tipo `Aluno`, com campos `númeroUSP`, `nome`, `endereço` e `anoDeIngresso`, será armazenado num registro dividido em campos correspondentes aos campos do `Aluno`. Uma aplicação poderá guardar, num mesmo arquivo, as informações sobre um conjunto de alunos. A aplicação poderá fazer busca, inserção, remoção e modificação de um elemento do conjunto.

- **Indexação**. O sistema permitirá localização eficiente de registros, dada uma chave de busca. A idéia é que consultas como "obter os dados do aluno cujo número USP é 2370084" (neste caso a chave de busca é o `númeroUSP`) ou "obter os nomes dos alunos que ingressaram na década de 80" (agora a chave de busca é o `anoIngresso`) possam ser realizadas sem que o sistema tenha de examinar seqüencialmente cada registro de aluno, até encontrar o(s) registro(s) desejado(s).

Nosso sistema de armazenamento de objetos será um pacote escrito em Java, voltado para aplicações escritas também em Java. Por lembrar um database system, vamos denominá-lo JBase[1]. O software do JBase será organizado em três camadas:

1. **Gerenciador de buffers**. Mantém num pool de buffers um subconjunto das páginas de um "banco de dados" residente em disco, com o objetivo de minimizar as operações de transferência de páginas entre a memória e o disco.

2. **Gerenciador de arquivos**. Implementa o conceito de "arquivo" como uma ou mais coleções de registros. Além de dividir cada página em registros, esta camada aloca as páginas de um arquivo às várias coleções de registros armazenadas no arquivo. O gerenciador de arquivos utiliza os serviços do gerenciador de buffers.

[1] Quem tiver alguma familiaridade com sistemas de gerenciamento de bancos de dados (SGBDs) vai notar que o JBase presta alguns dos serviços (não todos) normalmente oferecidos por SGBDs.

3. **Gerenciador de índices**. Mantém estruturas de apoio à busca de registros. Dada uma condição expressa em termos de uma chave de busca, esta camada permite a busca eficiente do subconjunto dos registros que satisfazem a condição. O gerenciador de índices é construído sobre as duas camadas anteriores.

Sistemas de gerenciamento de bancos de dados (SGBDs) tem camadas com funcionalidade semelhante. As camadas do JBase de fato correspondem às camadas inferiores de um SGBD. Num sistema de bancos de dados completo, entretanto, tais camadas levam em conta duas questões adicionais: controle de concorrência (podem haver acessos "simultâneos" ao banco de dados) e recuperação de falhas (a integridade dos dados é garantida mesmo que ocorram quedas do sistema). No JBase vamos deixar de lado essas questões.

2 A Primeira Etapa do Projeto

O objetivo desta etapa é implementar e testar o gerenciador de buffers do JBase. Para uma visão geral sobre gerenciamento de buffers, leia o texto "Storing Data: Disks and Files", disponível no xerox do CAMAT. Faça isso antes de prosseguir com o projeto.

Nesta etapa você vai escrever três classes Java:

- A classe `BufferManager`, que corresponde ao gerenciador de buffers propriamente dito. Ela mantém um "buffer pool" — um conjunto de buffers (também denominados frames), cada um com espaço para um bloco (ou página) do banco de dados.

- A classe `Page`. Os elementos do "buffer pool" são instâncias desta classe. Apesar do nome da classe, suas instâncias não correspondem a páginas do banco de dados, e sim a áreas de memória que *podem* conter páginas[2].

- A classe `DBFile`. Uma instância dessa classe é uma seqüência de blocos de disco que podem ser individualmente lidos ou escritos mediante chamadas a métodos `readBlock` ou `writeBlock`. Ela poderia ser implementada de modo que suas instâncias correspondessem a discos ou a partições de disco, mas não vamos fazer isso. Nossa implementação mapeará `DBFile`s em arquivos Unix[3].

Você escreverá também uma ou mais classes de teste, com o objetivo de exercitar o seu gerenciador de buffers e mostrar que ele funciona corretamente.

3 A Classe `BufferManager`

Esta classe tem métodos para

- trazer uma página do banco de dados para o buffer pool e prender ("alfinetar") a página a um frame;

- soltar ("desalfinetar") uma página presa a um frame;

[2]Um nome como `Frame` seria mais preciso. Eu prefiro `Page` mesmo, porque (1) um frame geralmente contém uma página, (2) frames vazios (sem páginas) não são vistos pelas camadas superiores do JBase, e (3) sempre que uma dessas camadas manipular um frame, o que ela na verdade deseja é manipular a página contida no frame.

[3]Mais precisamente: nossa implementação mapeará `DBFile`s em instâncias de uma classe padrão da plataforma Java, a classe `java.io.RandomAccessFile`.

- forçar a escrita em disco de uma página particular;
- forçar a escrita em disco de todas as páginas no buffer pool;
- mudar a política de substituição de páginas do pool.

Esses métodos estão descritos abaixo.

```
package br.usp.ime.jbase;
import java.io.IOException;

class BufferManager {

   // campos

    private Page[] bufferPool; // o buffer pool é um array de frames

    ...                        // outros campos (tabela de hash para
                               // busca de um frame por pageId, etc...)

    // construtor

    BufferManager(int numberOfFrames, int pageSize, int hashSize) {

        // Constrói um BufferManager com os parâmetros especificados.
        ...
    }

    // métodos

    Page pin(long pageId, boolean readPageIntoFrame)
            throws InvalidFileIdException,
                   InexistentPageException,
                   IOException {

        // Verifica se a página desejada está no buffer pool. Se estiver,
        // incrementa o pinCount do frame com a página e retorna uma
        // referência para esse frame. Se o pinCount era 0 antes da chamada
        // a pin (isto é, se a página era uma candidata à substituição e
        // está deixando de sê-lo), remove a página da lista de candidatas à
        // substituição.  Se a página desejada não estiver no pool e se
        // algum frame do pool estiver desocupado, põe a página nesse frame
        // e retorna uma referência para ele. Se todos os frames estiverem
        // ocupados e a lista de páginas candidatas a substituição não
        // estiver vazia, troca uma das páginas do pool (retirada da lista
        // de candidatas à substituição) pela página desejada, "alfineta"
        // essa página a seu frame no pool e retorna uma referência para
        // esse frame. Retorna null caso a página desejada não esteja no
        // pool e não haja frame desocupado e a lista de páginas candidatas
        // a substituição esteja vazia.
        ...
```

```
    }

    void unpin(Page page)
            throws NotPinnedException {

        // Decrementa o pinCount da página e, caso ele chegue a zero,
        // insere a página na lista de candidatas a substituição.
        // Lança uma exceção se for chamado com uma página cujo pinCount
        // já é zero.
        ...
    }

    void flush(Page page)
            throws InvalidFileIdException,
                   InexistentPageException,
                   IOException {

        // Caso o conteúdo da página especificada tenha sido alterado no
        // buffer pool, força sua escrita em disco.
        ...
    }

    void flushAll()
            throws InvalidFileIdException,
                   InexistentPageException,
                   IOException {

        // Força a escrita em disco de todas as páginas cujo conteúdo foi
        // alterado no buffer pool.
        ...
    }

    void setReplacementPolicy(boolean lru) {

        // Define a política de substituição de páginas (LRU ou MRU) a ser
        // usada pelo BufferManager daqui para frente. A política default
        // é a LRU.
        ...

    }

}
```

O `bufferPool` é uma coleção de frames gerenciada pelo `BufferManager`. É implementado como um array de objetos `Page`. Cada objeto `Page`, além de possuir um buffer com espaço para uma página do banco de dados, possui campos

```
long id;
int pinCount;
boolean dirty;
```

O `id` especifica a página do banco de dados contida no buffer do objeto `Page`. É um `long` que distingue uma página de todas as demais páginas. O `pinCount` é o número de "alfinetes" prendendo a página a seu frame no buffer pool. O flag `dirty` indica que o conteúdo da página foi atualizado no buffer pool mas não no banco de dados.

A classe `BufferManager` deve manter as seguintes estruturas de dados:

1. **Tabela de páginas no buffer pool**. É uma tabela de hash que, conceitualmente, associa identificadores de páginas a objetos `Page`. Implemente-a como um array de listas ligadas de objetos `Page`[4]. O comprimento desse array é dado pelo parâmetro `hashSize` do construtor de um `BufferManager`. O `BufferManager` usa esta tabela para descobrir se uma dada página se encontra no buffer pool ou não e, em caso afirmativo, em que frame ela está.

2. **Lista de frames desocupados**. É uma lista ligada com os objetos `Page` do buffer pool que não contém páginas do banco de dados.

3. **Lista de páginas candidatas a substituição**. É uma lista duplamente ligada com os objetos `Page` do buffer pool que contém páginas "soltas" (tem `pinCount` igual a zero). O `BufferManager` usa esta lista para implementar duas políticas de substituição de páginas: a política LRU (least recently used), segundo a qual a página usada menos recentemente é a "vítima" da substituição, e a política MRU (most recently used), segundo a qual a "vítima" é a página usada mais recentemente.

   A ordem das páginas nesta lista reflete a ordem de utilização das páginas. Quando o seu `pinCount` atinge zero, um objeto `Page` é inserido ao final da lista. Caso a política de substituição seja LRU, a lista é tratada como um fila (FIFO): a vítima da próxima substituição é retirada do início da lista. Caso a política de substituição seja MRU, a lista é tratada como uma pilha (LIFO): a vítima da próxima substituição é retirada do final da lista.

As substituições de páginas são efetuadas pelo método `pin`. Para substituir uma página, esse método deve:

1. Usar a política de substituição em vigor (LRU ou MRU) para escolher um frame contendo a página a ser substituída.

2. Caso a página escolhida esteja "suja" (`dirty`), deve descarregá-la para disco (isto é, deve escrevê-la no bloco de disco correspondente, usando o método `writeBlock` da classe `DBFile`).

3. Caso tenha sido chamado com `true` no parâmetro `readPageIntoFrame`, o método `pin` deve ler a página desejada (usando o método `readBlock` da classe `DBFile`) para o frame escolhido.

4. Atualizar com `false` o flag `dirty` desse frame.

5. Atualizar a tabela de páginas no buffer pool, refletindo o fato de que uma página saiu do pool e outra entrou.

[4] Há um bom motivo (que discutiremos em classe) para *não* se implementar esta tabela usando uma classe "pronta", como a classe `java.util.Hashtable` do JDK.

4 A Classe `Page`

Além dos campos `id`, `pinCount` e `dirty`, cada instância desta classe tem um buffer com espaço para uma página do banco de dados. Toda e qualquer transferência de informações entre uma aplicação e o banco de dados é feita através destes buffers, que portanto precisam conter dados de tipos variados: `bytes`, `chars`, `Strings`, `ints`, `longs`, etc. Como Java é uma linguagem "fortemente tipada", o buffer de um objeto `Page` não pode ser simplesmente um `array` de `bytes`. A classe `br.usp.ime.jbase.Buffer`, cujo código fonte está disponível na página do curso, resolve esse problema. Faça o objeto `Page` conter uma referência para um `Buffer`:

```
package br.usp.ime.jbase;

class Page {

    // campos
    long id;
    int pinCount;
    boolean dirty;
    Buffer buffer;
    Page next;
    Page lruNext;
    Page lruPrevious;
    ...

    // métodos
    ...
}
```

O campo `next` funciona como campo de encadeamento tanto para implementar as listas ligadas que compõe a tabela de hash como para a implementar a lista de frames desocupados. (Um só campo é suficiente, pois frames desocupados não aparecem na tabela de hash.) Já os frames ocupados por páginas com `pinCount` igual a zero aparecem tanto na tabela de hash como na lista duplamente ligada de páginas candidatas a substituição. Por esse motivo temos mais dois campos de encadeamento, `lruNext` e `lruPrevious`.

Para seu projeto ficar mais limpo e elegante, faça a classe `Page` funcionar como "fachada" para um `BufferManager`:

```
package br.usp.ime.jbase;
import java.io.IOException;

class Page {

    // campos
    ... // (os já vistos)
    static BufferManager bm; // nb: esta é uma variável de classe (static)

    // métodos

    static void setBufferManager(BufferManager bufMan) { bm = bufMan; }
```

```
    static BufferManager getBufferManager() { return bm; }

    static Page pin(long pageId, boolean readPageIntoFrame)
            throws InvalidFileIdException,
                   InexistentPageException,
                   IOException { return bm.pin(pageId, readPageIntoFrame); }

    void unpin() throws NotPinnedException { bm.unpin(this); }

    void flush()
            throws InvalidFileIdException,
                   InexistentPageException,
                   IOException { bm.flush(this); }

    static void flushAll()
            throws InvalidFileIdException,
                   InexistentPageException,
                   IOException { bm.flushAll(); }

    void setReplacementPolicy(boolean lru) { bm.setReplacementPolicy(lru) }

    ...

}
```

As chamadas aos "métodos de fachada" são simplesmente repassadas ao `BufferManager` referenciado pela variável de classe `bm`, que deve ter sido previamente inicializada atraves de uma chamada a `Page.setBufferManager`. Com este arranjo, os níveis superiores do JBase interagirão somente com a classe `Page` e com instâncias dessa classe:

```
Page.setBufferManager(new BufferManager(1024, 4096, 2617));
...
long pageId;
...
pageId = ...;
Page page = Page.pin(pageId, true); // prende a um frame a pag. especificada
...                                 // usa essa página
page.unpin();                       // e solta a página
```

5 Identificadores de Páginas e de Registros

Dissemos que o JBase usa identificadores de página, mas ainda não definimos o formato desses identificadores. Vamos fazer isso agora. Um identificador de página (page id) é um `long` com a forma

`0000ffffpppppppp` (64 bits)

onde `0000` indica 16 bits iguais a zero, `ffff` é um identificador de arquivo (file id) com 16 bits e `pppppppp` é um número de página com 32 bits. O page id identifica a `pppppppp`-ésima página do arquivo cujo file id é `ffff`.

Os 16 bits mais significativos do page id não são usados, e por isso são sempre iguais a zero. Nas etapas seguintes do projeto do JBase precisaremos também de identificadores de registros (record ids). Usaremos como record ids `long`s com formato análogo ao dos page ids, mas que não precisam conter zero nos 16 bits mais significativos:

`ssssffffpppppppp` (64 bits)

Como antes, `ffff` é um file id com 16 bits e `pppppppp` é um número de página com 32 bits. Os bits `ssss` contém um slot number que especifica um dos registros da `pppppppp`-ésima página do arquivo cujo file id é `ffff`.

É conveniente incluir, na classe `Page`, métodos de classe (`static`) para agrupar num `pageId` um par `(fileId, pageNumber)` especificado, para extrair o `pageNumber` de um `pageId` ou de um `recId` dado, e para extrair o `pageId` de um `recId` dado:

```
package br.usp.ime.jbase;

class Page {

    // campos
    ...

    // métodos
    ...

    static long id(short fileId, int pageNumber) {
        return ((long) fileId << 32) | ((long) pageNumber & 0xffffffffL);
    }

    static long numberFromPID(long pageId) { return pageId & 0xffffffffL; }

    static long numberFromRID(long recId) { return recId & 0xffffffffL; }

    static long idFromRID(long recId) { return recId & 0xffffffffffffL; }

}
```

6 A Classe DBFile

Vimos que um page id do JBase inclui um file id de 16 bits. Arquivos Unix, entretanto, não são identificados por números de 16 bits e sim por pathnames. A principal responsabilidade da classe `DBFile` é manter a associação entre file ids e arquivos Unix[5]. Cada objeto `DBFile` tem campos

```
short fileId;
String fileName;
int blockSize;
long numberOfBlocks;
transient RandomAccessFile raf;
```

[5]Mais precisamente: a principal responsabilidade da classe `DBFile` é manter a associação entre file ids e instâncias da classe `java.io.RandomAccessFile`.

A classe `DBFile` deve também manter, em variáveis de classe (`static`), duas tabelas: uma delas organizada para buscas por file id, associando file ids a `DBFiles`, a outra organizada para buscas por nome, associando nomes de arquivos (relativos ao diretório corrente) a file ids.

A primeira dessas tabelas pode ser facilmente implementada como um vetor de objetos `DBFile`, indexado por file id. A classe `java.util.Vector` ou a classe `java.util.ArrayList` do JDK 1.2 será útil aqui. Para a segunda tabela, use a classe `java.util.Hashtable`[6] ou a classe `java.util.HashMap` do JDK 1.2.

`DBFile` inclui métodos de classe para:

- Criar um `DBFile` com nome, tamanho de página e número de páginas especificados, alocando um file id para o `DBFile` criado. Esta operação poderia corresponder à criação de uma partição de disco ou "disco virtual" de tamanho fixo. Em nossa implementação ela cria um arquivo Unix com o nome especificado e associa a esse arquivo o file id recém-alocado, bem como os parâmetros de criação do arquivo (nome, tamanho de página e número de páginas).
- Obter um `DBFile` já existente, dado o seu nome.
- Obter um `DBFile` já existente, dado o seu file id.

`DBFile` inclui métodos de instância para:

- Ler ou escrever um bloco.
- Fechar o arquivo Unix usado por este objeto `DBFile`.
- Descarregar no arquivo Unix todas as escritas de bloco feitas neste objeto `DBFile` que ainda estiverem pendentes.
- Obter os parametros file id, nome, tamanho de bloco e número de blocos deste objeto `DBFile`.

Esse métodos são mostrados abaixo:

```
package br.usp.ime.jbase;

import java.io.IOException;
import java.io.FileNotFoundException;

class DBFile implements Serializable {

    // campos
    ...

    // métodos

    static DBFile create(String fileName, int blkSize, long nBlks)
            throws AlreadyExistsException, IOException {  ... }
```

[6] Veremos em classe que aqui — ao contrário do caso da tabela de páginas do `BufferManager` — não há motivo para não se usar uma classe "pronta".

```
    static DBFile instance(String fileName)
            throws FileNotFoundException, IOException { ... }

    static DBFile instance(short fileId) throws InvalidFileIdException,
                                                IOException { ... }

    int readBlock(long blockNo, byte[] buf)
             throws InexistentPageException, IOException { ... }

    void writeBlock(long blockNo, byte[] buf)
            throws InexistentPageException, IOException { ... }

    void close() throws IOException { ... }

    void sync() throws IOException { ... }

    short getFileId() { ... }

    String getFileName() { ... }

    int getBlockSize() { ... }

    long getNumberOfBlocks() { ... }

}
```

Sua implementação da classe `DBFile` deve:

- Usar a classe `java.io.RandomAccessFile` para manipular os arquivos do sistema operacional subjacente.
- Armazenar de modo persistente (no arquivo `DBFileTable` do diretório corrente) a associação entre file ids e `DBFiles`. Aqui serão muito úteis as facilidades que Java oferece para serialização de objetos[7].

7 Classes de Teste

Esta parte fica inteiramente a seu critério. Ela pode consistir numa única classe de teste, com um método `main` que exercita toda a funcionalidade do software desenvolvido nesta etapa do projeto. Ou pode ser um conjunto de classes menores, cada uma responsável por parte dos testes. Lembre-se que os testes devem ser suficientes para voce se convencer (e me convencer!) que a primeira etapa do seu projeto está pronta para ser utilizada pelos níveis superiores do JBase. Suas classes de teste devem, no mínimo:

- criar alguns (mais de um) `DBFiles`;
- usar os métodos `Page.pin` e `Page.unpin` para acessar as páginas desses arquivos e preenchê-las com dados de vários tipos;

[7] Note que declaramos a classe `DBFile` como `serializable` e o seu campo `raf` como `transient`.

- acessar novamente essas páginas e verificar se os dados foram escritos corretamente;
- exercitar as duas políticas de substituição de páginas[8];
- incluir um teste com muitas substituições de páginas e outro com poucas substituições. (Aqui você pode variar o número de buffers no pool.)

Para termos alguma uniformidade nos testes, vamos convencionar um tamanho de página de 4096 bytes.

8 O Que Voce Deve Entregar

- Pelo sistema de entrega eletrônica de trabalhos: um arquivo `.tar` contendo:
  - todos arquivos fonte (`.java`) desta etapa;
  - um `Makefile` que compila esses arquivos e roda o(s) programa(s) de teste;
  - um arquivo `README` com instruções adicionais que você julgar necessárias.

  Os ítens acima devem ser "empacotados" num único arquivo `.tar`, pois o sistema de entrega eletrônica recebe só *um* arquivo.
- Em papel: listagens dos arquivos `.java`, do `Makefile` e do arquivo `README`.

Uma pessoa que recebeu o seu arquivo `.tar` e o desempacotou deve poder compilar os fontes e rodar os testes com *muita* facilidade. Use o `Makefile` para automatizar ao máximo esse processo. Caso sejam necessários diretórios organizados de modo especial, inclua tais diretórios no arquivo `.tar`. Se for preciso acertar alguma variável do environment antes de rodar o `make`, não deixe de colocar essa informação no arquivo `README`.

Bom Projeto!

[8] Bônus: Além de exercitar as duas políticas, compare (medindo tempos de execução) os desempenhos dessas políticas em diferentes situações.