Bancos de dados em C

# Programando com GDBM

por Nik Frey - www.sxc.hut

Deixe que o GNU Database Manager gerencie os dados de seus aplicativos e tenha menos dores de cabeça.

Por Lucas Brasilino

O GDBM, ou *GNU Database Manager* [1], é uma biblioteca escrita na linguagem C que possibilita ao programador a manipulação de informações em arquivos de banco de dados. Foi desenvolvido como uma alternativa GNU ao tradicional DBM, que fazia parte das primeiras versões do Unix. Existem outras bibliotecas similares à GDBM, como por exemplo a famosa *Berkeley DB* [2], mantida pela Sleepycat Software, a QDBM [3], mantida por Mikio Hirabayashi e a CDB [4], mantida por Daniel Bernstein, entre outras.

As principais vantagens do uso do GDBM são a facilidade, a velocidade e pouco consumo de recursos do sistema, graças ao uso de algoritmos *hash* [5]. Entre os softwares conceituados que o adotaram estão o OpenLDAP, o Sendmail e o Cyrus SASL.

Como já foi dito, o GDBM é voltado a programadores e não a usuários, ao contrário dos conhecidos MySQL, PostgreSQL e outros. Com ele é possível "embutir" recursos de banco de dados em nossos programas. Linguagens como PHP, Python e Perl possuem funções para utilizá-lo.

**Tabela 1: exemplo de banco de dados**

| Matrícula (chave): | Nome (dado) |
|---|---|
| 1111: | Simone |
| 2222: | Cabral |
| 3333: | Pedro |
| 4444: | Augusto |

Neste artigo abordaremos seu uso na linguagem C. Os programas delete.c, fetch.c, seq.c, store.c e teste.c, mencionados ao longo do texto, podem ser baixados no site da Linux Magazine, em [6]

## Nas entranhas do GDBM

O armazenamento das informações é realizado na forma de chave/dado, em que a chave nunca pode ser repetida em um mesmo banco. Por exemplo, poderíamos criar um banco que contivesse as informações da **tabela 1**.

Seria impossível ter outra entrada com a chave "2222". Se tentássemos inseri-la, o GDBM retornaria um código de erro. No máximo poderíamos substituí-la. Trataremos disso mais adiante.

Cada banco de dados é um arquivo chamado de *banco de dados GDBM*. É possível que vários processos simultâneos abram o mesmo arquivo no modo de leitura. Já no modo de gravação só é permitido o acesso a um processo por vez.

Tanto o dado quanto a chave devem ser do tipo `datum`, que é uma estrutura definida apenas com dois membros: `dptr` e `dsize`. O membro `dptr` é um apontador para caracter (`char *`) onde deve-se apontar para a chave ou o dado. O membro `dsize` é um inteiro (`int`) que deve armazenar o comprimento da chave ou dado apontado por `dptr`. A **listagem 1** demonstra sua utilização.

Note que devemos sempre incluir o arquivo *header* (cabeçalho) `gdbm.h` em nossos programas, pois nele estão declarados todos os protótipos das funções que iremos abordar, bem como os objetos, constantes (*flags*) etc.

A função `strlen()` retorna o comprimento da cadeia de caracteres. Para maiores informações sobre ela execute o comando `man 3 strlen` no Shell.

Em programação C os objetos são tecnicamente estruturas. Como gosto das nuances da programação orientada a objetos, durante este artigo iremos abordar estruturas como objetos.

## Criando e abrindo um banco de dados

Para abrir um banco de dados, ou criá-lo caso não exista, utilizamos a função `gdbm_open()`. Essa função nos retorna um objeto do tipo `GDBM_FILE` que obrigatoriamente deve ser utilizado em todas as chamadas

**Listagem 1: uso do tipo *datum***

```
01 char chave[] = "1111", dado[] = "simone";
02 datum key, data;
03 key.dptr = chave;
04 key.dsize = strlen (chave);
05 data.dptr = dado;
06 data.dsize = strlen (dado);
```

**Listagem 2: criação de um banco de dados**

```
01 #include <stdio.h>
02 #include <gdbm.h>
03 int
04 main (void)
05 {
06  GDBM_FILE db;
07  db = gdbm_open ("database.db", 512, GDBM_WRCREAT, 0644, NULL);
09  if (db == NULL)
10    {
11     fprintf (stderr, "Não pude abrir/criar banco de dados!\n");
12     return 1;
13    }
14  gdbm_close (db);
15  return 0;
16 }
```

subseqüentes a essa biblioteca, excluindo-se apenas a chamada `gdbm_strerror()`, que retorna uma frase identificando um erro ocorrido. Veja na **listagem 2** como criar um novo banco.

Certamente esse programa não é muito útil, pois estamos abrindo (ou criando) um banco de dados, testando se ele foi mesmo aberto e imediatamente fechando-o. Seu intuito é ser puramente didático.

A função `gdbm_open()` recebe 5 parâmetros. São eles:

- O nome do arquivo, podendo-se incluir o caminho completo.
- Tamanho do bloco em bytes. A cada acesso ao arquivo, o GDBM transfere os dados em blocos de tamanho definido neste parâmetro. 512 bytes é o menor tamanho.
- Uma *flag* que determina a forma da abertura do arquivo.
- O modo (permissões de acesso), em notação octal.
- Uma função, declarada na forma `void func()`, que o GDBM chamará caso encontre um erro fatal. `NULL` fará com que seja chamada uma função padrão.

Existem várias flags disponíveis, são elas:

- `GDBM_WRITER`: o programa poderá tanto ler quanto escrever no arquivo, porém deverá ter acesso exclusivo.
- `GDBM_WRCREAT`: semelhante ao `GDBM_WRITER` mas, se o arquivo não existir, será criado.
- `GDBM_READER`: o programa poderá apenas ler dados do arquivo. É permitida a leitura por múltiplos programas simultaneamente.
- `GDBM_NEWDB`: semelhante à `GDBM_WRCREAT`, porém sempre criará um arquivo novo, sobrescrevendo um antigo com o mesmo nome, caso exista. Use-a com cuidado.

Essas flags podem ser combinadas através de um operador lógico `OR` (`|`) com duas outras flags bem específicas:

- `GDBM_SYNC`: Sincroniza todas as operações com o arquivo. O ponto positivo é que evitaremos corrupção de dados caso o sistema seja desligado indevidamente e o ponto negativo é que perdemos, e muito, em desempenho.
- `GDBM_NOLOCK`: Desliga as rotinas de *locking* gerenciadas pela biblioteca. O próprio programa deverá fazê-lo.

Por exemplo, poderíamos utilizar `gdbm_open()` para abrir o banco `database.db` para escrita no modo de sincronismo:

```
db = gdbm_open (“database.db”, 512, GDBM_WRITER | GDBM_SYNC, 0644, NULL);
```

Por padrão o GDBM abre um arquivo no modo assíncrono e gerencia seu próprio *locking*.

Na **listagem 2** vemos que a chamada ao `gdbm_open()` retorna `NULL` caso haja algum erro. Os erros mais comuns são a falta de permissão de acesso ao arquivo e a tentativa de acesso nos modos `GDBM_WRITER` ou `GDBM_WRCREAT` com um arquivo já aberto por outro processo.

A função `gdbm_close()`, como o nome já sugere, fecha um arquivo de banco de dados. Ela deve ser sempre chamada antes do fim do programa. Assim, o GDBM sincroniza todos os dados entre memória e o arquivo em disco evitando qualquer corrupção dos dados e libera áreas de memória alocadas para o objeto `GDBM_FILE`.

## Armazenando um dado

O programa `store.c` contém uma rotina completa de exemplo de como um par chave/dado é armazenado utilizando o GDBM. Para compilá-lo, devemos executar o comando:

```
$ gcc store.c -o store -lgdbm
```

A opção `-lgdbm` diz ao *link-editor* para ligar o binário à biblioteca GDBM (*libgdbm*). Em seguida, basta executá-lo passando como argumentos do binário `store` a chave e o dado:

```
$ ./store 1111 simone
```

Ao se listar o conteúdo do diretório corrente, o arquivo `database.db` deve ser encontrado o arquivo. Ele é o nosso banco de dados. Quem realmente armazena o par chave/dado é a função `gdbm_store()`:

```
ret = gdbm_store (db, key, data, GDBM_INSERT);
```

Essa função recebe 4 parâmetros:

- o objeto relacionado a um arquivo (`db`);
- a chave (`key`), previamente preenchida;
- o dado (`data`), previamente preenchido;
- e uma flag.

Essa flag pode assumir dois valores:

- `GDBM_INSERT`: insere um par chave/dado novo, retorna erro caso a chave exista;
- `GDBM_REPLACE`: insere ou substitui um par chave/dado.

A função `gdbm_store()` retorna um valor inteiro indicando se sua execução foi realizada ou não a contento. Caso retorne `0`, o par chave/dado foi inserido/substituído. Caso retorne `1`, o par chave/dado não foi inserido, provavelmente porque foi utilizada a flag `GDBM_INSERT` e a chave já existia no arquivo. Caso retorne `-1`, o armazenamento não foi possível porque o banco de dados foi aberto apenas para leitura (flag `GDBM_READER`) ou o objeto do tipo `GDBM_FILE` é `NULL` (o arquivo não foi aberto/criado) ou o membro `dptr` da chave ou dado é `NULL`.

Como exercício, insira todas as chaves/dados descritos na **tabela 1**.

## Buscando um dado

O programa `fetch.c` nos mostra como podemos buscar um dado referente a uma chave no arquivo. Basta chamar a função `gdbm_fetch()`:

```
data = gdbm_fetch (db, key);
```

Ou seja, ao passarmos a chave nos é retornado o dado, lembrando que este se encontra no membro `dptr`. *Um ponto muito importante*: o GDBM usa a função `malloc()` para alocar dinamicamente uma área de memória para armazenar o dado, portanto devemos liberá-la com a função `free()`, como podemos ver no código-fonte do `fetch.c`:

```
free (data.dptr);
```

Compile e execute-o, passando a chave como argumento:

```
$ gcc fetch.C -o fetch -lgdbm
$ ./fetch 1111
Chave: 1111
Dado: simone
```

## Excluindo um dado

Excluir um par chave/dado também é simples, basta utilizar a função `gdbm_delete()` informando uma chave. Para exemplificar, vamos utilizar o programa `delete.c`. Compile-o de forma semelhante aos programas mostrados anteriormente.

Como pode ser visto no código fonte, a exclusão está sendo feita pela chamada:

```
ret = gdbm_delete (db, key);
```

Essa chamada também retorna um valor inteiro, `0` caso a exclusão tenha sido realizada e `-1` caso não tenha sido realizada: nesse caso, ou a chave não foi encontrada ou o objeto `db` não é válido.

Vamos adicionar um dado e imediatamente excluí-lo:

```
$ ./store 5555 vilaca
$ ./delete 5555
```

## Acessando todo o banco de dados

O GDBM provê duas funções interessantes que nos permitem acessar todos os pares chave/dado. São as funções `gdbm_firstkey()` e `gdbm_nextkey()` que retornam, respectivamente, a primeira chave na estrutura interna do GDBM e a próxima chave.

É necessário esclarecer que a primeira chave a ser recuperada não será necessariamente a primeira chave que armazenamos, nem que as chaves serão recuperadas na ordem de armazenagem.

A idéia é simples: de posse da primeira chave retornada pela função `gdbm_firstkey()` faz-se um laço chamando repetitivamente `gdbm_nextkey()` até este retornar `NULL`, indicando que não há mais chaves a retornar. Essa rotina seria a seguinte:

```
for (key = gdbm_firstkey (db); key.dptr;
 key = gdbm_nextkey (db, key))
 {
    /* algum código */
 };
```

`gdbm_firstkey()` apenas recebe um argumento, já `gdbm_nextkey()` recebe dois. Seu segundo argumento é a chave acessada pela chamada anterior.

O programa `seq.c` utiliza esta rotina. Compile-o e rode-o como já explicado:

```
$ ./seq
Chave: 2222
Dado: cabral
Chave: 3333
Dado: pedro
Chave: 1111
Dado: simone
Chave: 4444
Dado: augusto
```

Utilizando essas duas funções, o GDBM nos garante que todos os pares chave/dados serão acessados.

## Erros

O GDBM possui seu próprio conjunto de mensagens de erro que podem ser acessadas pela função `gdbm_strerror()`. Em todos os exemplos de programas neste artigo esse conjunto foi utilizado para indicar algum problema. O único parâmetro dessa função é a variável inteira global `gdbm_errno` que é definida no arquivo `gdbm.h` (por isso não tivemos que declará-la).

O uso de `gdbm_sterror()` não é obrigatório, mas é bastante cômodo.

## Funções adicionais

Existem algumas funções que raramente são usadas, mas é interessante conhecê-las. A necessidade de sua utilização depende diretamente da forma como o programa foi escrito, e não da biblioteca em si.

A primeira é `gdbm_sync()`, que tem como argumento o objeto do banco de dados (`GDBM_FILE`). Como por padrão um arquivo é aberto na forma assíncrona, essa função força a sincronização entre os dados que estão num *buffer* na memória e o arquivo no disco. Ela é automaticamente chamada quando fechamos o banco (`gdbm_close()`). Portanto só é necessário usa-la se muitas operações de inclusão e exclusão foram feitas sem fechar e reabrir o arquivo.

A segunda é `gdbm_reorganize()`, que recebe o mesmo argumento que a função anterior. Sua utilidade é diminuir o tamanho do arquivo quando se fizeram muitas exclusões. Na realidade o GDBM não exclui "fisicamente" um par chave/dado do arquivo quando utilizamos `gdbm_delete()`; ele apenas marca aquele par como deletado e nunca mais o acessa, porém os dados continuam a ocupar espaço.

A terceira é `gdbm_setopt()`, que é utilizada para ajustar algumas opções da biblioteca, como ligar ou desligar o modo síncrono, tamanho do buffer interno etc.

A última é `gdbm_fdesc()`, utilizada para permitir ao programador avançado gerenciar ele próprio o "locking" de seus arquivos, desde que os mesmos tenham sido abertos com a flag `GDBM_NOLOCK`.

O GDBM esconde atrás de sua interface simples um mecanismo rápido e robusto de armazenamento e recuperação de dados. Suas possibilidades só são limitadas pela imaginação do programador. Ele também pode ser instrutivo para quem deseja se aventurar com bibliotecas similares, como a Berkeley DB. Mãos à obra! ■


**Sobre o autor**

*Lucas Brasilino começou a usar Linux nos idos de 95 e não largou mais. Trabalhou na LBS como instrutor e consultor e na Emprel (Prefeitura do Recife) na área de suporte a redes e servidores rodando Software Livre. É desenvolvedor do SIRI –* ***Sistema Integrado de Gestão de Recursos de Internet.*** *[7]*


**Informações**


[1] www.gnu.org/software/gdbm
[2] www.sleepycat.com/products/db.shtml
[3] qdbm.sourceforge.net
[4] cr.yp.to/cdb.html
[5] en.wikipedia.org/wiki/Hash_function
[5] sl.recife.pe.gov.br/projects/siri
[6] www.linuxmagazine.com.br/Magazine/current