20

A Database Library

20.1 Introduction

During the early 1980s, the UNIX System was considered a hostile environment for running multiuser database systems. (See Stonebraker [1981] and Weinberger [1982].) Earlier systems, such as Version 7, did indeed present large obstacles, since they did not provide any form of IPC (other than half-duplex pipes) and did not provide any form of byte-range locking. Many of these deficiencies were remedied, however. By the late 1980s, the UNIX System had evolved to provide a suitable environment for running reliable, multiuser database systems. Since then, numerous commercial firms have offered these types of database systems.

In this chapter, we develop a simple, multiuser database library of C functions that any program can call to fetch and store records in a database. (Such a database is often called a *key–value store*.) This library of C functions is usually only one part of a complete database system. We do not develop the other pieces, such as a query language, leaving these items to the many textbooks on database systems. Our interest is the parts of the UNIX System interface required by a database library, and how they relate to the topics we've already covered (such as record—byte-range—locking, in Section 14.3).

20.2 History

One popular library of database functions in the UNIX System is the `dbm`(3) library. This library was developed by Ken Thompson and uses a dynamic hashing scheme. It was originally provided with Version 7, appears in all BSD releases, and was also

provided in SVR4's BSD-compatibility library [AT&T 1990c]. The BSD developers extended the dbm library and called it ndbm. The ndbm library was included in BSD as well as in SVR4. The ndbm functions are standardized in the XSI option of the Single UNIX Specification.

Seltzer and Yigit [1991] provide a detailed history of the dynamic hashing algorithm used by the dbm library and other implementations of this library, including gdbm, the GNU version of the dbm library. Unfortunately, a basic limitation of all these implementations is that none allows concurrent updating of the database by multiple processes. These implementations provide no type of concurrency controls (such as record locking).

4.4BSD provided a new db(3) library that supports three forms of access: (a) record oriented, (b) hashing, and (c) a B-tree. Again, no form of concurrency was provided (as was plainly stated in the BUGS section of the db(3) manual page).

> Oracle (`http://www.oracle.com`) provides versions of the db library that do support concurrent access, locking, and transactions.

Most commercial database libraries do provide the concurrency controls required for multiple processes to update a database simultaneously. These systems typically use advisory locking, as we described in Section 14.3, but they often implement their own locking primitives to avoid the overhead of a system call to acquire an uncontested lock. These commercial systems usually implement their database using B+ trees [Comer 1979] or some dynamic hashing technique, such as linear hashing [Litwin 1980] or extendible hashing [Fagin et al. 1979].

Figure 20.1 summarizes the database libraries commonly found in the four operating systems described in this book. Note that on Linux, the gdbm library provides support for both dbm and ndbm functions.

Library	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
dbm			gdbm		•
ndbm	XSI	•	gdbm	•	•
db		•	•	•	

Figure 20.1 Support for database libraries on various platforms

20.3 The Library

The library we develop in this chapter will be similar to the ndbm library, but we'll add the concurrency control mechanisms to allow multiple processes to update the same database at the same time. We first describe the C interface to the database library, then in the next section describe the actual implementation.

When we open a database, we are returned a handle (an opaque pointer) representing the database. We'll pass this handle to the remaining database functions.

```
#include "apue_db.h"

DBHANDLE db_open(const char *pathname, int oflag, ... /* int mode */);
                                    Returns: database handle if OK, NULL on error

void db_close(DBHANDLE db);
```

If db_open is successful, two files are created: *pathname.idx* is the index file, and *pathname.dat* is the data file. The *oflag* argument is used as the second argument to open (Section 3.3) to specify how the files are to be opened (e.g., read-only, read–write, create file if it doesn't exist). The *mode* argument is used as the third argument to open (the file access permissions) if the database files are created.

When we're done with a database, we call db_close. It closes the index file and the data file and releases any memory that it allocated for internal buffers.

When we store a new record in the database, we have to specify the key for the record and the data associated with the key. If the database contained personnel records, the key could be the employee ID, and the data could be the employee's name, address, telephone number, date of hire, and the like. Our implementation requires that the key for each record be unique. (We can't have two employee records with the same employee ID, for example.)

```
#include "apue_db.h"

int db_store(DBHANDLE db, const char *key, const char *data,
             int flag);
                                    Returns: 0 if OK, nonzero on error (see following)
```

The *key* and *data* arguments are null-terminated character strings. The only restriction on these two strings is that neither can contain null bytes. They may, for example, contain newlines.

The *flag* argument can be DB_INSERT (to insert a new record), DB_REPLACE (to replace an existing record), or DB_STORE (to either insert or replace a record, whichever is appropriate). These three constants are defined in the apue_db.h header. If we specify either DB_INSERT or DB_STORE and the record does not exist, a new record is inserted. If we specify either DB_REPLACE or DB_STORE and the record already exists, the existing record is replaced with the new record. If we specify DB_REPLACE and the record doesn't exist, we set errno to ENOENT and return –1 without adding the new record. If we specify DB_INSERT and the record already exists, no record is inserted. In this case, the return value is 1 to distinguish it from a normal error return (–1).

We can fetch any record from the database by specifying its *key*.

```
#include "apue_db.h"

char *db_fetch(DBHANDLE db, const char *key);
                                    Returns: pointer to data if OK, NULL if record not found
```

The return value is a pointer to the data that was stored with the *key*, if the record is found. We can also delete a record from the database by specifying its *key*.

```
#include "apue_db.h"

int db_delete(DBHANDLE db, const char *key);
```

Returns: 0 if OK, –1 if record not found

In addition to fetching a record by specifying its key, we can go through the entire database, reading each record in turn. To do this, we first call `db_rewind` to rewind the database to the first record and then call `db_nextrec` in a loop to read each sequential record.

```
#include "apue_db.h"

void db_rewind(DBHANDLE db);

char *db_nextrec(DBHANDLE db, char *key);
```

Returns: pointer to data if OK, `NULL` on end of file

If *key* is a non-null pointer, `db_nextrec` returns the key by copying it to the memory starting at that location.

There is no order to the records returned by `db_nextrec`. All we're guaranteed is that we'll read each record in the database once. If we store three records with keys of A, B, and C, in that order, we have no idea in which order `db_nextrec` will return the three records. It might return B, then A, then C, or some other (apparently random) order. The actual order depends on the implementation of the database.

These seven functions provide the interface to the database library. We now describe the actual implementation that we have chosen.

20.4 Implementation Overview

Database access libraries often use two files to store the information: an index file and a data file. The index file contains the actual index value (the key) and a pointer to the corresponding data record in the data file. Numerous techniques can be used to organize the index file so that it can be searched quickly and efficiently for any key: hashing and B+ trees are popular. We have chosen to use a fixed-size hash table with chaining for the index file. We mentioned in the description of `db_open` that we create two files: one with a suffix of `.idx` and one with a suffix of `.dat`.

We store the key and the index as null-terminated character strings; they cannot contain arbitrary binary data. Some database systems store numerical data in a binary format (1, 2, or 4 bytes for an integer, for example) to save storage space. This complicates the functions and requires more work to make the database files portable between different computer systems. For example, if a network has two systems that use different formats for storing binary integers, we need to account for this difference if we want both systems to access the database. (It is not at all uncommon today to have systems with different architectures sharing files on a network.) Storing all the records, both keys and data, as character strings simplifies everything. It does require additional disk space, but that is a small cost for portability.

With `db_store`, only one record for each key is allowed. Some database systems allow a key to have multiple records and then provide a way to access all the records associated with a given key. Additionally, we have only a single index file, meaning that each data record can have only a single key (we don't support secondary keys). Some database systems allow each record to have multiple keys and often use one index file per key. Each time a new record is inserted or deleted, all index files must be updated accordingly. (An example of a file with multiple indexes is an employee file. We could have one index whose key is the employee ID and another whose key is the employee's Social Security number. Having an index whose key is the employee name could be a problem, as names are not always unique.)

Figure 20.2 shows a general picture of the database implementation.

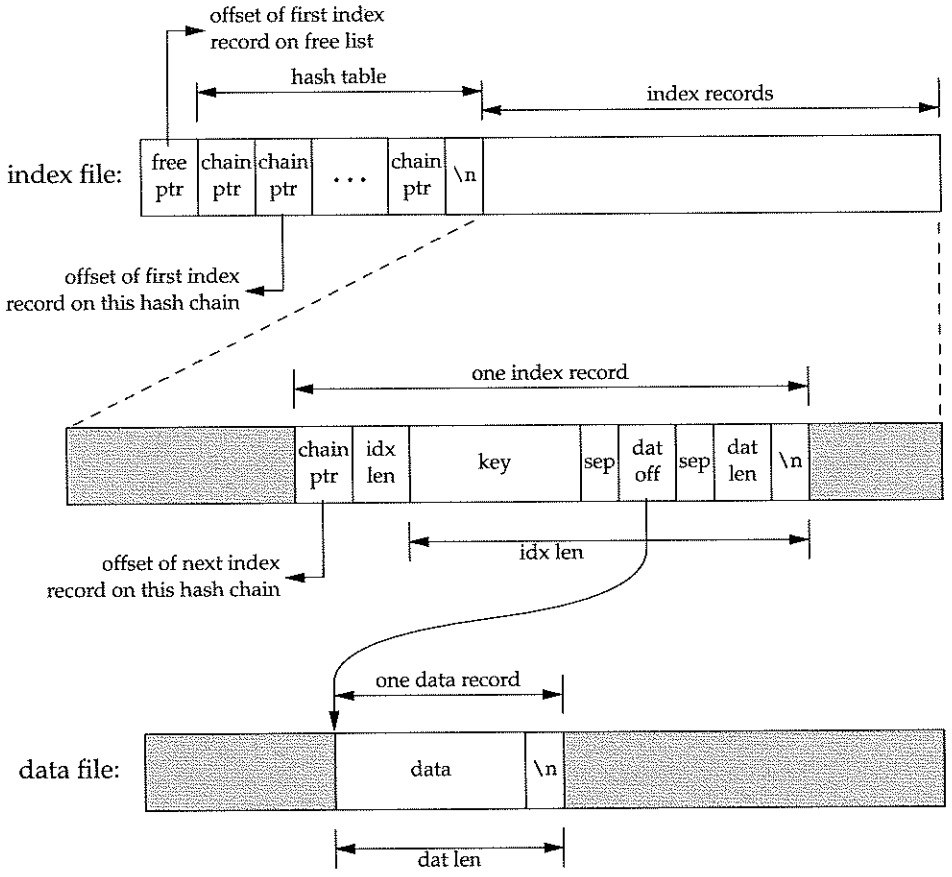

Figure 20.2 Arrangement of index file and data file

The index file consists of three portions: the free-list pointer, the hash table, and the index records. In Figure 20.2, all the fields called *ptr* are simply file offsets stored as an ASCII number.

To find a record in the database given its key, `db_fetch` calculates the hash value of the key, which leads to one hash chain in the hash table. (The *chain ptr* field could be 0, indicating an empty chain.) We then follow this hash chain, which is a linked list of all the index records with this hash value. When we encounter a *chain ptr* value of 0, we've hit the end of the hash chain.

Let's look at an actual database file. The program in Figure 20.3 creates a new database and writes three records to it. Since we store all the fields in the database as ASCII characters, we can look at the actual index file and data file using any of the standard UNIX System tools:

```
$ ls -l db4.*
-rw-r--r--  1 sar        28 Oct 19 21:33 db4.dat
-rw-r--r--  1 sar        72 Oct 19 21:33 db4.idx
$ cat db4.idx
   0  53  35   0
   0  10Alpha:0:6
   0  10beta:6:14
  17  11gamma:20:8
$ cat db4.dat
data1
Data for beta
record3
```

To keep this example small, we have set the size of each *ptr* field to four ASCII characters; the number of hash chains is set to 3. Since each *ptr* is a file offset, a four-character field limits the total size of the index file and data file to 10,000 bytes. When we do some performance measurements of the database system in Section 20.9, we set the size of each *ptr* field to six characters (allowing file sizes up to 1 million bytes) and the number of hash chains to more than 100.

The first line in the index file

```
   0  53  35   0
```

consists of the free-list pointer (0, the free list is empty) and the three hash chain pointers (53, 35, and 0). The next line

```
   0  10Alpha:0:6
```

shows the format of each index record. The first field (0) is the four-character chain pointer. This record is the end of its hash chain. The next field (10) is the four-character *idx len*, the length of the remainder of this index record. We read each index record using two `read`s: one to read the two fixed-size fields (the *chain ptr* and *idx len*) and another to read the remaining (variable-length) portion. The remaining three fields—*key*, *dat off*, and *dat len*—are delimited by a separator character (a colon in this case). We need the separator character, since each of these three fields is variable length. The separator character can't appear in the key. Finally, a newline terminates the index record. The newline isn't required, since *idx len* contains the length of the record. We store the newline to separate each index record so we can use the normal UNIX System tools, such as `cat` and `more`, with the index file. The *key* is the value that we specified

when we wrote the record to the database. The data offset (0) and data length (6) refer to the data file. We can see that the data record does start at offset 0 in the data file and has a length of 6 bytes.

```
#include "apue.h"
#include "apue_db.h"
#include <fcntl.h>

int
main(void)
{
    DBHANDLE    db;

    if ((db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) == NULL)
        err_sys("db_open error");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("db_store error for alpha");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("db_store error for beta");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("db_store error for gamma");

    db_close(db);
    exit(0);
}
```

Figure 20.3 Create a database and write three records to it

(As with the index file, we automatically append a newline to each data record, so we can use the normal UNIX System tools with the file. This newline at the end is not returned to the caller by `db_fetch`.)

If we follow the three hash chains in this example, we see that the first record on the first hash chain is at offset 53 (`gamma`). The next record on this chain is at offset 17 (`Alpha`), and this is the last record on the chain. The first record on the second hash chain is at offset 35 (`beta`), and it's the last record on the chain. The third hash chain is empty.

Note that the order of the keys in the index file and the order of their corresponding records in the data file is the same as the order of the calls to `db_store` in Figure 20.3. Since the `O_TRUNC` flag was specified for `db_open`, the index file and the data file were both truncated and the database initialized from scratch. In this case, `db_store` just appends the new index records and data records to the end of the corresponding file. We'll see later that `db_store` can also reuse portions of these two files that correspond to deleted records.

The choice of a fixed-size hash table for the index is a compromise. It allows fast access as long as each hash chain isn't too long. We want to be able to search for any key quickly, but we don't want to complicate the data structures by using either a B-tree or dynamic hashing. Dynamic hashing has the advantage that any data record can be

located with only two disk accesses (see Litwin [1980] or Fagin et al. [1979] for details). B-trees have the advantage of traversing the database in (sorted) key order (something that we can't do with the `db_nextrec` function using a hash table.)

20.5 Centralized or Decentralized?

Given multiple processes accessing the same database, we can implement the functions in two ways:

1. Centralized. Have a single process that is the database manager, and have it be the only process that accesses the database. The functions contact this central process using some form of IPC.
2. Decentralized. Have each function apply the required concurrency controls (locking) and then issue its own I/O function calls.

Database systems have been built using each of these techniques. Given adequate locking routines, the decentralized implementation is usually faster, because IPC is avoided. Figure 20.4 depicts the operation of the centralized approach.

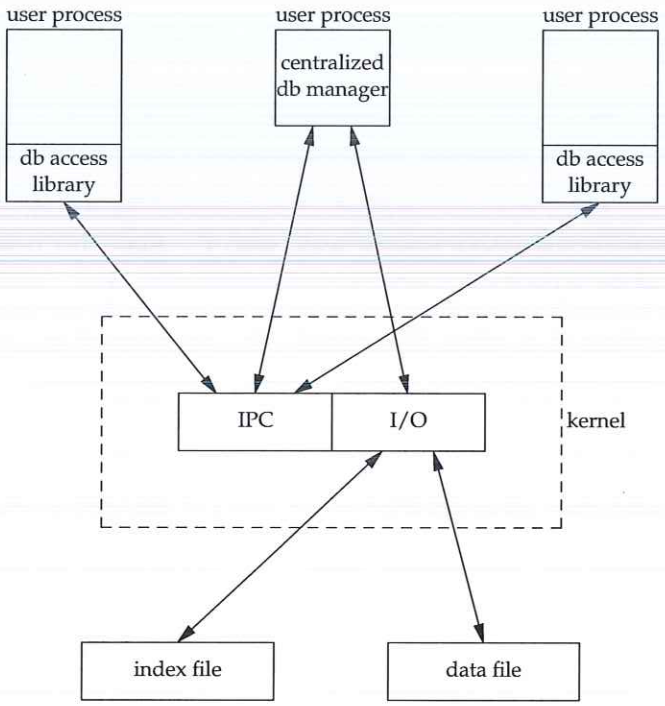

Figure 20.4 Centralized approach for database access

We purposely show the IPC going through the kernel, as most forms of message passing under the UNIX System operate this way. (Shared memory, as described in Section 15.9, avoids this copying of the data.) With the centralized approach, a record is read by the central process and then passed to the requesting process using IPC. This is a disadvantage of this design. Note that the centralized database manager is the only process that does I/O with the database files.

The centralized approach has the advantage that customer tuning of its operation may be possible. For example, we might be able to assign different priorities to different processes through the centralized process. This could affect the scheduling of I/O operations by the centralized process. With the decentralized approach, this is more difficult to do. We are usually at the mercy of the kernel's disk I/O scheduling policy and locking policy; that is, if three processes are waiting for a lock to become available, we cannot tell which process gets the lock next.

Another advantage of the centralized approach is that recovery is easier than with the decentralized approach. All the state information is in one place in the centralized approach, so if the database processes are killed, we have only one place to look to identify the outstanding transactions we need to resolve to restore the database to a consistent state.

The decentralized approach is shown in Figure 20.5. This is the design that we'll implement in this chapter.

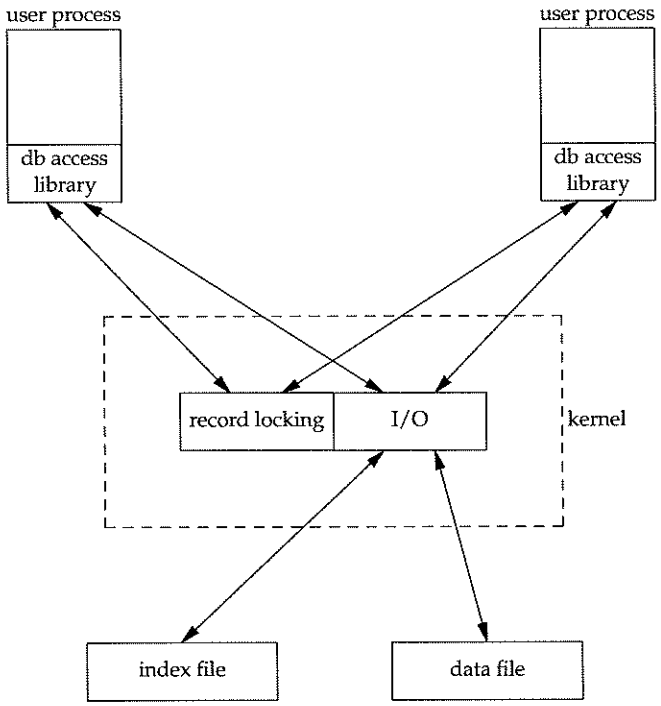

Figure 20.5 Decentralized approach for database access

The user processes that call the functions in the database library to perform I/O are considered cooperating processes, since they use byte-range locking to provide concurrent access.

20.6 Concurrency

We purposely chose a two-file implementation (an index file and a data file) because that is a common implementation technique (it simplifies space management in the files). It requires us to handle the locking interactions of both files. But there are numerous ways to handle the locking of these two files.

Coarse-Grained Locking

The simplest form of locking is to use one of the two files as a lock for the entire database and to require the caller to obtain this lock before operating on the database. We call this *coarse-grained locking*. For example, we can say that the process with a read lock on byte 0 of the index file has read access to the entire database. A process with a write lock on byte 0 of the index file has write access to the entire database. We can use the normal UNIX System byte-range locking semantics to allow any number of readers at one time, but only one writer at a time. (Recall Figure 14.3.) The functions `db_fetch` and `db_nextrec` require a read lock, and `db_delete`, `db_store`, and `db_open` all require a write lock. (The reason `db_open` requires a write lock is that if the file is being created, it has to write the empty free list and hash chains at the front of the index file.)

The problem with coarse-grained locking is that it limits concurrency. If a process is adding a record to one hash chain, another process should be able to read a record on a different hash chain.

Fine-Grained Locking

We enhance coarse-grained locking to allow more concurrency and call this *fine-grained locking*. We first require a reader or a writer to obtain a read lock or a write lock on the hash chain for a given record. We allow any number of readers at one time on any hash chain but only a single writer on a hash chain. Next, a writer needing to access the free list (either `db_delete` or `db_store`) must obtain a write lock on the free list. Finally, whenever it appends a new record to the end of either the index file or the data file, `db_store` has to obtain a write lock on that portion of the file.

We expect fine-grained locking to provide more concurrency than coarse-grained locking. In Section 20.9, we show some actual measurements. In Section 20.8, we show the source code for our implementation of fine-grained locking and discuss the details of implementing locking. (Coarse-grained locking is merely a simplification of the locking that we show.)

In the source code, we call `read`, `readv`, `write`, and `writev` directly. We do not use the standard I/O library. Although it is possible to use byte-range locking with the standard I/O library, careful handling of buffering is required. We don't want an

`fgets`, for example, to return data that was read into a standard I/O buffer 10 minutes ago if the data was modified by another process 5 minutes ago.

Our discussion of concurrency is predicated on the simple needs of the database library. Commercial systems often have additional requirements. See Chapter 16 of Date [2004] for additional details on concurrency.

20.7 Building the Library

The database library consists of two files: a public C header file and a C source file. We can build a static library using the commands

```
gcc -I../include -Wall -c db.c
ar rsv libapue_db.a db.o
```

Applications that want to link with `libapue_db.a` will also need to link with `libapue.a`, since we use some of our common functions in the database library.

If, on the other hand, we want to build a dynamic shared library version of the database library, we can use the following commands:

```
gcc -I../include -Wall -fPIC -c db.c
gcc -shared -Wl,-soname,libapue_db.so.1 -o libapue_db.so.1 \
    -L../lib -lapue -lc db.o
```

The resulting shared library, `libapue_db.so.1`, needs to be placed in a common directory where the dynamic linker/loader can find it. Alternatively, we can place it in a private directory and modify our `LD_LIBRARY_PATH` environment variable to include the private directory in the search path of the dynamic linker/loader.

> The steps used to build shared libraries vary among platforms. Here, we have shown how to do it on a Linux system with the GNU C compiler.

20.8 Source Code

We start by showing the `apue_db.h` header. This header is included by the library source code and all applications that call the library.

For the remainder of this text, we depart from the style of the previous examples in several ways. First, because the source code example is longer than usual, we number the lines. This makes it easier to link the discussion with the corresponding source code. Second, we place the description of the source code immediately below the source code on the same page.

> This style was inspired by John Lions in his book documenting the UNIX Version 6 operating system source code [Lions 1977, 1996]. It simplifies the task of studying large amounts of source code.

Note that we do not bother to number blank lines. Although this departs from the normal behavior of such tools as `pr`(1), we have nothing interesting to say about blank lines.

```
 1  #ifndef _APUE_DB_H
 2  #define _APUE_DB_H

 3  typedef    void *  DBHANDLE;

 4  DBHANDLE  db_open(const char *, int, ...);
 5  void      db_close(DBHANDLE);
 6  char     *db_fetch(DBHANDLE, const char *);
 7  int       db_store(DBHANDLE, const char *, const char *, int);
 8  int       db_delete(DBHANDLE, const char *);
 9  void      db_rewind(DBHANDLE);
10  char     *db_nextrec(DBHANDLE, char *);

11  /*
12   * Flags for db_store().
13   */
14  #define DB_INSERT     1    /* insert new record only */
15  #define DB_REPLACE    2    /* replace existing record */
16  #define DB_STORE      3    /* replace or insert */

17  /*
18   * Implementation limits.
19   */
20  #define IDXLEN_MIN    6    /* key, sep, start, sep, length, \n */
21  #define IDXLEN_MAX 1024    /* arbitrary */
22  #define DATLEN_MIN    2    /* data byte, newline */
23  #define DATLEN_MAX 1024    /* arbitrary */

24  #endif /* _APUE_DB_H */
```

[1–3] We use the `_APUE_DB_H` symbol to ensure that the contents of the header file are included only once. The `DBHANDLE` type represents an active reference to the database and is used to isolate applications from the implementation details of the database. Compare this technique with the way the standard I/O library exposes the `FILE` structure to applications.

[4–10] Next, we declare the prototypes for the database library's public functions. Since this header is included by applications that want to use the library, we don't declare the prototypes for the library's private functions here.

[11–24] The legal flags that can be passed to the `db_store` function are defined next, followed by fundamental limits of the implementation. These limits can be changed, if desired, to support bigger databases.

The minimum index record length is specified by `IDXLEN_MIN`. This represents a 1-byte key, a 1-byte separator, a 1-byte starting offset, another 1-byte separator, a 1-byte length, and a terminating newline character. (Recall the format of an index record from Figure 20.2.) An index record will usually be larger than `IDXLEN_MIN` bytes, but this is the bare minimum size.

The next file is db.c, the C source file for the library. For simplicity, we include all functions in a single file. This has the advantage that we can hide private functions by declaring them as static.

```
1   #include "apue.h"
2   #include "apue_db.h"
3   #include <fcntl.h>      /* open & db_open flags */
4   #include <stdarg.h>
5   #include <errno.h>
6   #include <sys/uio.h>    /* struct iovec */

7   /*
8    * Internal index file constants.
9    * These are used to construct records in the
10   * index file and data file.
11   */
12  #define IDXLEN_SZ     4    /* index record length (ASCII chars) */
13  #define SEP         ':'    /* separator char in index record */
14  #define SPACE       ' '    /* space character */
15  #define NEWLINE     '\n'   /* newline character */

16  /*
17   * The following definitions are for hash chains and free
18   * list chain in the index file.
19   */
20  #define PTR_SZ        7    /* size of ptr field in hash chain */
21  #define PTR_MAX 9999999    /* max file offset = 10**PTR_SZ - 1 */
22  #define NHASH_DEF   137    /* default hash table size */
23  #define FREE_OFF      0    /* free list offset in index file */
24  #define HASH_OFF PTR_SZ    /* hash table offset in index file */

25  typedef unsigned long  DBHASH; /* hash values */
26  typedef unsigned long  COUNT;  /* unsigned counter */
```

[1–6] We include apue.h because we use some of the functions from our private library. In turn, apue.h includes several standard header files, including <stdio.h> and <unistd.h>. We include <stdarg.h> because the db_open function uses the variable-argument functions declared by <stdarg.h>.

[7–26] The size of an index record is specified by IDXLEN_SZ. We use some characters, such as colon and newline, as delimiters in the database. We use the space character as "white out" when we delete a record.

Some of the values that we have defined as constants could also be made variable, with some added complexity in the implementation. For example, we set the size of the hash table to 137 entries. A better technique would be to let the caller specify this as an argument to db_open, based on the expected size of the database. We would then have to store this size at the beginning of the index file.

```
27  /*
28   * Library's private representation of the database.
29   */
30  typedef struct {
31    int    idxfd;  /* fd for index file */
32    int    datfd;  /* fd for data file */
33    char  *idxbuf; /* malloc'ed buffer for index record */
34    char  *datbuf; /* malloc'ed buffer for data record*/
35    char  *name;   /* name db was opened under */
36    off_t  idxoff; /* offset in index file of index record */
37                   /* key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
38    size_t idxlen; /* length of index record */
39                   /* excludes IDXLEN_SZ bytes at front of record */
40                   /* includes newline at end of index record */
41    off_t  datoff; /* offset in data file of data record */
42    size_t datlen; /* length of data record */
43                   /* includes newline at end */
44    off_t  ptrval; /* contents of chain ptr in index record */
45    off_t  ptroff; /* chain ptr offset pointing to this idx record */
46    off_t  chainoff; /* offset of hash chain for this index record */
47    off_t  hashoff;  /* offset in index file of hash table */
48    DBHASH nhash;    /* current hash table size */
49    COUNT  cnt_delok;    /* delete OK */
50    COUNT  cnt_delerr;   /* delete error */
51    COUNT  cnt_fetchok;  /* fetch OK */
52    COUNT  cnt_fetcherr; /* fetch error */
53    COUNT  cnt_nextrec;  /* nextrec */
54    COUNT  cnt_stor1;    /* store: DB_INSERT, no empty, appended */
55    COUNT  cnt_stor2;    /* store: DB_INSERT, found empty, reused */
56    COUNT  cnt_stor3;    /* store: DB_REPLACE, diff len, appended */
57    COUNT  cnt_stor4;    /* store: DB_REPLACE, same len, overwrote */
58    COUNT  cnt_storerr;  /* store error */
59  } DB;
```

[27–48] The DB structure is where we keep all the information for each open database. The DBHANDLE value that is returned by db_open and used by all the other functions is really just a pointer to one of these structures, but we hide that from the callers.

Since we store pointers and lengths as ASCII in the database, we convert these to numeric values and save them in the DB structure. We also save the hash table size even though it is fixed, just in case we decide to enhance the library to allow callers to specify the size when the database is created (see Exercise 20.7).

[49–59] The last ten fields in the DB structure count both successful and unsuccessful operations. If we want to analyze the performance of our database, we can write a function to return these statistics, but for now we only maintain the counters.

```
60  /*
61   * Internal functions.
62   */
63  static DB      *_db_alloc(int);
64  static void     _db_dodelete(DB *);
65  static int      _db_find_and_lock(DB *, const char *, int);
66  static int      _db_findfree(DB *, int, int);
67  static void     _db_free(DB *);
68  static DBHASH   _db_hash(DB *, const char *);
69  static char    *_db_readdat(DB *);
70  static off_t    _db_readidx(DB *, off_t);
71  static off_t    _db_readptr(DB *, off_t);
72  static void     _db_writedat(DB *, const char *, off_t, int);
73  static void     _db_writeidx(DB *, const char *, off_t, int, off_t);
74  static void     _db_writeptr(DB *, off_t, off_t);

75  /*
76   * Open or create a database.  Same arguments as open(2).
77   */
78  DBHANDLE
79  db_open(const char *pathname, int oflag, ...)
80  {
81      DB          *db;
82      int         len, mode;
83      size_t      i;
84      char        asciiptr[PTR_SZ + 1],
85                  hash[(NHASH_DEF + 1) * PTR_SZ + 2];
86                      /* +2 for newline and null */
87      struct stat statbuff;

88      /*
89       * Allocate a DB structure, and the buffers it needs.
90       */
91      len = strlen(pathname);
92      if ((db = _db_alloc(len)) == NULL)
93          err_dump("db_open: _db_alloc error for DB");
```

[60–74] We have chosen to name all the user-callable (public) functions starting with `db_` and all the internal (private) functions starting with `_db_`. The public functions were declared in the library's header file, `apue_db.h`. We declare the internal functions as `static` so they are visible only to functions residing in the same file (the file containing the library implementation).

[75–93] The `db_open` function has the same arguments as `open`(2). If the caller wants to create the database files, the optional third argument specifies the file permissions. The `db_open` function opens the index file and the data file, initializing the index file, if necessary. The function starts by calling `_db_alloc` to allocate and initialize a `DB` structure.

```
 94      db->nhash    = NHASH_DEF;/* hash table size */
 95      db->hashoff = HASH_OFF; /* offset in index file of hash table */
 96      strcpy(db->name, pathname);
 97      strcat(db->name, ".idx");

 98      if (oflag & O_CREAT) {
 99          va_list ap;

100          va_start(ap, oflag);
101          mode = va_arg(ap, int);
102          va_end(ap);

103          /*
104           * Open index file and data file.
105           */
106          db->idxfd = open(db->name, oflag, mode);
107          strcpy(db->name + len, ".dat");
108          db->datfd = open(db->name, oflag, mode);
109      } else {
110          /*
111           * Open index file and data file.
112           */
113          db->idxfd = open(db->name, oflag);
114          strcpy(db->name + len, ".dat");
115          db->datfd = open(db->name, oflag);
116      }

117      if (db->idxfd < 0 || db->datfd < 0) {
118          _db_free(db);
119          return(NULL);
120      }
```

[94–97] We continue to initialize the `DB` structure. The pathname passed in by the caller specifies the prefix of the database filenames. We append the suffix `.idx` to create the name for the database index file.

[98–108] If the caller wants to create the database files, we use the variable argument functions from `<stdarg.h>` to find the optional third argument. Then we use `open` to create and open the index file and data file. Note that the filename of the data file starts with the same prefix as the index file but has `.dat` as a suffix instead.

[109–116] If the caller doesn't specify the `O_CREAT` flag, then we're opening existing database files. In this case, we simply call `open` with two arguments.

[117–120] If an error occurs while we are opening or creating either database file, we call `_db_free` to clean up the `DB` structure and then return `NULL` to the caller. If one `open` succeeded and one failed, `_db_free` will take care of closing the open file descriptor, as we shall see shortly.

```
121     if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
122         /*
123          * If the database was created, we have to initialize
124          * it.  Write lock the entire file so that we can stat
125          * it, check its size, and initialize it, atomically.
126          */
127         if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
128             err_dump("db_open: writew_lock error");

129         if (fstat(db->idxfd, &statbuff) < 0)
130             err_sys("db_open: fstat error");

131         if (statbuff.st_size == 0) {
132             /*
133              * We have to build a list of (NHASH_DEF + 1) chain
134              * ptrs with a value of 0.  The +1 is for the free
135              * list pointer that precedes the hash table.
136              */
137             sprintf(asciiptr, "%*d", PTR_SZ, 0);
```

[121–130] We encounter locking if the database is being created. Consider two processes trying to create the same database at about the same time. Assume that the first process calls `fstat` and is blocked by the kernel after `fstat` returns. The second process calls `db_open`, finds that the length of the index file is 0, and initializes the free list and hash chain. The second process then writes one record to the database. At this point, the second process is blocked, and the first process continues executing right after the call to `fstat`. The first process finds the size of the index file to be 0 (since `fstat` was called before the second process initialized the index file), so the first process initializes the free list and hash chain, wiping out the record that the second process stored in the database. The way to prevent this is to use locking. We use the macros `readw_lock`, `writew_lock`, and `un_lock` from Section 14.3.

[131–137] If the size of the index file is 0, we have just created it, so we need to initialize the free list and hash chain pointers it contains. Note that we use the format string `%*d` to convert a database pointer from an integer to an ASCII string. (We'll use this type of format again in `_db_writeidx` and `_db_writeptr`.) This format tells `sprintf` to take the `PTR_SZ` argument and use it as the minimum field width for the next argument, which is 0 in this instance (here we are initializing the pointers to 0, since we are creating a new database). This has the effect of forcing the string created to be at least `PTR_SZ` characters (padded on the left with spaces). In `_db_writeidx` and `_db_writeptr`, we will pass a pointer value instead of zero, but we will first verify that the pointer value isn't greater than `PTR_MAX`, to guarantee that every pointer string we write to the database occupies exactly `PTR_SZ` (7) characters.

```
138             hash[0] = 0;
139             for (i = 0; i < NHASH_DEF + 1; i++)
140                 strcat(hash, asciiptr);
141             strcat(hash, "\n");
142             i = strlen(hash);
143             if (write(db->idxfd, hash, i) != i)
144                 err_dump("db_open: index file init write error");
145         }
146         if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
147             err_dump("db_open: un_lock error");
148     }
149     db_rewind(db);
150     return(db);
151 }
152 /*
153  * Allocate & initialize a DB structure and its buffers.
154  */
155 static DB *
156 _db_alloc(int namelen)
157 {
158     DB      *db;
159     /*
160      * Use calloc, to initialize the structure to zero.
161      */
162     if ((db = calloc(1, sizeof(DB))) == NULL)
163         err_dump("_db_alloc: calloc error for DB");
164     db->idxfd = db->datfd = -1;             /* descriptors */
165     /*
166      * Allocate room for the name.
167      * +5 for ".idx" or ".dat" plus null at end.
168      */
169     if ((db->name = malloc(namelen + 5)) == NULL)
170         err_dump("_db_alloc: malloc error for name");
```

[138–151] We continue to initialize the newly created database. We build the hash table and write it to the index file. Then we unlock the index file, reset the database file pointers, and return a pointer to the DB structure as the opaque handle for the caller to use with the other database functions.

[152–164] The _db_alloc function is called by db_open to allocate storage for the DB structure, an index buffer, and a data buffer. We use calloc to allocate memory to hold the DB structure and ensure that it is initialized to all zeros. Since this has the side effect of setting the database file descriptors to zero, we need to reset them to –1 to indicate that they are not yet valid.

[165–170] We allocate space to hold the name of the database file. We use this buffer to create both filenames by changing the suffix to refer to either the index file or the data file, as we saw in db_open.

```
171     /*
172      * Allocate an index buffer and a data buffer.
173      * +2 for newline and null at end.
174      */
175     if ((db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
176         err_dump("_db_alloc: malloc error for index buffer");
177     if ((db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
178         err_dump("_db_alloc: malloc error for data buffer");
179     return(db);
180   }
181   /*
182    * Relinquish access to the database.
183    */
184   void
185   db_close(DBHANDLE h)
186   {
187     _db_free((DB *)h);  /* closes fds, free buffers & struct */
188   }
189   /*
190    * Free up a DB structure, and all the malloc'ed buffers it
191    * may point to.  Also close the file descriptors if still open.
192    */
193   static void
194   _db_free(DB *db)
195   {
196     if (db->idxfd >= 0)
197         close(db->idxfd);
198     if (db->datfd >= 0)
199         close(db->datfd);
```

[171–180] We allocate space for buffers for the index and data files. The buffer sizes are defined in `apue_db.h`. An enhancement to the database library would be to allow these buffers to expand as required. We could keep track of the size of these two buffers and call `realloc` whenever we find we need a bigger buffer. Finally, we return a pointer to the `DB` structure that we allocated.

[181–188] The `db_close` function is a wrapper that casts a database handle to a `DB` structure pointer, passing it to `_db_free` to release any resources and free the `DB` structure.

[189–199] The `_db_free` function is called by `db_open` if an error occurs while opening the index file or data file and is also called by `db_close` when an application is done using the database. If the file descriptor for the database index file is valid, we close it. The same is done with the file descriptor for the data file. (Recall that when we allocate a new `DB` structure in `_db_alloc`, we initialize each file descriptor to –1. If we are unable to open one of the database files, the corresponding file descriptor will still be set to –1, and we will avoid trying to close it.)

```
200     if (db->idxbuf != NULL)
201         free(db->idxbuf);
202     if (db->datbuf != NULL)
203         free(db->datbuf);
204     if (db->name != NULL)
205         free(db->name);
206     free(db);
207   }

208   /*
209    * Fetch a record.  Return a pointer to the null-terminated data.
210    */
211   char *
212   db_fetch(DBHANDLE h, const char *key)
213   {
214     DB      *db = h;
215     char    *ptr;

216     if (_db_find_and_lock(db, key, 0) < 0) {
217         ptr = NULL;             /* error, record not found */
218         db->cnt_fetcherr++;
219     } else {
220         ptr = _db_readdat(db);  /* return pointer to data */
221         db->cnt_fetchok++;
222     }

223     /*
224      * Unlock the hash chain that _db_find_and_lock locked.
225      */
226     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
227         err_dump("db_fetch: un_lock error");
228     return(ptr);
229   }
```

[200–207] Next, we free any dynamically allocated buffers. We can safely pass a null pointer to `free`, so we don't need to check the value of each buffer pointer beforehand, but we do so anyway because we consider it better style to free only those objects that we allocated. (Not all deallocator functions are as forgiving as `free`.) Finally, we free the memory backing the `DB` structure.

[208–218] The `db_fetch` function is used to read a record given its key. We first try to find the record by calling `_db_find_and_lock`. If the record can't be found, we set the return value (`ptr`) to `NULL` and increment the count of unsuccessful record searches. Because `_db_find_and_lock` returns with the database index file locked, we can't return until we unlock it.

[219–229] If the record is found, we call `_db_readdat` to read the corresponding data record and increment the count of the successful record searches. Before returning, we unlock the index file by calling `un_lock`. Then we return a pointer to the record found (or `NULL` if the record wasn't found).

```
230   /*
231    * Find the specified record.  Called by db_delete, db_fetch,
232    * and db_store.  Returns with the hash chain locked.
233    */
234   static int
235   _db_find_and_lock(DB *db, const char *key, int writelock)
236   {
237     off_t   offset, nextoffset;
238     /*
239      * Calculate the hash value for this key, then calculate the
240      * byte offset of corresponding chain ptr in hash table.
241      * This is where our search starts.  First we calculate the
242      * offset in the hash table for this key.
243      */
244     db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
245     db->ptroff = db->chainoff;
246     /*
247      * We lock the hash chain here.  The caller must unlock it
248      * when done.  Note we lock and unlock only the first byte.
249      */
250     if (writelock) {
251         if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
252             err_dump("_db_find_and_lock: writew_lock error");
253     } else {
254         if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
255             err_dump("_db_find_and_lock: readw_lock error");
256     }
257     /*
258      * Get the offset in the index file of first record
259      * on the hash chain (can be 0).
260      */
261     offset = _db_readptr(db, db->ptroff);
```

[230–237] The `_db_find_and_lock` function is used internally by the library to find a record given its key. We set the `writelock` parameter to a nonzero value if we want to acquire a write lock on the index file while we search for the record. If we set `writelock` to zero, we read lock the index file while we search it.

[238–256] We prepare to traverse a hash chain in `_db_find_and_lock`. We convert the key into a hash value, which we use to calculate the starting address of the hash chain in the file (`chainoff`). We wait for the lock to be granted before going through the hash chain. Note that we lock only the first byte in the start of the hash chain. This increases concurrency by allowing multiple processes to search different hash chains at the same time.

[257–261] We call `_db_readptr` to read the first pointer in the hash chain. If this returns zero, the hash chain is empty.

```
262     while (offset != 0) {
263         nextoffset = _db_readidx(db, offset);
264         if (strcmp(db->idxbuf, key) == 0)
265             break;          /* found a match */
266         db->ptroff = offset; /* offset of this (unequal) record */
267         offset = nextoffset; /* next one to compare */
268     }
269     /*
270      * offset == 0 on error (record not found).
271      */
272     return(offset == 0 ? -1 : 0);
273 }

274 /*
275  * Calculate the hash value for a key.
276  */
277 static DBHASH
278 _db_hash(DB *db, const char *key)
279 {
280     DBHASH      hval = 0;
281     char        c;
282     int         i;

283     for (i = 1; (c = *key++) != 0; i++)
284         hval += c * i;      /* ascii char times its 1-based index */
285     return(hval % db->nhash);
286 }
```

[262–268] In the `while` loop, we go through each index record on the hash chain, comparing keys. We call `_db_readidx` to read each index record. It populates the `idxbuf` field with the key of the current record. If `_db_readidx` returns zero, we've reached the last entry in the chain.

[269–273] If `offset` is zero after the loop, we've reached the end of a hash chain without finding a matching key, so we return –1. Otherwise, we found a match (and exited the loop with the `break` statement), so we return success (0). In this case, the `ptroff` field contains the address of the previous index record, `datoff` contains the address of the data record, and `datlen` contains the size of the data record. As we make our way through the hash chain, we save the previous index record that points to the current index record. We'll use this when we delete a record, since we have to modify the chain pointer of the previous record to delete the current record.

[274–286] `_db_hash` calculates the hash value for a given key. It multiplies each ASCII character times its 1-based index and divides the result by the number of hash table entries. The remainder from the division is the hash value for this key. Recall that the number of hash table entries is 137, which is a prime number. According to Knuth [1998], prime hashes generally provide good distribution characteristics.

```
287   /*
288    * Read a chain ptr field from anywhere in the index file:
289    * the free list pointer, a hash table chain ptr, or an
290    * index record chain ptr.
291    */
292   static off_t
293   _db_readptr(DB *db, off_t offset)
294   {
295     char    asciiptr[PTR_SZ + 1];

296     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
297         err_dump("_db_readptr: lseek error to ptr field");
298     if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
299         err_dump("_db_readptr: read error of ptr field");
300     asciiptr[PTR_SZ] = 0;       /* null terminate */
301     return(atol(asciiptr));
302   }

303   /*
304    * Read the next index record.  We start at the specified offset
305    * in the index file.  We read the index record into db->idxbuf
306    * and replace the separators with null bytes.  If all is OK we
307    * set db->datoff and db->datlen to the offset and length of the
308    * corresponding data record in the data file.
309    */
310   static off_t
311   _db_readidx(DB *db, off_t offset)
312   {
313     ssize_t             i;
314     char            *ptr1, *ptr2;
315     char            asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
316     struct iovec    iov[2];
```

[287–302] `_db_readptr` reads any one of three different chain pointers: (a) the pointer at the beginning of the index file that points to the first index record on the free list, (b) the pointers in the hash table that point to the first index record on each hash chain, and (c) the pointers that are stored at the beginning of each index record (whether the index record is part of a hash chain or on the free list). We convert the pointer from ASCII to a long integer before returning it. No locking is done by this function; that is up to the caller.

[303–316] The `_db_readidx` function is used to read the record at the specified offset from the index file. On success, the function will return the offset of the next record in the list. In this case, the function will populate several fields in the `DB` structure: `idxoff` contains the offset of the current record in the index file, `ptrval` contains the offset of the next index entry in the list, `idxlen` contains the length of the current index record, `idxbuf` contains the actual index record, `datoff` contains the offset of the record in the data file, and `datlen` contains the length of the data record.

```
317     /*
318      * Position index file and record the offset.  db_nextrec
319      * calls us with offset==0, meaning read from current offset.
320      * We still need to call lseek to record the current offset.
321      */
322     if ((db->idxoff = lseek(db->idxfd, offset,
323       offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
324         err_dump("_db_readidx: lseek error");

325     /*
326      * Read the ascii chain ptr and the ascii length at
327      * the front of the index record.  This tells us the
328      * remaining size of the index record.
329      */
330     iov[0].iov_base = asciiptr;
331     iov[0].iov_len  = PTR_SZ;
332     iov[1].iov_base = asciilen;
333     iov[1].iov_len  = IDXLEN_SZ;
334     if ((i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
335         if (i == 0 && offset == 0)
336             return(-1);     /* EOF for db_nextrec */
337         err_dump("_db_readidx: readv error of index record");
338     }

339     /*
340      * This is our return value; always >= 0.
341      */
342     asciiptr[PTR_SZ] = 0;         /* null terminate */
343     db->ptrval = atol(asciiptr); /* offset of next key in chain */

344     asciilen[IDXLEN_SZ] = 0;      /* null terminate */
345     if ((db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
346       db->idxlen > IDXLEN_MAX)
347         err_dump("_db_readidx: invalid length");
```

[317–324] We start by seeking to the index file offset provided by the caller. We record the offset in the DB structure, so even if the caller wants to read the record at the current file offset (by setting `offset` to 0), we still need to call `lseek` to determine the current offset. Since an index record will never be stored at offset 0 in the index file, we can safely overload the value of 0 to mean "read from the current offset."

[325–338] We call `readv` to read the two fixed-length fields at the beginning of the index record: the chain pointer to the next index record and the size of the variable-length index record that follows.

[339–347] We convert the offset of the next record to an integer and store it in the `ptrval` field (this will be used as the return value for this function). Then we convert the length of the index record into an integer and save it in the `idxlen` field.

```
348     /*
349      * Now read the actual index record.  We read it into the key
350      * buffer that we malloced when we opened the database.
351      */
352     if ((i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
353         err_dump("_db_readidx: read error of index record");
354     if (db->idxbuf[db->idxlen-1] != NEWLINE)    /* sanity check */
355         err_dump("_db_readidx: missing newline");
356     db->idxbuf[db->idxlen-1] = 0;    /* replace newline with null */
357     /*
358      * Find the separators in the index record.
359      */
360     if ((ptr1 = strchr(db->idxbuf, SEP)) == NULL)
361         err_dump("_db_readidx: missing first separator");
362     *ptr1++ = 0;                /* replace SEP with null */
363     if ((ptr2 = strchr(ptr1, SEP)) == NULL)
364         err_dump("_db_readidx: missing second separator");
365     *ptr2++ = 0;                /* replace SEP with null */
366     if (strchr(ptr2, SEP) != NULL)
367         err_dump("_db_readidx: too many separators");
368     /*
369      * Get the starting offset and length of the data record.
370      */
371     if ((db->datoff = atol(ptr1)) < 0)
372         err_dump("_db_readidx: starting offset < 0");
373     if ((db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
374         err_dump("_db_readidx: invalid length");
375     return(db->ptrval);     /* return offset of next key in chain */
376 }
```

[348–356] We read the variable-length index record into the `idxbuf` field in the `DB` structure. The record should be terminated with a newline, which we replace with a null byte. If the index file is corrupt, we terminate and drop core by calling `err_dump`.

[357–367] We separate the index record into three fields: the key, the offset of the corresponding data record, and the length of the data record. The `strchr` function finds the first occurrence of the specified character in the given string. Here we look for the character that separates fields in the record (`SEP`, which we define to be a colon).

[368–376] We convert the data record offset and length into integers and store them in the `DB` structure. Then we return the offset of the next record in the hash chain. Note that we do not read the data record; that task is left to the caller. In the `db_fetch` function, for example, we don't read the data record until `_db_find_and_lock` has read the index record that matches the key that we're looking for.

```
377  /*
378   * Read the current data record into the data buffer.
379   * Return a pointer to the null-terminated data buffer.
380   */
381  static char *
382  _db_readdat(DB *db)
383  {
384    if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
385        err_dump("_db_readdat: lseek error");
386    if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
387        err_dump("_db_readdat: read error");
388    if (db->datbuf[db->datlen-1] != NEWLINE)    /* sanity check */
389        err_dump("_db_readdat: missing newline");
390    db->datbuf[db->datlen-1] = 0; /* replace newline with null */
391    return(db->datbuf);     /* return pointer to data record */
392  }
393  /*
394   * Delete the specified record.
395   */
396  int
397  db_delete(DBHANDLE h, const char *key)
398  {
399    DB      *db = h;
400    int     rc = 0;         /* assume record will be found */
401    if (_db_find_and_lock(db, key, 1) == 0) {
402        _db_dodelete(db);
403        db->cnt_delok++;
404    } else {
405        rc = -1;            /* not found */
406        db->cnt_delerr++;
407    }
408    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
409        err_dump("db_delete: un_lock error");
410    return(rc);
411  }
```

[377–392] The `_db_readdat` function populates the `datbuf` field in the `DB` structure with the contents of the data record, expecting that the `datoff` and `datlen` fields will have been properly initialized already.

[393–411] The `db_delete` function is used to delete a record given its key. We use `_db_find_and_lock` to determine whether the record exists in the database. If it does, we call `_db_dodelete` to do the work needed to delete the record. The third argument to `_db_find_and_lock` controls whether the chain is read locked or write locked. Here we are requesting a write lock, since we will potentially change the list. Since `_db_find_and_lock` returns with the lock still held, we need to unlock it, regardless of whether the record was found.

```
412  /*
413   * Delete the current record specified by the DB structure.
414   * This function is called by db_delete and db_store, after
415   * the record has been located by _db_find_and_lock.
416   */
417  static void
418  _db_dodelete(DB *db)
419  {
420    int     i;
421    char    *ptr;
422    off_t   freeptr, saveptr;

423    /*
424     * Set data buffer and key to all blanks.
425     */
426    for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
427        *ptr++ = SPACE;
428    *ptr = 0;   /* null terminate for _db_writedat */
429    ptr = db->idxbuf;
430    while (*ptr)
431        *ptr++ = SPACE;

432    /*
433     * We have to lock the free list.
434     */
435    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
436        err_dump("_db_dodelete: writew_lock error");

437    /*
438     * Write the data record with all blanks.
439     */
440    _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);
```

[412–431] The `_db_dodelete` function does all the work necessary to delete a record from the database. (This function is also called by `db_store`.) Most of the function just updates two linked lists: the free list and the hash chain for this key. When a record is deleted, we set its key and data record to blanks. This fact is used by `db_nextrec`, which we'll examine later in this section.

[432–440] We call `writew_lock` to write lock the free list. This step prevents two processes that are deleting records at the same time, on two different hash chains, from interfering with each other. Since we'll add the deleted record to the free list, which changes the free-list pointer, only one process at a time can be doing this.

We write the all-blank data record by calling `_db_writedat`. Note that there is no need for `_db_writedat` to lock the data file in this case. Since `db_delete` has write locked the hash chain for this record, we know that no other process is reading or writing this particular data record.

```
441     /*
442      * Read the free list pointer.  Its value becomes the
443      * chain ptr field of the deleted index record.  This means
444      * the deleted record becomes the head of the free list.
445      */
446     freeptr = _db_readptr(db, FREE_OFF);
447     /*
448      * Save the contents of index record chain ptr,
449      * before it's rewritten by _db_writeidx.
450      */
451     saveptr = db->ptrval;
452     /*
453      * Rewrite the index record.  This also rewrites the length
454      * of the index record, the data offset, and the data length,
455      * none of which has changed, but that's OK.
456      */
457     _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);
458     /*
459      * Write the new free list pointer.
460      */
461     _db_writeptr(db, FREE_OFF, db->idxoff);
462     /*
463      * Rewrite the chain ptr that pointed to this record being
464      * deleted.  Recall that _db_find_and_lock sets db->ptroff to
465      * point to this chain ptr.  We set this chain ptr to the
466      * contents of the deleted record's chain ptr, saveptr.
467      */
468     _db_writeptr(db, db->ptroff, saveptr);
469     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
470         err_dump("_db_dodelete: un_lock error");
471 }
```

[441–461] We read the free-list pointer and then update the index record so that its next record pointer is set to the first record on the free list. (If the free list was empty, this new chain pointer is 0.) We have already cleared the key. Then we update the free-list pointer with the offset of the index record we are deleting. This means that the free list is handled on a last-in, first-out basis; that is, deleted records are added to the front of the free list (although we remove entries from the free list on a first-fit basis).

We don't have a separate free list for each file. When we add a deleted index record to the free list, the index record still points to the deleted data record. There are better ways to do this, in exchange for added complexity.

[462–471] We update the previous record in the hash chain to point to the record after the one we are deleting, thus removing the deleted record from the hash chain. Finally, we unlock the free list.

```
472  /*
473   * Write a data record.  Called by _db_dodelete (to write
474   * the record with blanks) and db_store.
475   */
476  static void
477  _db_writedat(DB *db, const char *data, off_t offset, int whence)
478  {
479    struct iovec    iov[2];
480    static char     newline = NEWLINE;

481    /*
482     * If we're appending, we have to lock before doing the lseek
483     * and write to make the two an atomic operation.  If we're
484     * overwriting an existing record, we don't have to lock.
485     */
486    if (whence == SEEK_END) /* we're appending, lock entire file */
487        if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
488            err_dump("_db_writedat: writew_lock error");

489    if ((db->datoff = lseek(db->datfd, offset, whence)) == -1)
490        err_dump("_db_writedat: lseek error");
491    db->datlen = strlen(data) + 1;  /* datlen includes newline */

492    iov[0].iov_base = (char *) data;
493    iov[0].iov_len  = db->datlen - 1;
494    iov[1].iov_base = &newline;
495    iov[1].iov_len  = 1;
496    if (writev(db->datfd, &iov[0], 2) != db->datlen)
497        err_dump("_db_writedat: writev error of data record");

498    if (whence == SEEK_END)
499        if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
500            err_dump("_db_writedat: un_lock error");
501  }
```

[472–491] We call `_db_writedat` to write a data record. When we delete a record, we use `_db_writedat` to overwrite the record with blanks; `_db_writedat` doesn't need to lock the data file, because `db_delete` has write locked the hash chain for this record. Thus, no other process could be reading or writing this particular data record. When we cover `db_store` later in this section, we'll encounter the case in which `_db_writedat` is appending to the data file and has to lock it.

We seek to the location where we want to write the data record. The amount to write is the record size plus 1 byte for the terminating newline we add.

[492–501] We set up the `iovec` array and call `writev` to write the data record and newline. We can't assume that the caller's buffer has room at the end for us to append the newline, so we write the newline from a separate buffer. If we are appending a record to the file, we release the lock we acquired earlier.

```
502   /*
503    * Write an index record.  _db_writedat is called before
504    * this function to set the datoff and datlen fields in the
505    * DB structure, which we need to write the index record.
506    */
507   static void
508   _db_writeidx(DB *db, const char *key,
509                off_t offset, int whence, off_t ptrval)
510   {
511     struct iovec    iov[2];
512     char            asciiptrlen[PTR_SZ + IDXLEN_SZ + 1];
513     int             len;

514     if ((db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
515         err_quit("_db_writeidx: invalid ptr: %d", ptrval);
516     sprintf(db->idxbuf, "%s%c%lld%c%ld\n", key, SEP,
517       (long long)db->datoff, SEP, (long)db->datlen);
518     len = strlen(db->idxbuf);
519     if (len < IDXLEN_MIN || len > IDXLEN_MAX)
520         err_dump("_db_writeidx: invalid length");
521     sprintf(asciiptrlen, "%*lld%*d", PTR_SZ, (long long)ptrval,
522       IDXLEN_SZ, len);

523     /*
524      * If we're appending, we have to lock before doing the lseek
525      * and write to make the two an atomic operation.  If we're
526      * overwriting an existing record, we don't have to lock.
527      */
528     if (whence == SEEK_END)     /* we're appending */
529         if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
530           SEEK_SET, 0) < 0)
531             err_dump("_db_writeidx: writew_lock error");
```

[502–522] The `_db_writeidx` function is called to write an index record. After validating the next pointer in the chain, we create the index record and store the second half of it in `idxbuf`. We need the size of this portion of the index record to create the first half of the index record, which we store in the local variable `asciiptrlen`.

Note that we use casts to force the size of the arguments in the `sprintf` statements to match the format specifications. This is because the size of the `off_t` and `size_t` data types can vary among platforms. Even a 32-bit system can provide 64-bit file offsets, so we can't make any assumptions about the size of the `off_t` data type.

[523–531] As with `_db_writedat`, this function deals with locking only when a new index record is being appended to the index file. When `_db_dodelete` calls this function, we're rewriting an existing index record. In this case, the caller has write locked the hash chain, so no additional locking is required.

```
532     /*
533      * Position the index file and record the offset.
534      */
535     if ((db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
536         err_dump("_db_writeidx: lseek error");

537     iov[0].iov_base = asciiptrlen;
538     iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
539     iov[1].iov_base = db->idxbuf;
540     iov[1].iov_len  = len;
541     if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
542         err_dump("_db_writeidx: writev error of index record");

543     if (whence == SEEK_END)
544         if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
545           SEEK_SET, 0) < 0)
546             err_dump("_db_writeidx: un_lock error");
547 }

548 /*
549  * Write a chain ptr field somewhere in the index file:
550  * the free list, the hash table, or in an index record.
551  */
552 static void
553 _db_writeptr(DB *db, off_t offset, off_t ptrval)
554 {
555   char    asciiptr[PTR_SZ + 1];

556     if (ptrval < 0 || ptrval > PTR_MAX)
557         err_quit("_db_writeptr: invalid ptr: %d", ptrval);
558     sprintf(asciiptr, "%*lld", PTR_SZ, (long long)ptrval);

559     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
560         err_dump("_db_writeptr: lseek error to ptr field");
561     if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
562         err_dump("_db_writeptr: write error of ptr field");
563 }
```

[532–547] We seek to the location where we want to write the index record and save this offset in the `idxoff` field of the `DB` structure. Since we built the index record in two separate buffers, we use `writev` to store it in the index file. If we were appending to the file, we release the lock we acquired before seeking. This makes the seek and the write an atomic operation from the perspective of concurrently running processes appending new records to the same database.

[548–563] `_db_writeptr` is used to write a chain pointer to the index file. We validate that the chain pointer is within bounds, then convert it to an ASCII string. We seek to the specified offset in the index file and write the pointer.

```
564   /*
565    * Store a record in the database.  Return 0 if OK, 1 if record
566    * exists and DB_INSERT specified, -1 on error.
567    */
568   int
569   db_store(DBHANDLE h, const char *key, const char *data, int flag)
570   {
571     DB      *db = h;
572     int     rc, keylen, datlen;
573     off_t   ptrval;
574     if (flag != DB_INSERT && flag != DB_REPLACE &&
575       flag != DB_STORE) {
576         errno = EINVAL;
577         return(-1);
578     }
579     keylen = strlen(key);
580     datlen = strlen(data) + 1;      /* +1 for newline at end */
581     if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
582         err_dump("db_store: invalid data length");
583     /*
584      * _db_find_and_lock calculates which hash table this new record
585      * goes into (db->chainoff), regardless of whether it already
586      * exists or not. The following calls to _db_writeptr change the
587      * hash table entry for this chain to point to the new record.
588      * The new record is added to the front of the hash chain.
589      */
590     if (_db_find_and_lock(db, key, 1) < 0) { /* record not found */
591         if (flag == DB_REPLACE) {
592             rc = -1;
593             db->cnt_storerr++;
594             errno = ENOENT;     /* error, record does not exist */
595             goto doreturn;
596         }
```

[564–582] We use `db_store` to add a record to the database. We first validate the flag value we are passed. Then we make sure that the length of the data record is valid. If it isn't, we drop core and exit. This is OK for an example, but if we were building a production-quality library, we'd return an error status instead, which would give the application a chance to recover.

[583–596] We call `_db_find_and_lock` to see if the record already exists. It is OK if the record doesn't exist and either `DB_INSERT` or `DB_STORE` is specified, or if the record already exists and either `DB_REPLACE` or `DB_STORE` is specified. Replacing an existing record implies that the keys are identical but that the data records probably differ. Note that the final argument to `_db_find_and_lock` specifies that the hash chain must be write locked, as we will probably be modifying this hash chain.

```
597         /*
598          * _db_find_and_lock locked the hash chain for us; read
599          * the chain ptr to the first index record on hash chain.
600          */
601         ptrval = _db_readptr(db, db->chainoff);

602         if (_db_findfree(db, keylen, datlen) < 0) {
603             /*
604              * Can't find an empty record big enough. Append the
605              * new record to the ends of the index and data files.
606              */
607             _db_writedat(db, data, 0, SEEK_END);
608             _db_writeidx(db, key, 0, SEEK_END, ptrval);

609             /*
610              * db->idxoff was set by _db_writeidx.  The new
611              * record goes to the front of the hash chain.
612              */
613             _db_writeptr(db, db->chainoff, db->idxoff);
614             db->cnt_stor1++;
615         } else {
616             /*
617              * Reuse an empty record. _db_findfree removed it from
618              * the free list and set both db->datoff and db->idxoff.
619              * Reused record goes to the front of the hash chain.
620              */
621             _db_writedat(db, data, db->datoff, SEEK_SET);
622             _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
623             _db_writeptr(db, db->chainoff, db->idxoff);
624             db->cnt_stor2++;
625         }
```

[597–601] After we call `_db_find_and_lock`, the code divides into four cases. In the first two, no record was found, so we are adding a new record. We read the offset of the first entry on the hash list.

[602–614] Case 1: we call `_db_findfree` to search the free list for a deleted record with the same size key and same size data. If no such record is found, we have to append the new record to the ends of the index and data files. We call `_db_writedat` to write the data part, `_db_writeidx` to write the index part, and `_db_writeptr` to place the new record on the front of the hash chain. We increment a count (`cnt_stor1`) of the number of times we executed this case to allow us to characterize the behavior of the database.

[615–625] Case 2: `_db_findfree` found an empty record with the correct sizes and removed it from the free list (we'll see the implementation of `_db_findfree` shortly). We write the data and index portions of the new record and add the record to the front of the hash chain as we did in case 1. The `cnt_stor2` field counts how many times we've executed this case.

```
626     } else {                        /* record found */
627         if (flag == DB_INSERT) {
628             rc = 1;     /* error, record already in db */
629             db->cnt_storerr++;
630             goto doreturn;
631         }
632         /*
633          * We are replacing an existing record.  We know the new
634          * key equals the existing key, but we need to check if
635          * the data records are the same size.
636          */
637         if (datlen != db->datlen) {
638             _db_dodelete(db);   /* delete the existing record */
639             /*
640              * Reread the chain ptr in the hash table
641              * (it may change with the deletion).
642              */
643             ptrval = _db_readptr(db, db->chainoff);
644             /*
645              * Append new index and data records to end of files.
646              */
647             _db_writedat(db, data, 0, SEEK_END);
648             _db_writeidx(db, key, 0, SEEK_END, ptrval);
649             /*
650              * New record goes to the front of the hash chain.
651              */
652             _db_writeptr(db, db->chainoff, db->idxoff);
653             db->cnt_stor3++;
654         } else {
```

[626–631] Now we reach the two cases in which a record with the same key already exists in the database. If the caller isn't replacing the record, we set the return code to indicate that a record exists, increment the count of the number of store errors, and jump to the end of the function, where we handle the common return logic.

[632–654] Case 3: an existing record is being replaced, and the length of the new data record differs from the length of the existing one. We call `_db_dodelete` to delete the existing record. Recall that this places the deleted record at the head of the free list. Then we append the new record to the ends of the data and index files by calling `_db_writedat` and `_db_writeidx`. (There are other ways to handle this case. We could try to find a deleted record that has the correct data size.) The new record is added to the front of the hash chain by calling `_db_writeptr`. The `cnt_stor3` counter in the `DB` structure records the number of times we've executed this case.

```
655             /*
656              * Same size data, just replace data record.
657              */
658             _db_writedat(db, data, db->datoff, SEEK_SET);
659             db->cnt_stor4++;
660         }
661     }
662     rc = 0;     /* OK */
663   doreturn: /* unlock hash chain locked by _db_find_and_lock */
664     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
665         err_dump("db_store: un_lock error");
666     return(rc);
667   }
668   /*
669    * Try to find a free index record and accompanying data record
670    * of the correct sizes.  We're only called by db_store.
671    */
672   static int
673   _db_findfree(DB *db, int keylen, int datlen)
674   {
675     int     rc;
676     off_t   offset, nextoffset, saveoffset;
677     /*
678      * Lock the free list.
679      */
680     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
681         err_dump("_db_findfree: writew_lock error");
682     /*
683      * Read the free list pointer.
684      */
685     saveoffset = FREE_OFF;
686     offset = _db_readptr(db, saveoffset);
```

[655–661] Case 4: An existing record is being replaced, and the length of the new data record equals the length of the existing data record. This is the easiest case; we simply rewrite the data record and increment the counter (`cnt_stor4`) for this case.

[662–667] In the normal case, we set the return code to indicate success and fall through to the common return logic. We unlock the hash chain that was locked as a result of calling `_db_find_and_lock` and return to the caller.

[668–686] The `_db_findfree` function tries to find a free index record and associated data record of the specified sizes. We need to write lock the free list to avoid interfering with any other processes using the free list. After locking the free list, we get the pointer address at the head of the list.

```
687     while (offset != 0) {
688         nextoffset = _db_readidx(db, offset);
689         if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
690             break;      /* found a match */
691         saveoffset = offset;
692         offset = nextoffset;
693     }
694     if (offset == 0) {
695         rc = -1;    /* no match found */
696     } else {
697         /*
698          * Found a free record with matching sizes.
699          * The index record was read in by _db_readidx above,
700          * which sets db->ptrval.  Also, saveoffset points to
701          * the chain ptr that pointed to this empty record on
702          * the free list.  We set this chain ptr to db->ptrval,
703          * which removes the empty record from the free list.
704          */
705         _db_writeptr(db, saveoffset, db->ptrval);
706         rc = 0;
707         /*
708          * Notice also that _db_readidx set both db->idxoff
709          * and db->datoff.  This is used by the caller, db_store,
710          * to write the new index record and data record.
711          */
712     }
713     /*
714      * Unlock the free list.
715      */
716     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
717         err_dump("_db_findfree: un_lock error");
718     return(rc);
719 }
```

[687–693] The `while` loop in `_db_findfree` goes through the free list, looking for a record with matching key and data sizes. In this simple implementation, we reuse a deleted record only if the key length and data length equal the lengths for the new record being inserted. There are a variety of better ways to reuse this deleted space, in exchange for added complexity.

[694–712] If we can't find an available record of the requested key and data sizes, we set the return code to indicate failure. Otherwise, we write the previous record's chain pointer to point to the next chain pointer value of the record we have found. This removes the record from the free list.

[713–719] Once we are done with the free list, we release the write lock. Then we return the status to the caller.

```
720   /*
721    * Rewind the index file for db_nextrec.
722    * Automatically called by db_open.
723    * Must be called before first db_nextrec.
724    */
725   void
726   db_rewind(DBHANDLE h)
727   {
728     DB      *db = h;
729     off_t   offset;

730     offset = (db->nhash + 1) * PTR_SZ;  /* +1 for free list ptr */

731     /*
732      * We're just setting the file offset for this process
733      * to the start of the index records; no need to lock.
734      * +1 below for newline at end of hash table.
735      */
736     if ((db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
737         err_dump("db_rewind: lseek error");
738   }

739   /*
740    * Return the next sequential record.
741    * We just step our way through the index file, ignoring deleted
742    * records.  db_rewind must be called before this function is
743    * called the first time.
744    */
745   char *
746   db_nextrec(DBHANDLE h, char *key)
747   {
748     DB      *db = h;
749     char    c;
750     char    *ptr;
```

[720–738] The `db_rewind` function is used to reset the database to "the beginning;" we set the file offset for the index file to point to the first record in the index file (immediately following the hash table). (Recall the structure of the index file from Figure 20.2.)

[739–750] The `db_nextrec` function returns the next record in the database. The return value is a pointer to the data buffer. If the caller provides a non-null value for the `key` parameter, the corresponding key is copied to this address. The caller is responsible for allocating a buffer big enough to store the key. A buffer whose size is `IDXLEN_MAX` bytes is large enough to hold any key.

Records are returned sequentially, in the order that they happen to be stored in the database file. Thus, the records are not sorted by key value. Also, because we do not follow the hash chains, we can come across records that have been deleted, but we will not return these to the caller.

```
751     /*
752      * We read lock the free list so that we don't read
753      * a record in the middle of its being deleted.
754      */
755     if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
756         err_dump("db_nextrec: readw_lock error");

757     do {
758         /*
759          * Read next sequential index record.
760          */
761         if (_db_readidx(db, 0) < 0) {
762             ptr = NULL;     /* end of index file, EOF */
763             goto doreturn;
764         }

765         /*
766          * Check if key is all blank (empty record).
767          */
768         ptr = db->idxbuf;
769         while ((c = *ptr++) != 0  &&  c == SPACE)
770             ;   /* skip until null byte or nonblank */
771     } while (c == 0);   /* loop until a nonblank key is found */

772     if (key != NULL)
773         strcpy(key, db->idxbuf);    /* return key */
774     ptr = _db_readdat(db);  /* return pointer to data buffer */
775     db->cnt_nextrec++;

776   doreturn:
777     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
778         err_dump("db_nextrec: un_lock error");
779     return(ptr);
780   }
```

[751–756] We first need to read lock the free list so that no other processes can remove a record while we are reading it.

[757–771] We call `_db_readidx` to read the next record. We pass in an offset of 0 to tell `_db_readidx` to continue reading from the current offset. Since we are reading the index file sequentially, we can come across records that have been deleted. We want to return only valid records, so we skip any record whose key is all spaces (recall that `_db_dodelete` clears a key by setting it to all spaces).

[772–780] When we find a valid key, we copy it to the caller's buffer if one was supplied. Then we read the data record and set the return value to point to the internal buffer containing the data record. We increment a statistics counter, unlock the free list, and return the pointer to the data record.

The normal use of db_rewind and db_nextrec is in a loop of the form

```
db_rewind(db);
while ((ptr = db_nextrec(db, key)) != NULL) {
    /* process record */
}
```

As we warned earlier, there is no order to the returned records; they are not in key order.

If the database is being modified while db_nextrec is called from a loop, the records returned by db_nextrec are simply a snapshot of a changing database at some point in time. db_nextrec always returns a "correct" record when it is called; that is, it won't return a record that was deleted. But it is possible for a record returned by db_nextrec to be deleted immediately after db_nextrec returns. Similarly, if a deleted record is reused right after db_nextrec skips over the deleted record, we won't see that new record unless we rewind the database and go through it again. If it's important to obtain an accurate "frozen" snapshot of the database using db_nextrec, then no insertions or deletions can be going on at the same time.

Look at the locking used by db_nextrec. We're not going through any hash chain, and we can't determine the hash chain that a record belongs on. Therefore, it is possible for an index record to be in the process of being deleted when db_nextrec is reading the record. To prevent this race, db_nextrec read locks the free list, thereby avoiding any interaction with _db_dodelete and _db_findfree.

Before we conclude our study of the db.c source file, we need to describe the locking when new index records or data records are appended to the end of the file. In cases 1 and 3, db_store calls both _db_writeidx and _db_writedat with a third argument of 0 and a fourth argument of SEEK_END. This fourth argument is the flag to these two functions, indicating that the new record is being appended to the file. The technique used by _db_writeidx is to write lock the index file from the end of the hash chain to the end of file. This won't interfere with any other readers or writers of the database (since they will lock a hash chain), but it does prevent other callers of db_store from trying to append at the same time. The technique used by _db_writedat is to write lock the entire data file. Again, this won't interfere with other readers or writers of the database (since they don't even try to lock the data file), but it does prevent other callers of db_store from trying to append to the data file at the same time. (See Exercise 20.3.)

20.9 Performance

We wrote a test program to test the database library and to obtain some timing measurements of the database access patterns of typical applications. This program takes two command-line arguments: the number of children to create and the number of database records (*nrec*) for each child to write to the database. The program then creates an empty database (by calling db_open), forks the number of child processes, and waits for all the children to terminate. Each child performs the following steps.

1. Write *nrec* records to the database.
2. Read the *nrec* records back by key value.
3. Perform the following loop *nrec* × *5* times.
   a. Read a random record.
   b. Every 37 times through the loop, delete a random record.
   c. Every 11 times through the loop, insert a new record and read the record back.
   d. Every 17 times through the loop, replace a random record with a new record. Every other one of these replacements is a record with the same size data; the alternate is a record with a longer data portion.
4. Delete all the records that this child wrote. Every time a record is deleted, ten random records are looked up.

The number of operations performed on the database is counted by the `cnt_xxx` variables in the `DB` structure, which were incremented in the functions. The number of operations differs from one child to the next, since the random-number generator used to select records is initialized in each child to the child's process ID. A typical count of the operations performed in each child is shown in Figure 20.6.

Operation	Calls to `fcntl` (per operation)		Operation count (*nrec* = 2,000)
	Coarse-grained locking	Fine-grained locking	
`db_store`, `DB_INSERT`, no empty record, appended	2	8	2,920
`db_store`, `DB_INSERT`, empty record reused	2	4	468
`db_store`, `DB_REPLACE`, different data length, appended	2	8	405
`db_store`, `DB_REPLACE`, equal data length	2	2	416
`db_store`, record not found	2	2	71
`db_fetch`, record found	2	2	32,873
`db_fetch`, record not found	2	2	2,966
`db_delete`, record found	2	4	3,388
`db_delete`, record not found	2	2	422

Figure 20.6 Typical count of operations performed by each child

We performed about ten times more fetches than stores or deletions, which is probably typical of many database applications.

Each child is performing these operations (fetching, storing, and deleting) only with the records that the child wrote. The concurrency controls are being exercised because all the children are operating on the same database (albeit different records in the same database). The total number of records in the database increases in proportion to the number of children. (With one child, *nrec* records are originally written to the database. With two children, *nrec* × *2* records are originally written, and so on.)

To test the concurrency provided by coarse-grained locking versus fine-grained locking and to compare the three types of locking (no locking, advisory locking, and

mandatory locking), we ran three versions of the test program. The first version used the source code shown in Section 20.8, which we've called fine-grained locking. The second version changed the locking calls to implement coarse-grained locking, as described in Section 20.6. The third version had all locking calls removed, so we could measure the overhead involved in locking. We can run the first and second versions (fine-grained locking and coarse-grained locking) using either advisory or mandatory locking, by changing the permission bits on the database files. (In all the tests reported in this section, we measured the times for mandatory locking using only the implementation of fine-grained locking.)

All the timing tests in this section were done on an Intel Core-i5 system running Linux 3.2.0. This system has four cores, which allows up to four processes to run concurrently.

Single-Process Results

Figure 20.7 shows the results when only a single child process ran, with an *nrec* of 2,000, 6,000, and 12,000.

nrec	No locking			Advisory locking						Mandatory locking		
				Coarse-grained locking			Fine-grained locking			Fine-grained locking		
	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock
2,000	0.10	0.22	0.33	0.17	0.33	0.51	0.13	0.38	0.51	0.14	0.43	0.58
6,000	0.59	1.32	1.91	0.88	2.13	3.03	0.90	2.14	3.05	0.99	2.52	3.53
12,000	4.37	9.58	13.97	5.38	12.60	18.01	5.34	12.63	18.01	5.53	15.03	20.60

Figure 20.7 Single child, varying *nrec*, different locking techniques

The last 12 columns give the corresponding times in seconds. In all cases, the user CPU time plus the system CPU time approximately equals the clock time. This set of tests was CPU limited and not disk limited.

The six columns under "Advisory locking" are almost equal for each row. This makes sense because for a single process; there is no difference between coarse-grained locking and fine-grained locking, except for the extra calls to `fcntl`.

Comparing no locking with advisory locking, we see that adding the locking calls increases the system CPU time by 32% to 73%. Even though the locks are never used (since only a single process is running), the system call overhead in the calls to `fcntl` adds time. Also note that the user CPU time is about the same for all four versions of locking. Since the user code is almost equivalent (except for the number of calls to `fcntl`), this makes sense.

The final point to note from Figure 20.7 is that mandatory locking increases the system CPU time by 13% to 19% compared to advisory locking. Since the number of locking calls is the same for advisory fine-grained locking and mandatory fine-grained locking, the additional system call overhead must be from the reads and writes.

The next test was to try the no-locking program with multiple children. The results, as expected, were random errors. Normally, records that were added to the database couldn't be found, and the test program aborted. Different errors occurred every time the test program was run. This illustrates a classic race condition: multiple processes updating the same file without using any form of locking.

Multiple-Process Results

The final set of measurements looks mainly at the differences between coarse-grained locking and fine-grained locking. As we said earlier, intuitively, we expect fine-grained locking to provide additional concurrency, since there is less time that portions of the database are locked from other processes. Figure 20.8 shows the results for an *nrec* of 2,000, varying the number of children from 1 to 16.

# Proc	Advisory locking							Mandatory locking			
	Coarse-grained locking			Fine-grained locking			Δ Clock	Fine-grained locking			Δ Sys
	User	Sys	Clock	User	Sys	Clock	Percent	User	Sys	Clock	Percent
1	0.14	0.35	0.50	0.14	0.35	0.50	0	0.15	0.42	0.58	20
2	0.60	1.43	1.88	0.54	1.36	1.10	71	0.65	2.01	1.59	48
3	0.97	2.67	3.18	1.37	3.73	2.20	45	1.62	5.67	3.28	52
4	2.38	6.17	5.59	2.83	8.15	4.07	37	3.29	12.35	6.31	52
5	3.72	10.17	8.37	4.28	11.86	6.09	37	4.96	18.47	9.49	56
6	5.02	14.52	11.52	6.04	17.46	8.89	30	6.66	26.38	13.22	51
7	7.00	20.16	15.84	8.06	23.23	11.88	33	9.12	36.13	18.09	56
8	9.12	26.20	20.31	10.50	30.50	15.48	31	11.81	47.20	23.49	55
9	11.60	33.91	25.64	13.40	37.80	19.29	33	14.54	60.23	29.66	59
10	14.28	42.24	31.35	16.39	47.01	23.74	32	17.84	74.05	36.27	58
11	17.37	51.12	37.50	19.71	56.59	28.57	31	21.57	90.14	44.10	59
12	20.70	60.48	44.24	23.47	66.10	33.34	33	25.57	108.94	53.11	65
13	25.13	70.67	51.96	27.70	77.76	39.21	33	29.71	133.31	63.07	71
14	28.40	82.23	59.88	32.34	91.45	46.22	30	34.22	155.80	73.86	70
15	32.23	94.26	68.30	36.32	102.97	51.82	32	39.05	180.66	84.14	75
16	37.24	107.87	78.67	42.17	118.20	59.72	32	44.11	208.28	96.82	76

Figure 20.8 Comparison of various locking techniques, *nrec* = 2,000

All times are in seconds and are the total for the parent and all its children. There are many items to consider from this data.

The first thing to notice is that the sum of the user and system times exceeds the clock time when multiple processes are used. This seems odd at first, but is normal when multiple cores are present. What happens is that all concurrently executing processes accumulate time as they execute; the CPU processing times shown are the sum of the times of all the cores used by the program. Because we can run multiple processes at the same time (one per core), the CPU processing times can exceed the clock time.

The eighth column, labeled "Δ Clock," is the percentage difference between the clock times from advisory coarse-grained locking and advisory fine-grained locking. This is a measurement of how much concurrency we obtain by going from coarse-

grained locking to fine-grained locking. On the system used for these tests, coarse-grained locking is the same as fine-grained locking for one process, but becomes more expensive (by about 30%) with multiple processes.

We would like the clock time to decrease from coarse-grained to fine-grained locking, which it does as soon as we start using multiple processes. However, we expect the system time to remain higher for fine-grained locking for any number of processes, because we are issuing more `fcntl` calls with fine-grained locking than with coarse-grained locking. If we total the number of `fcntl` calls in Figure 20.6, we have an average of 87,858 for coarse-grained locking and 115,520 for fine-grained locking. We expect this increase of 31% more calls to `fcntl` to result in increased system time for fine-grained locking. Therefore, the decrease in system time for fine-grained locking with two processes, and the relatively small increase with more than two processes, is puzzling.

There are two reasons for this behavior. First, recall from Figure 20.7 that there is no significant difference between coarse-grained locking times and fine-grained locking times when there is no contention for the locks. This shows that the CPU overhead of the extra `fcntl` calls doesn't affect the performance of the test program. The second reason is that with coarse-grained locking, we hold locks for longer periods of time, thus increasing the likelihood that other processes will block on a lock. With fine-grained locking, the locking is done over shorter intervals, so there is less chance that processes will block. If we count the number of times `fcntl` blocks, we will see that processes block more frequently with coarse-grained locking. For example, with four processes, coarse-grained locking blocks almost five times more frequently than with fine-grained locking. The extra work that processes need to do to put themselves to sleep and wake up more often with coarse-grained locking increases the system time, reducing the difference in system times between the two locking approaches.

The final column in Figure 20.8, labeled "Δ Sys," is the percentage increase in the system CPU time from advisory fine-grained locking to mandatory fine-grained locking. These percentages show that mandatory locking adds significantly (between 20% and 76%) to the system time as concurrency increases.

Since the user code for all these tests is almost identical (there are some additional `fcntl` calls for both advisory fine-grained and mandatory fine-grained locking), we expect the user CPU times to be the same across any row.

> The first time we ran these tests, we measured the user times for coarse-grained locking to be almost twice as long as the times for fine-grained locking when multiple processes competed for the locks. Because the two versions of the database are the same, except for the number of calls to `fcntl`, this made no sense. After investigating, we discovered that because there was more contention with coarse-grained locking, processes were waiting longer, and the operating system decided to reduce the CPU clock frequency to save power. With fine-grained locking, there was more activity, so the system increased the CPU clock frequency. This (artificially) made the coarse-grained locking tests run more slowly than the fine-grained tests. After disabling the frequency scaling feature, we measured the performance of the test without this bias, and the difference in user times was much smaller.

The values in the first row of Figure 20.8 are similar to those for an *nrec* of 2,000 in Figure 20.7. This corresponds to our expectation.

Figure 20.9 is a graph of the data from Figure 20.8 for advisory fine-grained locking. We plot the clock time as the number of processes goes from 1 to 16. We also plot the user CPU time divided by the number of processes and the system CPU time divided by the number of processes.

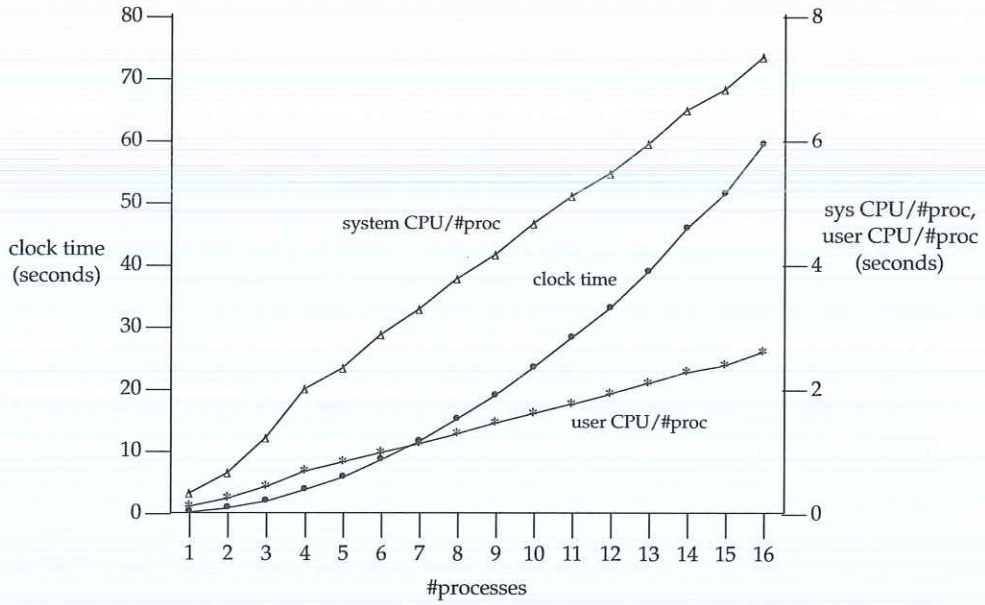

Figure 20.9 Values from Figure 20.8 for advisory fine-grained locking

Note that both CPU times, divided by the number of processes, are linear but that the plot of the clock time is nonlinear. The probable reason is the added amount of CPU time used by the operating system to switch between processes as the number of processes increases. This operating system overhead would show up as an increased clock time, but shouldn't affect the CPU times of the individual processes.

The reason the user CPU time increases with the number of processes is that there are more records in the database. Each hash chain is getting longer, so it takes the `_db_find_and_lock` function longer, on average, to find a record.

20.10 Summary

This chapter has taken a long look at the design and implementation of a database library. Although we've kept the library small and simple for presentation purposes, it contains the record locking required to allow concurrent access by multiple processes.

We've also looked at the performance of this library with various numbers of processes using no locking, advisory locking (fine-grained and coarse-grained), and mandatory locking. With a single process, we saw that advisory locking adds between

29% and 59% to the clock time over no locking and that mandatory locking adds about another 15% over advisory locking.

Exercises

20.1 The locking in `_db_dodelete` is somewhat conservative. For example, we could allow more concurrency by not write locking the free list until we really need to; that is, the call to `writew_lock` could be moved between the calls to `_db_writedat` and `_db_readptr`. What happens if we do this?

20.2 If `db_nextrec` did not read lock the free list and a record that it was reading was also in the process of being deleted, describe how `db_nextrec` could return the correct key but an all-blank (hence incorrect) data record. (Hint: Look at `_db_dodelete`.)

20.3 At the end of Section 20.8, we described the locking performed by `_db_writeidx` and `_db_writedat`. We said that this locking didn't interfere with other readers and writers except those making calls to `db_store`. Is this true if mandatory locking is being used?

20.4 How would you integrate the `fsync` function into this database library?

20.5 In `db_store`, we write the data record before the index record. What happens if you do it in the opposite order?

20.6 Create a new database and write some number of records to the database. Write a program that calls `db_nextrec` to read each record in the database, and call `_db_hash` to calculate the hash value for each record. Print a histogram of the number of records on each hash chain. Is the hashing function in `_db_hash` adequate?

20.7 Modify the database functions so that the number of hash chains in the index file can be specified when the database is created.

20.8 Compare the performance of the database functions when the database is (a) on the same host as the test program and (b) on a different host accessed via NFS. Does the record locking provided by the database library still work?

20.9 The database reuses free list records only if the sizes of the key buffer and data buffer match the needed sizes exactly. Modify the database to allow larger buffer sizes on the free list to satisfy the request. How do you have to change the persistent format of the database to support this feature?

20.10 After implementing a solution to Exercise 20.9, write a tool to convert one database format to the other.

This page intentionally left blank