1 mod_perl and dbm files

1.1 Description

Small databases can be implemented pretty efficiently using dbm files, but there are still some precautions that must be taken to use properly under mod_perl.

1.2 Where and Why to use dbm files

Some of the earliest databases implemented on Unix were dbm files, and many are still in use today. As of this writing the Berkeley DB is the most powerful dbm implementation (http://www.sleepycat.com).

If you need a light database, with an easy API, using simple key-value pairs to store and manipulate a relatively small number of records, this is a solution that should be amongst the first you consider.

With dbm, it is rare to read the whole database into memory. Combine this feature with the use of smart storage techniques, and dbm files can be manipulated much faster than flat files. Flat file databases can be very slow on insert, update and delete operations, when the number of records starts to grow into the thousands. Sort algorithms on flat files can be very time-consuming.

The maximum practical size of a dbm database depends on many factors--your data, your hardware and the desired response times of course included--but as a rough guide consider 5,000 to 10,000 records to be reasonable.

We will talk mostly about the Berkley DB version 1.x, as it provides the best functionality while having a good speed and almost no limitations. Other implementations might be faster in some cases, but they are either limited in the length of the maximum value or the total number of records.

There is a number of Perl interfaces to the major dbm implementations, to list a few: `DB_File`, `NDBM_File`, `ODBM_File`, `GDBM_File`, and `SDBM_File`. The original Perl module for Berkeley DB was DB_File, which was written to interface to Berkeley DB version 1.85. The newer Perl module for Berkeley DB is `BerkeleyDB`, which was written to interface to version 2.0 and subsequent releases. Because Berkeley DB version 2.X has a compatibility API for version 1.85, you can (and should!) build `DB_File` using version 2.X of Berkeley DB, although `DB_File` will still only support the 1.85 functionality.

Several different indexing algorithms (known also as access methods) can be used with dbm implementations:

- The `HASH` access method gives an `0(1)` complexity of search and update, fast insert and delete, but a slow sort (which you have to implement yourself). (Used by almost all dbm implementations)
- The `BTREE` access method allows arbitrary key/value pairs to be stored in a sorted, balanced binary tree. This allows us to get a sorted sequence of data pairs in `0(1)`, but at the expense of much slower insert, update, delete operations than is the case with `HASH`. (Available mostly in Berkeley DB)
- The `RECNO` access method is more complicated, and enables both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in `HASH` and `BTREE`. In this case the key will consist of a record (line) number. (Available mostly in Berkeley DB)

- The `QUEUE` access method stores fixed-length records with logical record numbers as keys. It is designed for fast inserts at the tail and has a special cursor consume operation that deletes and returns a record from the head of the queue. The `QUEUE` access method uses record level locking. (Available only in Berkeley DB version 3.x)

Most often you will want to use the `HASH` method, but there are many considerations and your choice may be dictated by your application.

In recent years dbm databases have been extended to allow you to store more complex values, including data structures. The `MLDBM` module can store and restore the whole symbol table of your script, including arrays and hashes.

It is important to note that you cannot simply switch a dbm file from one storage algorithm to another. The only way to change the algorithm is to copy all the records one by one into a new dbm file, which was initialized according to a desired access method. You can use a script like this:

```
#!/usr/bin/perl -w

#
# This script takes as its parameters a list of Berkeley DB
# file(s) which are stored with the DB_BTREE algorithm.  It
# will back them up using the .bak extension and create
# instead dbms with the same records but stored using the
# DB_HASH algorithm
#
# Usage: btree2hash.pl filename(s)

use strict;
use DB_File;
use Fcntl;

  # Do checks
die "Usage: btree2hash.pl filename(s))\n" unless @ARGV;

foreach my $filename (@ARGV) {

  die "Can’t find $filename: $!\n"
    unless -e $filename and -r $filename;

    # First backup the file
  rename "$filename", "$filename.btree"
    or die "can’t rename $filename $filename.btree:$!\n";

    # tie both dbs (db_hash is a fresh one!)
  tie my %btree , ’DB_File’,"$filename.btree", O_RDWR|O_CREAT,
      0660, $DB_BTREE or die "Can’t tie $filename.btree: $!";
  tie my %hash ,  ’DB_File’,"$filename" , O_RDWR|O_CREAT,
      0660, $DB_HASH  or die "Can’t tie $filename: $!";

    # copy DB
  %hash = %btree;
```

```
      # untie
    untie %btree ;
    untie %hash ;
  }
```

Note that some dbm implementations come with other conversion utilities as well.

1.3 mod_perl and dbm

Where does mod_perl fit into the picture?

If you need to access a dbm file in your mod_perl code in the read only mode the operation would be much faster if you keep the dbm file open (tied) all the time and therefore ready to be used. This will work with dynamic (read/write) databases accesses as well, but you need to use locking and data flushing to avoid data corruption.

Although mod_perl and dbm can give huge performance gains compared to the use of flat file databases you should be very careful. In addition to the need for locking, you need to consider the consequences of `die()` and unexpected process death.

If your locking mechanism cannot handle dropped locks, a stale lock can deactivate your whole site. You can enter a deadlock situation if two processes simultaneously try to acquire locks on two separate databases. Each has locked only one of the databases, and cannot continue without locking the second. Yet this will never be freed because it is locked by the other process. If your processes all ask for their DB files in the same order, this situation cannot occur.

If you modify the DB you should be make very sure that you flush the data and synchronize it, especially when the process serving your handler unexpectedly dies. In general your application should be tested very thoroughly before you put it into production to handle important data.

1.4 Locking dbm Handlers and Write Lock Starvation Hazards

One has to deploy dbm file locking if there is chance that some process will want to write to it. Note that once you need to do locking you do it even when all you want is to read from the file. Since if you don’t, it’s possible that someone writes to the file at this very moment and you may read partly updated data.

Therefore we should distinguish between *READ* and *WRITE* locks. Before doing an operation on the dbm file, we first issue either a *READ* or a *WRITE* lock request, according to our needs.

If we are making a *READ* lock request, it is granted as soon as the *WRITE* lock on the file is removed if any or if it is already *READ* locked. The lock status becomes *READ* on success.

If we make a *WRITE* lock request, it is granted as soon as the file becomes unlocked. The lock status becomes *WRITE* on success.

The treatment of the *WRITE* lock request is most important.

If the DB is *READ* locked, a process that makes a *WRITE* request will poll until there are no reading or writing processes left. Lots of processes can successfully read the file, since they do not block each other. This means that a process that wants to write to the file may never get a chance to squeeze in, since it needs to obtain an exclusive lock.

The following diagram represents a possible scenario where everybody can read but no one can write (pX's represent different processes running for different times and all acquiring the read lock on the dbm file):

```
  [-p1-]                  [--p1--]
     [--p2--]                 [--p2--]
   [---------p3---------] [--------p3----....
                 [------p4-----]
```

The result is a starving process, which will timeout the request, and it will fail to update the DB. Ken Williams solved the above problem with his `Tie::DB_Lock` module, which is discussed in one of the following sections.

There are several locking wrappers for `DB_File` in CPAN right now. Each one implements locking differently and has different goals in mind. It is therefore worth knowing the difference, so that you can pick the right one for your application.

1.5 Flawed Locking Methods Which Must Not Be Used

Caution: The suggested locking methods in the Camel book and `DB_File` man page (before version 1.72, fixed in 1.73) are flawed. If you use them in an environment where more than one process can modify the dbm file, it can get corrupted!!! The following is an explanation of why this happens.

You may not use a tied file's filehandle for locking, since you get the filehandle after the file has been already tied. It's too late to lock. The problem is that the database file is locked *after* it is opened. When the database is opened, the first 4k (in Berkley dbm library) is read and then cached in memory. Therefore, a process can open the database file, cache the first 4k, and then block while another process writes to the file. If the second process modifies the first 4k of the file, when the original process gets the lock is now has an inconsistent view of the database. If it writes using this view it may easily corrupt the database on disk.

This problem can be difficult to trace because it does not cause corruption every time a process has to wait for a lock. One can do quite a bit of writing to a database file without actually changing the first 4k. But once you suspect this problem you can easily reproduce it by making your program modify the records in the first 4k of the DB.

You better resort to using the standard modules for locking instead of inventing your own.

If your dbm file is used only in the read-only mode generally there is no need for locking at all. If you access the dbm file in read/write mode, the safest method is to tie() the dbm file after acquiring an external lock and untie() before the lock is released. So to access the file in shared mode (FLOCK_SH) one should following this pseudo-code:

```
flock FLOCK_SH <===== start critical section
tie()
read...
untie()
flock FLOCK_UN <===== end critical section
```

Similar for the exclusive (EX), write access:

```
flock FLOCK_EX <===== start critical section
tie()
write...
sync()
untie()
flock FLOCK_UN <===== end critical section
```

However you might want to save a few tie()/untie() calls if the same request accesses the dbm file more than once. You should be careful though. Based on the caching effect explained above, a process can perform an atomic downgrade of an exclusive lock to a shared one without re-tie()ing the file:

```
flock FLOCK_EX <===== start critical section
tie()
write...
sync()
               <===== end critical section
flock FLOCK_SH <===== start critical section
read...
untie()
flock FLOCK_UN <===== end critical section
```

because it has the updated data in its cache. By atomic, we mean it's ensured that the lock status gets changed, without any other process getting an exclusive access in between.

If you can ensure that one process safely upgrades a shared lock with an exclusive lock, one can save on tie()/untie(). But this operation might lead to a dead-lock if two processes try to upgrade a shared lock with exclusive at the same time. Remember that in order to acquire an exclusive lock, all other processes need to release *all* locks. If your OS locking implementation resolves this deadlock by denying one of the upgrade requests, make sure your program handles that appropriately. The process that were denied has to untie() the dbm file and then ask for an exclusive lock.

A dbm file has always to be untie()'ed before the locking is released (unless you do an atomic downgrade from exclusive to shared as we have just explained). Remember that if at any given moment a process wants to lock and access the dbm file it has to re-tie() this file, if it was tied already. If this is not done, the integrity of the dbm file is not ensured.

To conclude, the safest method of reading from dbm file is to lock the file before tie()-ing it, untie() it before releasing the lock, and in the case of write to call sync() before untie()-ing it.

1.6 Locking Wrappers Overview

Here are some of the correctly working dbm locking wrappers on (three of them are available from CPAN):

- `Tie::DB_Lock` -- `DB_File` wrapper which creates copies of the dbm file for read access, so that you have kind of a multiversioning concurrent read system. However, updates are still serial. After each update the read-only copies of the dbm file are recreated. Use this wrapper in situations where reads may be very lengthy and therefore write starvation problem may occur. On the other hand if you have big dbm files, it may create a big load on the system if the updates are quite frequent. More information.

- `Tie::DB_FileLock` -- `DB_File` wrapper that has the ability to lock and unlock the database while it is being used. Avoids the tie-before-flock problem by simply re-tie-ing the database when you get or drop a lock. Because of the flexibility in dropping and re-acquiring the lock in the middle of a session, this can be used in a system that will work with long updates and/or reads. Refer to the `Tie::DB_FileLock` manpage for more information.

- `DB_File::Lock` -- extremely lightweight `DB_File` wrapper that simply flocks an external lockfile before tie-ing the database and drops the lock after untie. Allows one to use the same lockfile for multiple databases to avoid deadlock problems, if desired. Use this for databases where updates and reads are quick and simple flock locking semantics are enough. Refer to `DB_File::Lock` manpage for more information.

- `DB_File::Lock2` -- does the same thing as `DB_File::Lock`, but has a slightly different implementation. I wrote it before David Harris released his `DB_File::Lock` and I didn't want to kill mine, so I'll keep it here for a while :).

- On some Operating Systems (FreeBSD is one example) it is possible to lock on tie:

  ```
  tie my %t, 'DB_File', $TOK_FILE, O_RDWR | O_EXLOCK, 0664;
  ```

  and only release the lock by un-tie()-ing the file. Check if the `O_EXLOCK` flag is available on your operating system before you try to use this method!

1.7 Tie::DB_Lock

`Tie::DB_Lock` ties hashes to databases using shared and exclusive locks. This module, by Ken Williams, solves the problems raised in the previous section.

The main difference from what I have described above is that `Tie::DB_Lock` copies a dbm file on read. Reading processes do not have to keep the file locked while they read it, and writing processes can still access the file while others are reading. This works best when you have lots of long-duration reading, and a few short bursts of writing.

The drawback of this module is the heavy IO performed when every reader makes a fresh copy of the DB. With big dbm files this can be quite a disadvantage and can slow the server down considerably.

An alternative would be to have one copy of the dbm image shared by all the reading processes. This can cut the number of files that are copied, and puts the responsibility of copying the read-only file on the writer, not the reader. It would need some care to make sure it does not disturb readers when putting a new read-only copy into place.

1.8 DB_File::Lock2

Here is *code/DB_File-Lock2.pm*:

```
package DB_File::Lock2;
require 5.004;

use strict;

BEGIN {
    # RCS/CVS compliant:  must be all one line, for MakeMaker
  $DB_File::Lock2::VERSION = do { my @r = (q$Revision: 1.1 $ =~ /\d+/g); sprintf "%d."."%02d" x $#r, @r };

}

use DB_File ();
use Fcntl qw(:flock O_RDWR O_CREAT);
use Carp qw(croak carp verbose);
use Symbol ();

@DB_File::Lock2::ISA    = qw( DB_File );
%DB_File::Lock2::lockfhs = ();

use constant DEBUG => 0;

  # file creation permissions mode
use constant PERM_MODE => 0660;

  # file locking modes
%DB_File::Lock2::locks =
  (
   read  => LOCK_SH,
   write => LOCK_EX,
  );

# SYNOPSIS:
# tie my %mydb, ’DB_File::Lock2’, $filepath,
#     [’read’ || ’write’, ’HASH’ || ’BTREE’]
# while (my($k,$v) = each %mydb) {
#   print "$k => $v\n";
# }
# untie %mydb;
#########
sub TIEHASH {
  my $class     = shift;
  my $file      = shift;
  my $lock_mode = lc shift || ’read’;
  my $db_type   = shift || ’HASH’;

  die "Dunno about lock mode: [$lock_mode].\n
       Valid modes are ’read’ or ’write’.\n"
    unless $lock_mode eq ’read’ or $lock_mode eq ’write’;

  # Critical section starts here if in write mode!
```

```
    # create an external lock
  my $lockfh = Symbol::gensym();
  open $lockfh, ">$file.lock" or die "Cannot open $file.lock for writing: $!\n";
  unless (flock $lockfh, $DB_File::Lock2::locks{$lock_mode}) {
    croak "cannot flock: $lock_mode => $DB_File::Lock2::locks{$lock_mode}: $!\n";
  }

  my $self = $class->SUPER::TIEHASH
    ($file,
     O_RDWR|O_CREAT,
     PERM_MODE,
     ($db_type eq 'BTREE' ? $DB_File::DB_BTREE : $DB_File::DB_HASH )
    );

    # remove the package name in case re-blessing occurs
  (my $id = "$self") =~ s/^[^=]+=//;

    # cache the lock fh
  $DB_File::Lock2::lockfhs{$id} = $lockfh;

  return $self;

} # end of sub new


# DESTROY is automatically called when a tied variable
# goes out of scope, on explicit untie() or when the program is
# interrupted, e.g. with a die() call.
#
# It unties the db by forwarding it to the parent class,
# unlocks the file and removes it from the cache of locks.
###########
sub DESTROY{
  my $self = shift;

  $self->SUPER::DESTROY(@_);

    # now it safe to unlock the file, (close() unlocks as well). Since
    # the object has gone we remove its lock filehandler entry
    # from the cache.
  (my $id = "$self") =~ s/^[^=]+=//; # see 'sub TIEHASH'
  close delete $DB_File::Lock2::lockfhs{$id};

    # Critical section ends here if in write mode!

  print "Destroying ".__PACKAGE__."\n" if DEBUG;

}

####
END {
  print "Calling the END from ".__PACKAGE__."\n" if DEBUG;

}

1;
```

which does the locking by using an external lockfile.

This allows you to gain the lock before the file is tied. Note that it’s not yet on CPAN and so is linked from here in its entirety. Note also that this code still needs some testing, so *be careful* if you use it on a production machine.

You use it like this:

```
use DB_File::Lock2 ();
```

A simple tie, READ lock and untie

```
use DB_File::Lock2 ();
my $dbfile = "/tmp/test";
tie my %mydb, 'DB_File::Lock2', $dbfile, 'read';
print $mydb{foo} if exists $mydb{foo};
untie %mydb;
```

You can even skip the `untie()` call. When `$mydb` goes out of scope everything will be done automatically. However it is better to use the explicit call, to make sure the critical sections between lock and unlock are as short as possible. This is especially important when requesting an exclusive (write) lock.

The following example shows how it might be convenient to skip the explicit `untie()`. In this example, we don't need to save the intermediate result, we just return and the cleanup is done automatically.

```
use DB_File::Lock2 ();
my $dbfile = "/tmp/test";
print user_exists("stas") ? "Yes" : "No";
sub user_exists{
  my $username = shift || '';

  warn("No username passed\n"), return 0 unless $username;

  tie my %mydb, 'DB_File::Lock2', $dbfile, 'read';

  # if we match the username return 1, else 0
  return $mydb{$username} ? 1 : 0;

} # end of sub user_exists
```

Now let's write all the upper case characters and their respective ASCII values to a dbm file. Then read the file and print the contents of the DB, unsorted.

```
use DB_File::Lock2 ();
my $dbfile = "/tmp/test";

  # write
tie my %mydb, 'DB_File::Lock2', $dbfile,'write';
for (0..26) {
  $mydb{chr 65+$_} = $_;
}
untie %mydb;

  # now, read them and printout (unsorted)
  # notice that 'read' is a default lock mode
tie %mydb, 'DB_File::Lock2', $dbfile;
while (my($k,$v) = each %mydb) {
  print "$k => $v\n";
}
untie %mydb;
```

If your CGI script is interrupted, the `DESTROY` block will take care of unlocking the dbm file and flush any changes. So your DB will be safe against possible corruption because of unclean program termination.

1.9 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman <stas (at) stason.org>

1.10 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: