NAME

DbLockTab − lock manager

SYNOPSIS

```
#include <db_cxx.h>

static int
DbLockTab::open(const char *dir,
        int flags, int mode, DbEnv *dbenv, DbLockTab **regionp);

int
DbLockTab::id(u_int32_t *idp);

int
DbLockTab::vec(u_int32_t locker, int flags,
        DB_LOCKREQ list[], int nlist, DB_LOCKREQ **elistp);

int
DbLockTab::get(u_int32_t locker, int flags,
        const Dbt *obj, const db_lockmode_t lock_mode, DB_LOCK *lock);

int
DbLockTab::close();

static int
DbLockTab::unlink(const char *dir, int force, DbEnv *dbenv);

int
DbLockTab::detect(int flags, int atype);
```

DESCRIPTION

The DB library is a family of classes that provides a modular programming interface to transactions and record-oriented file access. The library includes support for transactions, locking, logging and file page caching, as well as various indexed access methods. Many of the classes (e.g., the file page caching class) are useful independent of the other DB classes, although some classes are explicitly based on other classes (e.g., transactions and logging). For a general description of the DB package, see *db_intro*(3).

This manual page describes the specific details of the locking interface.

The *DbLockTab* class is intended to provide general-purpose locking. While designed to work with the other Db functions, this class is also useful for more general locking purposes. Locks can be shared between processes. In most cases, when multiple treads or processes are using locking, the deadlock detector, *db_deadlock*(1), should be run.

DbLockTab::open

The *DbLockTab::open* method copies a pointer, to the lock table identified by the **directory** *dir*, into the memory location referenced by *regionp*.

If the *dbenv* argument to *DbLockTab::open* was initialized using *DbEnv::appinit*, *dir* is interpreted as described by *DbEnv*(3).

Otherwise, if *dir* is not NULL, it is interpreted relative to the current working directory of the process. If *dir* is NULL, the following environment variables are checked in order: ‘‘TMPDIR’’, ‘‘TEMP’’, and ‘‘TMP’’. If one of them is set, lock table files are created relative to the directory it specifies. If none of them are set, the first possible one of the following directories is used: */var/tmp*, */usr/tmp*, */temp*, */tmp*, *C:/temp* and *C:/tmp*.

All files associated with the lock table are created in this directory. This directory must already exist when DbLockTab::open is called. If the lock table already exists, the process must have permission to read and write the existing files. If the lock table does not already exist, it is optionally created and initialized.

The *flags* and *mode* arguments specify how files will be opened and/or created when they don’t already exist. The flags value is specified by **or**’ing together one or more of the following values:

DB_CREATE
Create any underlying files, as necessary. If the files do not already exist and the DB_CREATE flag is not specified, the call will fail.

DB_THREAD
Cause the DbLockTab handle returned by the *DbLockTab::open* method to be useable by multiple threads within a single address space, i.e., to be ‘‘free-threaded’’.

All files created by the lock subsystem are created with mode *mode* (as described in *chmod*(2)) and modified by the process' umask value at the time of creation (see *umask*(2)). The group ownership of created files is based on the system and directory defaults, and is not further specified by DB.

The locking subsystem is configured based on which set methods have been used. It is expected that applications will use a single DbEnv object as the argument to all of the subsystems in the DB package. The fields of the DbEnv object used by *DbLockTab::open* are described below. As references to the DbEnv object may be maintained by *DbLockTab::open*, it is necessary that the DbEnv object and memory it references be valid until the object is destroyed. Any of the DbEnv fields that are not explicitly set will default to appropriate values.

The following fields in the DbEnv object may be initialized, using the appropriate set method, before calling *DbLockTab::open*:

void *(*db_errcall)(char *db_errpfx, char *buffer);
FILE *db_errfile;
const char *db_errpfx;
class ostream *db_error_stream;
int db_verbose;
The error fields of the DbEnv behave as described for *DbEnv*(3).

const u_int8_t lk_conflicts[][];
A *lk_modes* by *lk_modes* array. A non-0 value for the array element:

lk_conflicts[requested_mode][held_mode]

indicates that requested_mode and held_mode conflict. The ‘‘not-granted’’ mode must be represented by 0. If *lk_conflicts* is NULL, the conflicts array *db_rw_conflicts* is used; see the section below entitled ‘‘STANDARD LOCK MODES’’ for a description of that array.

u_int32_t lk_detect;
If non-0, specifies that the deadlock detector be run whenever a lock conflict occurs, and specifies which transaction should be aborted in the case of a deadlock. The *lk_detect* field must be set to one of the following values.

DB_LOCK_DEFAULT
Use the default policy as specified in the *db_deadlock*(1) man page.

DB_LOCK_OLDEST
Abort the oldest transaction.

DB_LOCK_RANDOM
Abort a random transaction involved in the deadlock.

DB_LOCK_YOUNGEST
Abort the youngest transaction.

unsigned int lk_max;

The maximum number of locks to be held or requested in the table. This value is used by *DbLockTab::open* to estimate how much space to allocate for various lock-table data structures. If *lk_max* is not explicitly set, a default value is used.

int lk_modes;

The number of lock modes to be recognized by the lock table (including the ‘‘not-granted’’ mode). If *lk_modes* is 0, the value DB_LOCK_RW_N is used; see below for a description of that value.

The *DbLockTab::open* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

DbLockTab::id

The *DbLockTab::id* method copies a locker ID, which is guaranteed to be unique in the specified lock table, into the memory location referenced by *idp*.

The access methods (see *Db::open* in *Db*(3)), generate a unique locker ID for each file that is opened with locking. During Db access method operation, this locker ID will be used for all lock calls unless a transaction identifier was specified for the call, in which case the transaction ID specified is used for locking.

The *DbLockTab::id* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

DbLockTab::vec

The *DbLockTab::vec* method atomically obtains and releases one or more locks from the specified table. The *DbLockTab::vec* method is intended to support acquisition or trading of multiple locks under one lock table semaphore, as is needed for lock coupling or in multigranularity locking for lock escalation.

The *locker* argument specified to *DbLockTab::vec* is an unsigned 32-bit integer quantity. It represents the entity requesting or releasing the lock.

The *flags* value must be set to 0 or the following value:

DB_LOCK_NOWAIT

If a lock cannot be granted because the requested lock conflicts with an existing lock, return immediately instead of waiting for the lock to become available.

The *list* array provided to *DbLockTab::vec* is typedef’d in <db_cxx.h> as DB_LOCKREQ. A DB_LOCKREQ structure has at least the following fields, which must be initialized before calling *DbLockTab::vec*:

lockop_t op;

The operation to be performed, which must be set to one of the following values:

DB_LOCK_GET

Get a lock, as defined by the values of *locker*, *obj* and *mode*. Upon return from *DbLockTab::vec*, if the *lock* field is non-NULL, a reference to the acquired lock is stored there. (This reference is invalidated by any call to *DbLockTab::vec* or *DbLock::put* that releases the lock.) See *DbLock*(3).

DB_LOCK_PUT

The lock referenced by the contents of the *lock* field is released.

DB_LOCK_PUT_ALL

All locks held by the *locker* are released. (Any locks acquired as a part of the current call to *DbLockTab::vec* that appear after the DB_LOCK_PUT_ALL entry are not considered for this operation).

DB_LOCK_PUT_OBJ
All locks held by the *locker*, on the object *obj*, with the mode specified by *lock_mode*, are released. A *lock_mode* of DB_LOCK_NG indicates that all locks on the object should be released. Note that any locks acquired as a part of the current call to *DbLockTab::vec* that occur before the DB_LOCK_PUT_OBJ will also be released; those acquired afterwards will not be released.

const Dbt obj;
An untyped byte string that specifies the object to be locked or released.

const lockmode_t mode;
The lock mode, used as an index into object's conflict array.

DB_LOCK lock;
A lock reference.

The *nlist* argument specifies the number of elements in the *list* array.

If any of the requested locks cannot be acquired, or any of the locks to be released cannot be released, the operations before the failing operation are guaranteed to have completed successfully, and *DbLockTab::vec* returns a non-zero value. In addition, if *elistp* is not NULL, it is set to point to the DB_LOCKREQ entry that was being processed when the error occurred.

In the case of an error, *DbLockTab::vec* may return one of the following values:

DB_LOCK_DEADLOCK
The specified *locker* was selected as a victim in order to resolve a deadlock.

DB_LOCK_NOTHELD
The lock cannot be released, as it was not held by the *locker*.

DB_LOCK_NOTGRANTED
A lock was requested that could not be granted and the *flag* parameter was set to DB_LOCK_NOWAIT. In this case, if non-NULL, *elistp* identifies the request that was granted.

Otherwise, the *DbLockTab::vec* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

DbLockTab::get
The *DbLockTab::get* method is a simple interface to the

DbLockTab::vec functionality, and is equivalent to calling the

DbLockTab::vec method with the *locker* argument, *elistp* and *conflict* arguments, and a single element *list* array, for which the *op* field is DB_LOCK_GET, and the *obj*, *lock_mode* and *lock* fields are represented by the arguments of the same name. Note that the type of the *obj* argument to *DbLockTab::get* is different from the *obj* element found in the DB_LOCKREQ structure. The *DbLockTab::get* method returns success and failure as described for the

DbLockTab::vec method.

DbLockTab::close
The *DbLockTab::close* method disassociates the calling process from the lock table. The object should not be used after a call to close. Note that *DbLockTab::close* does not release any locks still held by the closing process. (This provides functionality for long-lived locks.)

Processes that wish to have all their locks released can do so by issuing the appropriate *DbLockTab::vec* call.

In addition, if the *dir* argument to *DbLockTab::open* was NULL and *dbenv* was not initialized using *DbEnv::appinit*, all files created for this shared region will be removed, as if *DbLockTab::unlink* were called.

When multiple threads are using the DbLockTab handle concurrently, only a single thread may call the *DbLockTab::close* method.

The *DbLockTab::close* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

DbLockTab::unlink

The *DbLockTab::unlink* method destroys the lock table identified by the directory *dir*, removing all files used to implement the lock table. (The directory *dir* is not removed.) If there are processes that have called *DbLockTab::open* without calling *DbLockTab::close* (i.e., there are processes currently using the lock table), *DbLockTab::unlink* will fail without further action, unless the force flag is set, in which case *DbLockTab::unlink* will attempt to remove the lock table files regardless of any processes still using the lock table.

The result of attempting to forcibly destroy the region when a process has the region open is unspecified. Processes using a shared memory region maintain an open file descriptor for it. On UNIX systems, the region removal should succeed and processes that have already joined the region should continue to run in the region without change, however processes attempting to join the lock table will either fail or attempt to create a new region. On other systems, e.g., WNT, where the *unlink*(2) system call will fail if any process has an open file descriptor for the file, the region removal will fail.

In the case of catastrophic or system failure, database recovery must be performed (see *db_recover*(1) or the DB_RECOVER flags to *DbEnv::appinit*(3)). Alternatively, if recovery is not required because no database state is maintained across failures, it is possible to clean up a lock table by removing all of the files in the directory specified to the *DbLockTab::open* method, as lock table files are never created in any directory other than the one specified to *DbLockTab::open*. Note, however, that this has the potential to remove files created by the other DB subsystems in this database environment.

The *DbLockTab::unlink* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

DbLockTab::detect

The *DbLockTab::detect* method runs one iteration of the deadlock detector on the table represented by DbLockTab. The deadlock detector traverses the lock table, detects deadlocks, and if it finds one, marks one of the participating transactions for abort and then returns.

The flags value is specified by **or**'ing together one or more of the following values:

DB_LOCK_CONFLICT

Only run the deadlock detector if a lock conflict has occurred since the last time that the deadlock detector was run.

The *atype* parameter specifies which transaction to abort in the case of deadlock. It must be set to one of values described above for the *lk_detect* field of the *DbEnv* object.

The *DbLockTab::detect* method throws a *DbException*(3) or returns the value of *errno* on failure and 0 on success.

The *DbLockTab::detect* method is based on the C *lock_detect* function, which is the underlying function used by the *db_deadlock*(1) utility. See the source code for the *db_deadlock* utility for an example of using *lock_detect* in a UNIX environment.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *db_lock*:

DB_HOME

If the *dbenv* argument to *DbLockTab::open* was initialized using *db_appinit*, the environment variable DB_HOME may be used as the path of the database home for the interpretation of the *dir* argument to *DbLockTab::open*, as described in *db_appinit*(3).

TMPDIR

If the *dbenv* argument to *DbLockTab::open* was NULL or not initialized using *db_appinit*, the environment variable TMPDIR may be used as the directory in which to create the lock table, as described in the *DbLockTab::open* section above.

STANDARD LOCK MODES

The include file <db_cxx.h> declares two commonly used conflict arrays:

const u_int8_t db_lock_rw_conflicts[];

This is a conflict array for a simple scheme using shared and exclusive lock modes.

const u_int8_t db_lock_riw_conflicts[];

This is a conflict array that involves various intent lock modes (e.g., intent shared) that are used for multigranularity locking.

Their associated sizes are DB_LOCK_RW_N and DB_LOCK_RIW_N.

In addition, the include file <db_cxx.h> defines the type *db_lockmode_t*, which is the type of the lock modes used with the standard tables above:

DB_LOCK_NG

not granted (always 0)

DB_LOCK_READ

read (shared)

DB_LOCK_WRITE

write (exclusive)

ERRORS

Methods marked as returning *errno* will, by default, throw an exception that encapsulates the error information. The default error behavior can be changed, see *DbException*(3).

The *DbLockTab::open* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DbLock::unlink(3), close(2), db_version(3), fcntl(2), fflush(3), lseek(2), malloc(3), memcpy(3), memset(3), mmap(2), munmap(2), open(2), sigfillset(3), sigprocmask(2), stat(2), strcpy(3), strdup(3), strerror(3), strlen(3), unlink(2), and write(2).

In addition, the *DbLockTab::open* method may fail and throw a *DbException*(3) or return *errno* for the following conditions:

[EAGAIN]

The shared memory region was locked and (repeatedly) unavailable.

[EINVAL]

An invalid flag value or parameter was specified.

The DB_THREAD flag was specified and spinlocks are not implemented for this architecture.

The *DbLockTab::vec* method may fail and throw a *DbException*(3) or return *errno* for any of the errors specified for the following DB and library functions: DbLock::detect(3), fcntl(2), fflush(3), lseek(2), memcpy(3), memset(3), mmap(2), munmap(2), strerror(3), and write(2).

In addition, the *DbLockTab::vec* method may fail and throw a *DbException*(3) or return *errno* for the following conditions:

[EACCES]
An attempt was made to release lock held by another locker.

[EINVAL]
An invalid flag value or parameter was specified.

The *DbLockTab::get* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DbLock::detect(3), fcntl(2), fflush(3), lseek(2), memcpy(3), memset(3), mmap(2), munmap(2), strerror(3), and write(2).

In addition, the *DbLockTab::get* method may fail and throw a *DbException*(3) or return *errno* for the following conditions:

[EINVAL]
An invalid flag value or parameter was specified.

The *DbLockTab::close* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: close(2), fcntl(2), fflush(3), munmap(2), and strerror(3).

The *DbLockTab::unlink* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: close(2), fcntl(2), fflush(3), malloc(3), memcpy(3), memset(3), mmap(2), munmap(2), open(2), sigfillset(3), sigprocmask(2), stat(2), strcpy(3), strdup(3), strerror(3), strlen(3), and unlink(2).

In addition, the *DbLockTab::unlink* method may fail and throw a *DbException*(3) or return *errno* for the following conditions:

[EBUSY]
The shared memory region was in use and the force flag was not set.

The *DbLockTab::detect* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: calloc(3), fcntl(2), fflush(3), lseek(2), malloc(3), memcpy(3), memset(3), mmap(2), munmap(2), strerror(3), and write(2).

BUGS

If a process dies while holding locks, those locks remain held and are **never** released. In this case, all processes should exit as quickly as possible, so that *db_recover* can be run.

SEE ALSO

db_archive(1), *db_checkpoint*(1), *db_deadlock*(1), *db_dump*(1), *db_load*(1), *db_recover*(1), *db_stat*(1), *db_intro*(3), *db_internal*(3), *db_thread*(3), *Db*(3), *Dbc*(3), *DbEnv*(3), *DbException*(3), *DbInfo*(3), *DbLock*(3), *DbLockTab*(3), *DbLog*(3), *DbLsn*(3), *DbMpool*(3), *DbMpoolFile*(3), *Dbt*(3), *DbTxn*(3), *DbTxnMgr*(3)