NAME

db_txn − DB transaction management

SYNOPSIS

```
#include <db.h>

int
txn_open(const char *dir,
        u_int32_t flags, int mode, DB_ENV *dbenv, DB_TXNMGR **regionp);

int
txn_begin(DB_TXNMGR *txnp, DB_TXN *pid, DB_TXN **tid);

int
txn_prepare(DB_TXN *tid);

int
txn_commit(DB_TXN *tid);

int
txn_abort(DB_TXN *tid);

u_int32_t
txn_id(DB_TXN *tid);

int
txn_checkpoint(const DB_TXNMGR *txnp, u_int32_t kbyte, u_int32_t min);

int
txn_close(DB_TXNMGR *txnp);

int
txn_unlink(const char *dir, int force, DB_ENV *dbenv);

int
txn_stat(DB_TXNMGR *txnp,
        DB_TXN_STAT **statp, void *(*db_malloc)(size_t));
```

DESCRIPTION

The DB library is a family of groups of functions 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 functional groups (e.g., the file page caching functions) are useful independent of the other DB functions, although some functional groups are explicitly based on other functional groups (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 DB transaction support.

The *db_txn* functions are the library interface that provides transaction semantics. Full transaction support is provided by a collection of modules that provide interfaces to the services required for transaction processing. These services are recovery (see *db_log*(3)), concurrency control (see *db_lock*(3)), and the management of shared data (see *db_mpool*(3)). Transaction semantics can be applied to the access methods described in *db_open*(3) through function call parameters.

The model intended for transactional use (and the one that is used by the access methods) is write-ahead logging provided by *db_log*(3) to record both before- and after-images. Locking follows a two-phase protocol, with all locks being released at transaction commit.

txn_open

The *txn_open* function copies a pointer, to the transaction region identified by the **directory** *dir*, into the memory location referenced by *regionp*.

If the *dbenv* argument to *txn_open* was initialized using *db_appinit*, *dir* is interpreted as described by *db_appinit*(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, transaction region 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 transaction region are created in this directory. This directory must already exist when

transaction region_open is called. If the transaction region already exists, the process must have permission to read and write the existing files. If the transaction region 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 DB_TXNMGR handle returned by the *txn_open* function to be useable by multiple threads within a single address space, i.e., to be ‘‘free-threaded’’.

DB_TXN_NOSYNC
: On transaction commit, do not synchronously flush the log. This means that transactions exhibit the ACI (atomicity, consistency and isolation) properties, but not D (durability), i.e., database integrity will be maintained but it is possible that some number of the most recently committed transactions may be undone during recovery instead of being redone.

  The number of transactions that are potentially at risk is governed by how often the log is checkpointed (see *db_checkpoint*(1)) and how many log updates can fit on a single log page.

All files created by the transaction 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 transaction subsystem is configured based on the *dbenv* argument to *txn_open*, which is a pointer to a structure of type DB_ENV (typedef’d in <db.h>). Applications will normally use the same DB_ENV structure (initialized by *db_appinit*(3)), as an argument to all of the subsystems in the DB package.

References to the DB_ENV structure are maintained by DB, so it may not be discarded until the last close function, corresponding to an open function for which it was an argument, has returned. In order to ensure compatibility with future releases of DB, all fields of the DB_ENV structure that are not explicitly set should be initialized to 0 before the first time the structure is used. Do this by declaring the structure external or static, or by calling the C library routine *bzero*(3) or *memset*(3).

The fields of the DB_ENV structure used by *txn_open* are described below. If *dbenv* is NULL or any of its fields are set to 0, defaults appropriate for the system are used where possible.

The following fields in the DB_ENV structure may be initialized before calling *txn_open*:

void *(*db_errcall)(char *db_errpfx, char *buffer);
FILE *db_errfile;
const char *db_errpfx;
int db_verbose;
: The error fields of the DB_ENV behave as described for *db_appinit*(3).

DB_LOG *lg_info;
The logging region that is being used for this transaction environment. The *lg_info* field contains a return value from the function *log_open*. **Logging is required for transaction environments, and it is an error to not specify a logging region.**

DB_LOCKTAB *lk_info;
The locking region that is being used for this transaction environment. The *lk_info* field contains a return value from the function *lock_open*. If *lk_info* is NULL, no locking is done in this transaction environment.

u_int32_t tx_max;
The maximum number of simultaneous transactions that are supported. This bounds the size of backing files and is used to derive limits for the size of the lock region and logfiles. When there are more than *tx_max* concurrent transactions, calls to *txn_begin* may cause backing files to grow. If *tx_max* is 0, a default value is used.

```
int (*tx_recover)(DB_LOG *logp, DBT *log_rec,
        DB_LSN *lsnp, int redo, void *info);
```
A function that is called by *txn_abort* during transaction abort. This function takes five arguments:

logp A pointer to the transaction log (DB_LOG *).

log_rec
A log record.

lsnp A pointer to a log sequence number (DB_LSN *).

redo An integer value that is set to one of the following values:

DB_TXN_BACKWARD_ROLL
The log is being read backward to determine which transactions have been committed and which transactions were not (and should therefore be aborted during recovery).

DB_TXN_FORWARD_ROLL
The log is being played forward, any transaction ids encountered that have not been entered into the list referenced by *info* should be ignored.

DB_TXN_OPENFILES
The log is being read to open all the files required to perform recovery.

DB_TXN_REDO
Redo the operation described by the log record.

DB_TXN_UNDO
Undo the operation described by the log record.

info An opaque pointer used to reference the list of transaction IDs encountered during recovery.

If *recover* is NULL, the default is that only DB access method operations are transaction protected, and the default recover function will be used.

The *txn_open* function returns the value of *errno* on failure and 0 on success.

txn_begin
The *txn_begin* function creates a new transaction in the designated transaction manager, copying a pointer to a DB_TXN that uniquely identifies it into the memory referenced by *tid*. If the *pid* argument is non-NULL, the new transaction is a nested transaction with the transaction indicated by *pid* as its parent.

Transactions may not span threads, i.e., each transaction must begin and end in the same thread, and each transaction may only be used by a single thread.

The *txn_begin* function returns the value of *errno* on failure and 0 on success.

txn_prepare
The *txn_prepare* function initiates the beginning of a two phase commit. In a distributed transaction environment, *db* can be used as a local transaction manager. In this case, the distributed transaction manager

must send *prepare* messages to each local manager. The local manager must then issue a *txn_prepare* and await its successful return before responding to the distributed transaction manager. Only after the distributed transaction manager receives successful responses from all of its *prepare* messages should it issue any *commit* messages.

The *txn_prepare* function returns the value of *errno* on failure and 0 on success.

txn_commit

The *txn_commit* function ends the transaction specified by the *tid* argument. If DB_TXN_NOSYNC was not specified, a commit log record is written and flushed to disk, as are all previously written log records. If the transaction is nested, its locks are acquired by the parent transaction, otherwise its locks are released. Any applications that require strict two-phase locking must not release any locks explicitly, leaving them all to be released by *txn_commit*.

The *txn_commit* function returns the value of *errno* on failure and 0 on success.

txn_abort

The *txn_abort* function causes an abnormal termination of the transaction. The log is played backwards and any necessary recovery operations are initiated through the *recover* function specified to *txn_open*. After recovery is completed, all locks held by the transaction are acquired by the parent transaction in the case of a nested transaction or released in the case of a non-nested transaction. As is the case for *txn_commit*, applications that require strict two phase locking should not explicitly release any locks.

The *txn_abort* function returns the value of *errno* on failure and 0 on success.

txn_id

The *txn_id* function returns the unique transaction id associated with the specified transaction. Locking calls made on behalf of this transaction should use the value returned from *txn_id* as the locker parameter to the *lock_get* or *lock_vec* calls.

txn_close

The *txn_close* function detaches a process from the transaction environment specified by the DB_TXNMGR pointer. All mapped regions are unmapped and any allocated resources are freed. Any uncommitted transactions are aborted.

In addition, if the *dir* argument to *txn_open* was NULL and *dbenv* was not initialized using *db_appinit*, all files created for this shared region will be removed, as if *txn_unlink* were called.

When multiple threads are using the DB_TXNMGR handle concurrently, only a single thread may call the *txn_close* function.

The *txn_close* function returns the value of *errno* on failure and 0 on success.

txn_unlink

The *txn_unlink* function destroys the transaction region identified by the directory *dir*, removing all files used to implement the transaction region. (The directory *dir* is not removed.) If there are processes that have called *txn_open* without calling *txn_close* (i.e., there are processes currently using the transaction region), *txn_unlink* will fail without further action, unless the force flag is set, in which case *txn_unlink* will attempt to remove the transaction region files regardless of any processes still using the transaction region.

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 transaction region 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 and DB_RECOVER_FATAL flags to *db_appinit*(3)). Alternatively, if recovery is not required because no database state is maintained across failures, it is possible to clean up a transaction region by removing all of the files in the directory specified to the *txn_open* function, as transaction region files are never created in any directory other than the one specified to *txn_open*. Note, however, that this has the potential to remove files created by the other DB subsystems in this database environment.

The *txn_unlink* function returns the value of *errno* on failure and 0 on success.

txn_checkpoint

The *txn_checkpoint* function syncs the underlying memory pool, writes a checkpoint record to the log and then flushes the log.

If either *kbyte* or *min* is non-zero, the checkpoint is only done if more than *min* minutes have passed since the last checkpoint, or if more than *kbyte* kilobytes of log data have been written since the last checkpoint.

The *txn_checkpoint* function returns the value of *errno* on failure, 0 on success, and DB_INCOMPLETE if there were pages that needed to be written but that *memp_sync*(3) was unable to write immediately. In this case, the *txn_checkpoint* call should be retried.

The *txn_checkpoint* function is the underlying function used by the *db_checkpoint*(1) utility. See the source code for the *db_checkpoint* utility for an example of using *txn_checkpoint* in a UNIX environment.

txn_stat

The *txn_stat* function creates a statistical structure and copies a pointer to it into the user-specified memory location.

Statistical structure are created in allocated memory. If *db_malloc* is non-NULL, it is called to allocate the memory, otherwise, the library function *malloc*(3) is used. The function *db_malloc* must match the calling conventions of the *malloc*(3) library routine. Regardless, the caller is responsible for deallocating the returned memory. To deallocate the returned memory, free each returned memory pointer; pointers inside the memory do not need to be individually freed.

The transaction region statistics are stored in a structure of type DB_TXN_STAT (typedef'd in <db.h>). The following DB_TXN_STAT fields will be filled in:

u_int32_t st_refcnt;
: The number of references to the region.

u_int32_t st_regsize;
: The size of the region.

DB_LSN st_last_ckp;
: The LSN of the last checkpoint.

DB_LSN st_pending_ckp;
: The LSN of any checkpoint that is currently in progress. If *st_pending_ckp* is the same as *st_last_ckp* there is no checkpoint in progress.

time_t st_time_ckp;
: The time the last completed checkpoint finished (as returned by *time*(2)).

u_int32_t st_last_txnid;
: The last transaction ID allocated.

u_int32_t st_maxtxns;
: The maximum number of active transactions supported by the region.

u_int32_t st_naborts;
: The number of transactions that have aborted.

u_int32_t st_nactive;
: The number of transactions that are currently active.

u_int32_t st_nbegins;
: The number of transactions that have begun.

u_int32_t st_ncommits;
: The number of transactions that have committed.

u_int32_t st_region_wait;
: The number of times that a thread of control was forced to wait before obtaining the region lock.

u_int32_t st_region_nowait;
: The number of times that a thread of control was able to obtain the region lock without waiting.

DB_TXN_ACTIVE *st_txnarray;

A pointer to an array of *st_nactive* DB_TXN_ACTIVE structures, describing the currently active transactions. The following fields of the DB_TXN_ACTIVE structure (typedef'd in <db.h>) will be filled in:

u_int32_t txnid;

The transaction ID as returned by *txn_begin*(3).

DB_LSN lsn;

The LSN of the transaction-begin record.

TRANSACTIONS

Creating transaction protected applications using the DB access methods requires little system customization. In most cases, the default parameters to the locking, logging, memory pool, and transaction subsystems will suffice. Applications can use *db_appinit*(3) to perform this initialization, or they may do it explicitly.

Each database operation (i.e., any call to a function underlying the handles returned by *db_open*(3) and *db_cursor*(3)) is normally performed on behalf of a unique locker. If multiple calls on behalf of the same locker are desired, then transactions must be used.

Once the application has initialized the DB subsystems that it is using, it may open the DB access method databases. For applications performing transactions, the databases must be opened after subsystem initialization, and cannot be opened as part of a transaction. Once the databases are opened, the application can group sets of operations into transactions, by surrounding the operations with the appropriate *txn_begin*, *txn_commit* and *txn_abort* calls. Databases accessed by a transaction must not be closed during the transaction. Note, it is not necessary to transaction protect read-only transactions, unless those transactions require repeatable reads.

The DB access methods will make the appropriate calls into the lock, log and memory pool subsystems in order to guarantee that transaction semantics are applied. When the application is ready to exit, all outstanding transactions should have been committed or aborted. At this point, all open DB files should be closed. Once the DB database files are closed, the DB subsystems should be closed, either explicitly or by calling *db_appexit*(3).

It is also possible to use the locking, logging and transaction subsystems of DB to provide transaction semantics to objects other than those described by the DB access methods. In these cases, the application will need more explicit customization of the subsystems as well as the development of appropriate data-structure-specific recovery functions.

For example, consider an application that provides transaction semantics to data stored in plain UNIX files accessed using the *read*(2) and *write*(2) system calls. The operations for which transaction protection is desired are bracketed by calls to *txn_begin* and *txn_commit*.

Before data are referenced, the application must make a call to the lock manager, *db_lock*, for a lock of the appropriate type (e.g., read) on the object being locked. The object might be a page in the file, a byte, a range of bytes, or some key. It is up to the application to ensure that appropriate locks are acquired. Before a write is performed, the application should acquire a write lock on the object, by making an appropriate call to the lock manager, *db_lock*. Then, the application should make a call to the log manager, *db_log*, to record enough information to redo the operation in case of failure after commit and to undo the operation in case of abort. As discussed in the *db_log*(3) manual page, the application is responsible for providing any necessary structure to the log record. For example, the application must understand what part of the log record is an operation code, what part identifies the file being modified, what part is redo information, and what part is undo information.

After the log message is written, the application may issue the write system call. After all requests are issued, the application may call *txn_commit*. When *txn_commit* returns, the caller is guaranteed that all necessary log writes have been written to disk.

At any time, the application may call *txn_abort*, which will result in the appropriate calls to the *recover* function to restore the ''database'' to a consistent pre-transaction state. (The recover function must be able to either re-apply or undo the update depending on the context, for each different type of log record.)

If the application should crash, the recovery process uses the *db_log* interface to read the log and call the *recover* function to restore the database to a consistent state.

The *txn_prepare* function provides the core functionality to implement distributed transactions, but it does not manage the notification of distributed transaction managers. The caller is responsible for issuing *txn_prepare* calls to all sites participating in the transaction. If all responses are positive, the caller can issue a *txn_commit*. If any of the responses are negative, the caller should issue a *txn_abort*. In general, the *txn_prepare* call requires that the transaction log be flushed to disk.

TRANSACTION ID LIMITS

The transaction ID space in Berkeley DB is 2^31, or 2 billion entries. It is possible that some environments may need to be aware of this limitation. Consider an application performing 600 transactions a second for 15 hours a day. The transaction ID space will run out in roughly 66 days:

$$2^{31} / (600 * 15 * 60 * 60) = 66$$

Doing only 100 transactions a second exhausts the transaction ID space in roughly one year.

The transaction ID space is reset each time recovery is run. If you reach the end of your transaction ID space, shut down your applications and restart them after running recovery (see *db_recover*(1) for more information). The most recently allocated transaction ID is the *st_last_txnid* value in the transaction statistics information, and is displayed by the *db_stat*(1) utility.

ENVIRONMENT VARIABLES

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

DB_HOME

If the *dbenv* argument to *txn_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 *txn_open*, as described in *db_appinit*(3).

TMPDIR

If the *dbenv* argument to *txn_open* was NULL or not initialized using *db_appinit*, the environment variable TMPDIR may be used as the directory in which to create the transaction region, as described in the *txn_open* section above.

ERRORS

The *txn_open* function may fail and return *errno* for any of the errors specified for the following DB and library functions: 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), time(3), txn_unlink(3), unlink(2), and write(2).

In addition, the *txn_open* function may fail and return *errno* for the following conditions:

[EINVAL]

An invalid flag value or parameter was specified.

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

The *dbenv* parameter was NULL.

[EAGAIN]

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

The *txn_begin* function may fail and return *errno* for any of the errors specified for the following DB and library functions: fcntl(2), fflush(3), log_put(3), lseek(2), malloc(3), memcpy(3), memset(3), mmap(2), munmap(2), strerror(3), and write(2).

In addition, the *txn_begin* function may fail and return *errno* for the following conditions:

[ENOSPC]
The maximum number of concurrent transactions has been reached.

The *txn_prepare* function may fail and return *errno* for any of the errors specified for the following DB and library functions: fcntl(2), fflush(3), log_flush(3), and strerror(3).

The *txn_commit* function may fail and return *errno* for any of the errors specified for the following DB and library functions: fcntl(2), fflush(3), lock_vec(3), log_put(3), malloc(3), memcpy(3), and strerror(3).

In addition, the *txn_commit* function may fail and return *errno* for the following conditions:

[EINVAL]
The transaction was aborted.

The *txn_abort* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBenv->tx_recover(3), fcntl(2), fflush(3), lock_vec(3), log_get(3), memset(3), and strerror(3).

[EINVAL]
The transaction was already aborted.

The *txn_checkpoint* function may fail and return *errno* for any of the errors specified for the following DB and library functions: fcntl(2), fflush(3), log_compare(3), log_put(3), malloc(3), memcpy(3), memp_sync(3), memset(3), strerror(3), and time(3).

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

The *txn_close* function may fail and return *errno* for any of the errors specified for the following DB and library functions: close(2), fcntl(2), fflush(3), log_flush(3), munmap(2), strerror(3), and txn_abort(3).

The *txn_unlink* function may fail and 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 *txn_unlink* function may fail and return *errno* for the following conditions:

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

The *txn_stat* function may fail and return *errno* for any of the errors specified for the following DB and library functions: fcntl(2), and malloc(3).

SEE ALSO

LIBTP: Portable, Modular Transactions for UNIX, Margo Seltzer, Michael Olson, USENIX proceedings, Winter 1992.

BUGS

Nested transactions are not yet implemented.

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_appinit*(3), *db_cursor*(3), *db_dbm*(3), *db_internal*(3), *db_lock*(3), *db_log*(3), *db_mpool*(3), *db_open*(3), *db_thread*(3), *db_txn*(3)