NAME

DbMpool – shared memory buffer pool

SYNOPSIS

```
#include <db_cxx.h>

static int
DbMpool::open(char *dir,
        u_int32_t flags, int mode, DbEnv *dbenv, DbMpool **regionp);

int
DbMpool::close();

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

int
DbMpool::db_register(int ftype,
        int (*pgin)(db_pgno_t pgno, void *pgaddr, Dbt *pgcookie),
        int (*pgout)(db_pgno_t pgno, void *pgaddr, Dbt *pgcookie));

int
DbMpool::trickle(int pct, int *nwrotep);

int
DbMpool::sync(LSN *lsn);

int
DbMpool::stat(DB_MPOOL_STAT **gsp,
        DB_MPOOL_FSTAT *(*fsp)[], void *(*db_malloc)(size_t));
```

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 memory pool interface.

The *DbMpool*(3) and *DbMpoolFile*(3) classes are the library interface intended to provide general-purpose, page-oriented buffer management of one or more files. While designed to work with the other Db classes, this class is also useful for more general purposes. The memory pools (DbMpool's) are referred to in this document as simply ''pools''. Pools may be shared between processes. Pools are usually filled by pages from one or more files (DbMpoolFile's). Pages in the pool are replaced in LRU (least-recently-used) order, with each new page replacing the page that has been unused the longest. Pages retrieved from the pool using *DbMpoolFile::get* are ''pinned'' in the pool, by default, until they are returned to the pool's control using the *DbMpoolFile::put* method.

DbMpool::open

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

If the *dbenv* argument to *DbMpool::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, memory pool 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 memory pool are created in this directory. This directory must already exist when DbMpool::open is called. If the memory pool already exists, the process must have permission to

read and write the existing files. If the memory pool 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_MPOOL_PRIVATE
: Create a private MPOOL that is not shared with any other process (although it may be shared with other threads).

DB_NOMMAP
: Always copy files in this memory pool into the local cache instead of mapping them into process memory (see the description of the *mp_mmapsize* field of the DbEnv object for further information).

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

All files created by the memory pool subsystem (other than files created by the *memp_fopen* method, which are separately specified) 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 memory pool 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 *DbMpool::open* are described below. As references to the DbEnv object may be maintained by *DbMpool::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 *DbMpool::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).

size_t mp_mmapsize;
: Files that are opened read-only in the pool (and that satisfy a few other criteria) are, by default, mapped into the process address space instead of being copied into the local cache. This can result in better-than-usual performance, as available virtual memory is normally much larger than the local cache, and page faults are faster than page copying on many systems. However, in the presence of limited virtual memory it can cause resource starvation, and in the presence of large databases, it can result in immense process sizes. If *mp_mmapsize* is non-zero, it specifies the maximum file size, in bytes, for a file to be mapped into the process address space. By default, it is set to 10Mb.

size_t mp_size;
: The suggested size of the pool, in bytes. This should be the size of the normal working data set of the application, with some small amount of additional memory for unusual situations. (Note, the working set is not the same as the number of simultaneously referenced pages, and should be quite a bit larger!) The default cache size is 128K bytes (16 8K byte pages), and may not be less than 20K bytes.

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

DbMpool::close

The *DbMpool::close* method closes the pool indicated by the DbMpool object, as returned by *DbMpool::open*. This method does not imply a call to *DbMpoolFile::sync*, but does imply a call to *DbMpoolFile::close* for any remaining open DbMpoolFile objects returned to this process by calls to *DbMpoolFile::open*.

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

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

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

DbMpool::unlink

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

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 memory pool 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 *DbEnv::appinit*(3)). Alternatively, if recovery is not required because no database state is maintained across failures, it is possible to clean up a memory pool by removing all of the files in the directory specified to the *DbMpool::open* method, as memory pool files are never created in any directory other than the one specified to *DbMpool::open*. Note, however, that this has the potential to remove files created by the other DB subsystems in this database environment.

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

DbMpool::db_register

The *DbMpool::db_register* method registers page-in and page-out functions for files of type *ftype* in the specified pool.

If the *pgin* function is non-NULL, it is called each time a page is read into the memory pool from a file of type *ftype*, or a page is created for a file of type *ftype* (see the DB_MPOOL_CREATE flag for the *DbMpoolFile::get* method). If the *pgout* function is non-NULL, it is called each time a page is written to a file of type *ftype*.

Both the *pgin* and *pgout* functions are called with the page number, a pointer to the page being read or written, and any argument *pgcookie* that was specified to the *DbMpoolFile::open* method when the file was opened. The *pgin* and *pgout* functions should return 0 on success, and an applicable non-zero *errno* value on failure, in which case the *DbMpool* method calling it will also fail, returning that *errno* value.

The purpose of the *DbMpool::db_register* method is to support processing when pages are entered into, or flushed from, the pool. A file type must be specified to make it possible for unrelated threads or processes, that are sharing a pool, to evict each other's pages from the pool. Applications should call *DbMpool::db_register*, during initialization, for each type of file requiring input or output processing that will be

sharing the underlying pool. (No registry is necessary for the standard access method types, btree, hash and recno, as *Db::open*(3) registers them separately.)

If a thread or process does not call *DbMpool::db_register* for a file type, it is impossible for it to evict pages for any file requiring input or output processing from the pool. For this reason, *DbMpool::db_register* should always be called by each application sharing a pool for each type of file included in the pool, regardless of whether or not the application itself uses files of that type.

There are no standard values for *ftype*, *pgin*, *pgout* and *pgcookie*, except that the *ftype* value for a file must be a non-zero positive number, as negative numbers are reserved for internal use by the DB library. For this reason, applications sharing a pool must coordinate their values amongst themselves.

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

DbMpool::trickle

The *DbMpool::trickle* method ensures that at least *pct* percent of the pages in the shared memory pool are clean by writing dirty pages to their backing files. If the *nwrotep* argument is non-NULL, the number of pages that were written to reach the correct percentage is returned in the memory location it references.

The purpose of the *DbMpool::trickle* method is to enable a memory pool manager to ensure that a page is always available for reading in new information without having to wait for a write.

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

DbMpool::sync

The *DbMpool::sync* method ensures that all the modified pages in the pool with log sequence numbers (LSNs) less than the *lsn* argument are written to disk.

The *DbMpool::sync* method throws a *DbException*(3) or returns the value of *errno* on failure, 0 on success, and DB_INCOMPLETE if there were pages which need to be written but which *DbMpool::sync* was unable to write immediately. In addition, if *DbMpool::sync* returns success, the value of *lsn* will be overwritten with the largest LSN from any page which was written by *DbMpool::sync* to satisfy this request.

The purpose of the *DbMpool::sync* method is to enable a transaction manager to ensure, as part of a checkpoint, that all pages modified by a certain time have been written to disk. Pages in the pool which cannot be written back to disk immediately (e.g., are currently pinned) are written to disk as soon as it is possible to do so. The expected behavior of the transaction manager is to call the *DbMpool::sync* method and then, if the return indicates that some pages could not be written immediately, to wait briefly and retry again with the same LSN until the *DbMpool::sync* method returns that all pages have been written.

To support the *DbMpool::sync* functionality, it is necessary that the pool methods know the location of the LSN on the page for each file type. This location should be specified when the file is opened using the *DbMpoolFile::open* method. (Note, it is not required that the LSN be aligned on the page in any way.)

DbMpool::stat

The *DbMpool::stat* method creates statistical structures and copies pointers to them into user-specified memory locations. The statistics include the number of files participating in the pool, the active pages in the pool, and information as to how effective the cache has been.

Statistical structures 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.

If *gsp* is non-NULL, the global statistics for the memory pool object are copied into the memory location it references. The global statistics are stored in a structure of type DB_MPOOL_STAT (typedef'd in <db_cxx.h>).

The following DB_MPOOL_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.
size_t st_cachesize;
Cache size in bytes.
u_int32_t st_cache_hit;
Requested pages found in the cache.
u_int32_t st_cache_miss;
Requested pages not found in the cache.
u_int32_t st_map;
Requested pages mapped into the process' address space (there is no available information as to whether or not this request caused disk I/O, although examining the application page fault rate may be helpful).
u_int32_t st_page_create;
Pages created in the cache.
u_int32_t st_page_in;
Pages read into the cache.
u_int32_t st_page_out;
Pages written from the cache to the backing file.
u_int32_t st_ro_evict;
Clean pages forced from the cache.
u_int32_t st_rw_evict;
Dirty pages forced from the cache.
u_int32_t st_hash_buckets;
Number of hash buckets in buffer hash table.
u_int32_t st_hash_searches;
Total number of buffer hash table lookups.
u_int32_t st_hash_longest;
The longest chain ever encountered in buffer hash table lookups.
u_int32_t st_hash_examined;
Total number of hash elements traversed during hash table lookups.
u_int32_t st_page_clean;
Clean pages currently in the cache.
u_int32_t st_page_dirty;
Dirty pages currently in the cache.
u_int32_t st_page_trickle;
Dirty pages written using the *memp_trickle* interface.

If *fsp* is non-NULL, a pointer to a NULL-terminated variable length array of statistics for individual files in the memory pool is copied into the memory location it references. If no individual files currently exist in the memory pool, *fsp* will be set to NULL.

The per-file statistics are stored in structures of type DB_MPOOL_FSTAT (typedef'd in <db_cxx.h>). The following DB_MPOOL_FSTAT fields will be filled in for each file in the pool, i.e., each element of the array:

char *file_name;
The name of the file.
size_t st_pagesize;
Page size in bytes.
u_int32_t st_cache_hit;
Requested pages found in the cache.
u_int32_t st_cache_miss;
Requested pages not found in the cache.

u_int32_t st_map;
Requested pages mapped into the process' address space.
u_int32_t st_page_create;
Pages created in the cache.
u_int32_t st_page_in;
Pages read into the cache.
u_int32_t st_page_out;
Pages written from the cache to the backing file.

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

ENVIRONMENT VARIABLES

The following environment variables affect the behavior of *DbMpoolFile*:

DB_HOME
If the *dbenv* argument to *DbMpool::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 *DbMpool::open*, as described in *db_appinit*(3).

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

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 *DbMpool::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: DBmemp->pgin(3), DBmemp->pgout(3), DbLog::compare(3), DbLog::flush(3), DbMpool::close(3), DbMpool::unlink(3), close(2), db_version(3), fcntl(2), fflush(3), fsync(2), lseek(2), malloc(3), memcmp(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), unlink(2), and write(2).

In addition, the *DbMpool::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.

A NULL pathname was specified without the DB_MPOOL_PRIVATE flag.

The specified cache size was impossibly small.

The *DbMpool::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: DbMpoolFile::close(3), close(2), fcntl(2), fflush(3), munmap(2), and strerror(3).

The *DbMpool::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 *DbMpool::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 *DbMpool::db_register* method may fail and throw a *DbException*(3)

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

The *DbMpool::trickle* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), DbLog::compare(3), DbLog::flush(3), close(2), fcntl(2), fflush(3), fsync(2), lseek(2), malloc(3), memcmp(3), memcpy(3), memset(3), mmap(2), open(2), sigfillset(3), sigprocmask(2), stat(2), strcpy(3), strdup(3), strerror(3), strlen(3), time(3), unlink(2), and write(2).

In addition, the *DbMpool::trickle* 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 *DbMpool::sync* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), DbLog::compare(3), DbLog::flush(3), close(2), fcntl(2), fflush(3), fsync(2), lseek(2), malloc(3), memcmp(3), memcpy(3), memset(3), mmap(2), open(2), qsort(3), realloc(3), sigfillset(3), sigprocmask(2), stat(2), strcpy(3), strdup(3), strerror(3), strlen(3), time(3), unlink(2), and write(2).

In addition, the *DbMpool::sync* 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 *DbMpool::sync* method was called without logging having been initialized in the environment.

The *DbMpool::stat* method may fail and throw a *DbException*(3)

or return *errno* for any of the errors specified for the following DB and library functions: fcntl(2), malloc(3), memcpy(3), and strlen(3).

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)