NAME

db_mpool − shared memory buffer pool

SYNOPSIS

```
#include <db.h>

int
memp_open(char *dir,
        u_int32_t flags, int mode, DB_ENV *dbenv, DB_MPOOL **regionp);

int
memp_close(DB_MPOOL *mp);

int
memp_fopen(DB_MPOOL *mp, char *file, u_int32_t flags, int mode,
        size_t pagesize, DB_MPOOL_FINFO *finfop, DB_MPOOLFILE **mpf);

int
memp_fclose(DB_MPOOLFILE *mpf);

int
memp_fget(DB_MPOOLFILE *mpf,
        db_pgno_t *pgnoaddr, u_int32_t flags, void **pagep);

int
memp_fput(DB_MPOOLFILE *mpf, void *pgaddr, u_int32_t flags);

int
memp_fset(DB_MPOOLFILE *mpf, void *pgaddr, u_int32_t flags);

int
memp_fsync(DB_MPOOLFILE *mpf);

int
memp_unlink(const char *dir, int force, DB_ENV *);

int
memp_register(DB_MPOOL *mp, int ftype,
        int (*pgin)(db_pgno_t pgno, void *pgaddr, DBT *pgcookie),
        int (*pgout)(db_pgno_t pgno, void *pgaddr, DBT *pgcookie));

int
memp_trickle(DB_MPOOL *mp, int pct, int *nwrotep);

int
memp_sync(DB_MPOOL *mp, LSN *lsn);

int
memp_stat(DB_MPOOL *mp, DB_MPOOL_STAT **gsp,
        DB_MPOOL_FSTAT *(*fsp)[], 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 memory pool interface.

The *db_mpool* functions 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 functions, these functions are also useful for more general purposes. The memory pools (DB_MPOOL'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 (DB_MPOOLFILE'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 *memp_fget* are ‘‘pinned’’ in the pool, by default, until they are returned to the pool's control using the *memp_fput* function.

memp_open

The *memp_open* function copies a pointer, to the memory pool identified by the **directory** *dir*, into the memory location referenced by *regionp*.

If the *dbenv* argument to *memp_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, 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

memory pool_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 DB_ENV structure for further information).

DB_THREAD

Cause the DB_MPOOL handle returned by the *memp_open* function 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* function, 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 the *dbenv* argument to *memp_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 *memp_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 *memp_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).

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

memp_close

The *memp_close* function closes the pool indicated by the DB_MPOOL pointer *mp*, as returned by *memp_open*. This function does not imply a call to *memp_fsync*, but does imply a call to *memp_fclose* for any remaining open DB_MPOOLFILE pointers returned to this process by calls to *memp_fopen*.

In addition, if the *dir* argument to *memp_open* was NULL and *dbenv* was not initialized using *db_appinit*, or the DB_MPOOL_PRIVATE flag was set, all files created for this shared region will be removed, as if *memp_unlink* were called.

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

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

memp_fopen

The *memp_fopen* function opens a file in the pool specified by the DB_MPOOL argument, copying the DB_MPOOLFILE pointer representing it into the memory location referenced by *mpf*.

The *file* argument is the name of the file to be opened. If *file* is NULL, a private file is created that cannot be shared with any other process (although it may be shared with other threads).

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_NOMMAP

Always copy this file into the local cache instead of mapping it into process memory (see the description of the *mp_mmapsize* field of the DB_ENV structure for further information).

DB_RDONLY

Open any underlying files for reading only. Any attempt to write the file using the pool functions will fail, regardless of the actual permissions of the file.

All files created by the function *memp_fopen* 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 *pagesize* argument is the size, in bytes, of the unit of transfer between the application and the pool, although it is not necessarily the unit of transfer between the pool and the source file.

Files opened in the pool may be further configured based on the *finfop* argument to *memp_fopen*, which is a pointer to a structure of type DB_MPOOL_FINFO (typedef'd in <db.h>). No references to the *finfop* structure are maintained by DB, so it may be discarded when the *memp_fopen* function returns. In order to ensure compatibility with future releases of DB, all fields of the DB_MPOOL_FINFO 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_MPOOL_FINFO structure used by *memp_fopen* are described below. If *finfop* is NULL or any of its fields are set to their default value, defaults appropriate for the system are used.

int ftype;

The *ftype* field should be the same as a *ftype* argument previously specified to the *memp_register* function, unless no input or output processing of the file's pages are necessary, in which case it should be 0. (See the description of the *memp_register* function for more information.)

DBT *pgcookie;

The *pgcookie* field contains the byte string that is passed to the *pgin* and *pgout* functions for this file, if any. If no *pgin* or *pgout* functions are specified, the *pgcookie* field should be NULL. (See the description of the *memp_register* function for more information.)

u_int8_t *fileid;

The *fileid* field is a unique identifier for the file. The mpool functions must be able to uniquely identify files in order that multiple processes sharing a file will correctly share its underlying pages. Normally, the *fileid* field should be NULL and the mpool functions will use the file's device and inode numbers (see *stat*(2)) for this purpose. On some filesystems, (e.g., FAT or NFS) file device and inode numbers are not necessarily unique across system reboots. **Applications wanting to maintain a shared memory buffer pool across system reboots, where the pool contains pages from files stored on such filesystems, must specify a unique file identifier to the** *memp_fopen* call and each process opening or registering the file must provide the same unique identifier. If the *fileid* field is non-NULL, it must reference a DB_FILE_ID_LEN (as defined in <db.h>) length array of bytes that will be used to uniquely identify the file. This should not be necessary for most applications. Specifically, it is not necessary if the memory pool is re-instantiated after each system reboot, the application is using the DB access methods instead of calling the pool functions explicitly, or the files in the memory pool are stored on filesystems where the file device and inode numbers do not change across system reboots.

int32_t lsn_offset;

The *lsn_offset* field is the zero-based byte offset in the page of the page's log sequence number (LSN), or −1 if no LSN offset is specified. (See the description of the *memp_sync* function for more information.)

u_int32_t clear_len;

The *clear_len* field is the number of initial bytes in a page that should be set to zero when the page is created as a result of the DB_MPOOL_CREATE or DB_MPOOL_NEW flags being specified to *memp_fget*. If *finfop* is NULL or *clear_len* is 0, the entire page is cleared.

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

memp_fclose

The *memp_fclose* function closes the source file indicated by the DB_MPOOLFILE pointer *mpf*. This function does not imply a call to *memp_fsync*, i.e. no pages are written to the source file as as a result of calling *memp_fclose*.

In addition, if the *file* argument to *memp_fopen* was NULL, any underlying files created for this DB_MPOOLFILE will be removed.

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

memp_fget

The *memp_fget* function copies a pointer to the page with the page number specified by *pgnoaddr*, from the source file specified by the DB_MPOOLFILE pointer *mpf*, into the memory location referenced by *pagep*. If the page does not exist or cannot be retrieved, *memp_fget* will fail.

The returned page is size_t type aligned.

Page numbers begin at 0, e.g., the first page in the file is page number 0, not page number 1.

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

DB_MPOOL_CREATE
: If the specified page does not exist, create it. In this case, the *pgin* function, if specified, is called.

DB_MPOOL_LAST
: Return the last page of the source file and copy its page number to the location referenced by *pgnoaddr*.

DB_MPOOL_NEW
: Create a new page in the file and copy its page number to the location referenced by *pgnoaddr*. In this case, the *pgin* function, if specified, is not called.

The DB_MPOOL_CREATE, DB_MPOOL_LAST and DB_MPOOL_NEW flags are mutually exclusive.

Created pages have all their bytes set to 0, unless otherwise specified when the file was opened.

All pages returned by *memp_fget* will be retained (i.e. ''pinned'') in the pool until a subsequent call to *memp_fput*.

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

memp_fput

The *memp_fput* function indicates that the page referenced by *pgaddr* can be evicted from the pool. *Pgaddr* must be an address previously returned by *memp_fget*.

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

DB_MPOOL_CLEAN
: Clear any previously set modification information (i.e., don't bother writing the page back to the source file).

DB_MPOOL_DIRTY
: The page has been modified and must be written to the source file before being evicted from the pool.

DB_MPOOL_DISCARD
: The page is unlikely to be useful in the near future, and should be discarded before other pages in the pool.

The DB_MPOOL_CLEAN and DB_MPOOL_DIRTY flags are mutually exclusive.

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

memp_fset

The *memp_fset* function sets the flags associated with the page referenced by *pgaddr* without unpinning it from the pool. *Pgaddr* must be an address previously returned by *memp_fget*. The *flags* argument to *memp_fset* is specified by **or**'ing together one or more of the values specified as flags for the *memp_fput* call.

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

memp_fsync

The *memp_fsync* function writes all pages associated with the DB_MPOOLFILE pointer *mpf*, that were marked as modified using *memp_fput* or *memp_fset*, back to the source file. If any of the modified pages are also pinned (i.e., currently referenced by this or another process) *memp_fsync* will ignore them.

The *memp_fsync* function returns the value of *errno* on failure, 0 on success, and DB_INCOMPLETE if there were pages which were modified but which *memp_fsync* was unable to write.

memp_unlink

The *memp_unlink* function 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 *memp_open* without calling *memp_close* (i.e., there are processes currently using the memory pool), *memp_unlink* will fail without further action, unless the force flag is set, in which case *memp_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 *db_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 *memp_open* function, as memory pool files are never created in any directory other than the one specified to *memp_open*. Note, however, that this has the potential to remove files created by the other DB subsystems in this database environment.

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

memp_register

The *memp_register* function 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 *memp_fget* function). 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 *memp_fopen* function 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 *db_mpool* function calling it will also fail, returning that *errno* value.

The purpose of the *memp_register* function 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 *memp_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 *memp_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, *memp_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 *memp_register* function returns the value of *errno* on failure and 0 on success.

memp_trickle

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

memp_sync

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

The *memp_sync* function returns the value of *errno* on failure, 0 on success, and DB_INCOMPLETE if there were pages which need to be written but which *memp_sync* was unable to write immediately. In addition, if *memp_sync* returns success, the value of *lsn* will be overwritten with the largest LSN from any page which was written by *memp_sync* to satisfy this request.

The purpose of the *memp_sync* function 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 *memp_sync* function 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 *memp_sync* function returns that all pages have been written.

To support the *memp_sync* functionality, it is necessary that the pool functions 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 *memp_fopen* function. (Note, it is not required that the LSN be aligned on the page in any way.)

memp_stat

The *memp_stat* function 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 *mp* 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.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.

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.

If *fsp* is non-NULL, a pointer to a NULL-terminated variable length array of statistics for individual files, in the memory pool *mp*, 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.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 *memp_stat* function returns the value of *errno* on failure and 0 on success.

ENVIRONMENT VARIABLES

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

DB_HOME
  If the *dbenv* argument to *memp_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 *memp_open*, as described in *db_appinit*(3).

TMPDIR
  If the *dbenv* argument to *memp_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 *memp_open* section above.

ERRORS

The *memp_open* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), close(2), db_version(3), fcntl(2), fflush(3), fsync(2), log_compare(3), log_flush(3), lseek(2), malloc(3), memcmp(3), memcpy(3), memp_close(3), memp_unlink(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 *memp_open* function may fail and 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 *memp_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), memp_fclose(3), munmap(2), and strerror(3).

The *memp_fopen* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), close(2), fcntl(2), fflush(3), fsync(2), log_compare(3), log_flush(3), 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 *memp_fopen* function may fail and return *errno* for the following conditions:

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

  The file has already been entered into the pool, and the *pagesize* value is not the same as when the file was entered into the pool, or the length of the file is not zero or a multiple of the *pagesize*.

  The DB_RDONLY flag was specified for an in-memory pool.

The *memp_fclose* 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), munmap(2), and strerror(3).

The *memp_fget* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), close(2), fcntl(2), fflush(3), fsync(2), log_compare(3), log_flush(3), lseek(2), malloc(3), memcmp(3), memcpy(3), memset(3), mmap(2), open(2), read(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 *memp_fget* function may fail and return *errno* for the following conditions:

[EAGAIN]
The page reference count has overflowed. (This should never happen unless there's a bug in the application.)

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

The DB_MPOOL_NEW flag was set and the source file was not opened for writing.

The requested page does not exist and DB_MPOOL_CREATE was not set.

More than one of DB_MPOOL_CREATE, DB_MPOOL_LAST and DB_MPOOL_NEW was set.

[ENOMEM]
The cache is full and no more pages will fit in the pool.

The *memp_fput* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), close(2), fcntl(2), fflush(3), fsync(2), log_compare(3), log_flush(3), 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 *memp_fput* function may fail and return *errno* for the following conditions:

[EACCES]
The DB_MPOOL_DIRTY flag was set and the source file was not opened for writing.

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

The *pgaddr* parameter does not reference a page returned by *memp_fget*.

More than one of DB_MPOOL_CLEAN and DB_MPOOL_DIRTY was set.

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

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

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

The *memp_fsync* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), close(2), fcntl(2), fflush(3), fsync(2), log_compare(3), log_flush(3), lseek(2), malloc(3), memcpy(3), memset(3), open(2), qsort(3), realloc(3), sigfillset(3), sigprocmask(2), stat(2), strcpy(3), strdup(3), strerror(3), strlen(3), unlink(2), and write(2).

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

The *memp_trickle* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), close(2), fcntl(2), fflush(3), fsync(2), log_compare(3), log_flush(3), 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 *memp_trickle* function may fail and return *errno* for the following conditions:

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

The *memp_sync* function may fail and return *errno* for any of the errors specified for the following DB and library functions: DBmemp->pgin(3), DBmemp->pgout(3), close(2), fcntl(2), fflush(3), fsync(2), log_compare(3), log_flush(3), 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 *memp_sync* function may fail and return *errno* for the following conditions:

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

The *memp_sync* function was called without logging having been initialized in the environment.

The *memp_stat* function may fail and 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_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)