NAME

xdl_set_allocator, xdl_malloc, xdl_free, xdl_realloc, xdl_init_mmfile, xdl_free_mmfile, xdl_mmfile_iscompact, xdl_seek_mmfile, xdl_read_mmfile, xdl_write_mmfile, xdl_writem_mmfile, xdl_mmfile_writeallocate, xdl_mmfile_ptradd, xdl_mmfile_first, xdl_mmfile_next, xdl_mmfile_size, xdl_mmfile_cmp, xdl_mmfile_compact, xdl_diff, xdl_patch, xdl_merge3, xdl_bdiff_mb, xdl_bdiff, xdl_bdiff_tgsize, xdl_bpatch − File Differential Library support functions

SYNOPSIS

#include <xdiff.h>

int xdl_set_allocator(memallocator_t const ****malt*);**
void *xdl_malloc(unsigned int *size***);**
void xdl_free(void * *ptr***);**
void *xdl_realloc(void * *ptr***, unsigned int** *nsize***);**
int xdl_init_mmfile(mmfile_t * *mmf***, long** *bsize***, unsigned long** *flags***);**
void xdl_free_mmfile(mmfile_t * *mmf***);**
int xdl_mmfile_iscompact(mmfile_t * *mmf***);**
int xdl_seek_mmfile(mmfile_t * *mmf***, long** *off***);**
long xdl_read_mmfile(mmfile_t * *mmf***, void *** *data***, long** *size***);**
long xdl_write_mmfile(mmfile_t * *mmf***, void const *** *data***, long** *size***);**
long xdl_writem_mmfile(mmfile_t * *mmf***, mmbuffer_t *** *mb***, int** *nbuf***);**
void *xdl_mmfile_writeallocate(mmfile_t * *mmf***, long** *size***);**
long xdl_mmfile_ptradd(mmfile_t * *mmf***, char *** *ptr***, long** *size***, unsigned long** *flags***);**
void *xdl_mmfile_first(mmfile_t * *mmf***, long *** *size***);**
void *xdl_mmfile_next(mmfile_t * *mmf***, long *** *size***);**
long xdl_mmfile_size(mmfile_t * *mmf***);**
int xdl_mmfile_cmp(mmfile_t * *mmf1***, mmfile_t *** *mmf2***);**
int xdl_mmfile_compact(mmfile_t * *mmfo***, mmfile_t *** *mmfc***, long** *bsize***, unsigned long** *flags***);**
int xdl_diff(mmfile_t * *mmf1***, mmfile_t *** *mmf2***, xpparam_t const *** *xpp***, xdemitconf_t const *** *xecfg***, xdemitcb_t *** *ecb***);**
int xdl_patch(mmfile_t * *mmf***, mmfile_t *** *mmfp***, int** *mode***, xdemitcb_t *** *ecb***, xdemitcb_t *** *rjecb***);**
int xdl_merge3(mmfile_t * *mmfo***, mmfile_t *** *mmf1***, mmfile_t *** *mmf2***, xdemitcb_t *** *ecb***, xdemitcb_t *** *rjecb***);**
int xdl_bdiff_mb(mmbuffer_t * *mmb1***, mmbuffer_t *** *mmb2***, bdiffparam_t const *** *bdp***, xdemitcb_t *** *ecb***);**
int xdl_bdiff(mmfile_t * *mmf1***, mmfile_t *** *mmf2***, bdiffparam_t const *** *bdp***, xdemitcb_t *** *ecb***);**
long xdl_bdiff_tgsize(mmfile_t * *mmfp***);**
int xdl_bpatch(mmfile_t * *mmf***, mmfile_t *** *mmfp***, xdemitcb_t *** *ecb***);**

DESCRIPTION

The **LibXDiff** library implements basic and yet complete functionalities to create file differences/patches to both binary and text files. The library uses memory files as file abstraction to achieve both performance and portability. For binary files, **LibXDiff** implements (with some modification) the algorithm described in *File System Support for Delta Compression* by *Joshua P. MacDonald*, while for text files it follows directives described in *An O(ND) Difference Algorithm and Its Variations* by *Eugene W. Myers*. Memory files used by the library are basically a collection of buffers that store the file content. There are two different requirements for memory files when passed to diff/patch functions. Text files for diff/patch functions require that a single line do not have to spawn across two different memory file blocks. Binary diff/patch functions require memory files to be compact. A compact memory files is a file whose content is stored inside a single block. Functionalities inside the library are available to satisfy these rules. Using the **XDL_MMF_ATOMIC** memory file flag it is possible to make writes to not split the written record across different blocks, while the functions **xdl_mmfile_iscompact**() , **xdl_mmfile_compact**() and **xdl_mmfile_writeallocate**() are usefull to test if the file is compact and to create a compacted version of the file itself. The text file differential output uses the raw unified output format, by omitting the file header since the result is always relative to a single compare operation (between two files). The output format of the binary patch file is proprietary (and binary) and it is basically a collection of copy and insert commands, like described inside the MacDonald paper.

Functions

The following functions are defined:

int xdl_set_allocator(memallocator_t const * *malt***);**

The **LibXDiff** library enable the user to set its own memory allocator, that will be used for all the following memory requests. The allocator must be set before to start calling the **LibXDiff** library with a call to **xdl_set_allocator**(). The memory allocator structure contains the following members:

```
typedef struct s_memallocator {
        void *priv;
        void *(*malloc)(void *priv, unsigned int size);
        void (*free)(void *priv, void *ptr);
        void *(*realloc)(void *priv, void *ptr, unsigned int nsize);
} memallocator_t;
```

The **malloc**() function pointer will be used by **LibXDiff** to request a memory block of *size* bytes. The **free**() function pointer will be called to free a previously allocated block *ptr* , while the **realloc**() will be used to resize the *ptr* to a new *nsize* size in bytes. The **priv** structure member will be passed to the **malloc**(),**free**(),**realloc**() functions as first parameter. The **LibXDiff** user must call **xdl_set_allocator**() before starting using the library, otherwise **LibXDiff** functions will fail due to the lack of memory allocation support. A typical initialization sequence for **POSIX** systems will use the standard **malloc**(3), **free**(3), **realloc**(3) and will look like:

```
void *wrap_malloc(void *priv, unsigned int size) {
        return malloc(size);
}

void wrap_free(void *priv, void *ptr) {
        free(ptr);
}

void *wrap_realloc(void *priv, void *ptr, unsigned int size) {
        return realloc(ptr, size);
}

void my_init_xdiff(void) {
        memallocator_t malt;

        malt.priv = NULL;
        malt.malloc = wrap_malloc;
        malt.free = wrap_free;
        malt.realloc = wrap_realloc;
        xdl_set_allocator(&malt);
}
```

void *xdl_malloc(unsigned int *size***);**

Allocates a memory block of *size* bytes using the **LibXDiff** memory allocator. The user can specify its own allocator using the **xdl_set_allocator**() function. The **xdl_malloc**() return a pointer to the newly allocated block, or **NULL** in case of failure.

void xdl_free(void * *ptr***);**

Free a previously allocated memory block pointed by *ptr*. The *ptr* block must has been allocated using either **xdl_malloc**() or **xdl_realloc**().

void *xdl_realloc(void * *ptr***, unsigned int** *nsize***);**

Resizes the memory block pointed by *ptr* to a new size *nsize*. Return the resized block if successful, or **NULL** in case the reallocation fails. After a successful reallocation, the old *ptr* block is to be considered no more valid.

int xdl_init_mmfile(mmfile_t * *mmf***, long** *bsize***, unsigned long** *flags***);**

Initialize the memory file *mmf* by requiring an internal block size of *bsize*. The *flags* parameter is a combination of the following flags :

XDL_MMF_ATOMIC Writes on the memory file will be atomic. That is, the data will not be split on two or more different blocks.

Once an **xdl_init_mmfile**() succeeded, a matching **xdl_free_mmfile**() must be called when the user has done using the memory file, otherwise serious memory leaks will happen. The function return 0 if succeed or -1 if an error is encountered.

void xdl_free_mmfile(mmfile_t * *mmf***);**

Free all the data associated with the *mmf* memory file.

int xdl_mmfile_iscompact(mmfile_t * *mmf***);**

Returns an integer different from 0 if the *mmf* memory file is compact, 0 otherwise. A compact memory file is one that have the whole content stored inside a single block.

int xdl_seek_mmfile(mmfile_t * *mmf***, long** *off***);**

Set the current data pointer of the memory file *mmf* to the specified offset *off* from the beginning of the file itself. Returns 0 if successful or -1 if an error happened.

long xdl_read_mmfile(mmfile_t * *mmf***, void *** *data***, long** *size***);**

Request to read *size* bytes from the memory file *mmf* by storing the data inside the *data* buffer. Returns the number of bytes read into the *data* buffer. The amount of data read can be lower than the specified *size*. The function returns -1 if an error happened.

long xdl_write_mmfile(mmfile_t * *mmf***, void const *** *data***, long** *size***);**

Request to write *size* bytes from the specified buffer *data* into the memory file *mmf*. If the memory file has been created using the **XDL_MMF_ATOMIC** flag, the write request will not be split across different blocks. Note that all write operations done on memory files do append data at the end the file, and writes in the middle of it are allowed. This is because the library memory file abstraction does not need this functionality to be available. The function returns the number of bytes written or a number lower than *size* if an error happened.

long xdl_writem_mmfile(mmfile_t *_mmf_, mmbuffer_t *_mb_, int _nbuf_);

Request to sequentially write *nbuf* memory buffers passed inside the array *mb* into the memory file *mmf*. The memory buffer structure is defined as :

```
typedef struct s_mmbuffer {
        char *ptr;
        long size;
} mmbuffer_t;
```

The *ptr* field is a pointer to the user data, whose size is specified inside the *size* structure field. The function returns the total number of bytes written or a lower number if an error happened.

void *xdl_mmfile_writeallocate(mmfile_t *_mmf_, long _size_);

The function request to allocate a write buffer of *size* bytes in the *mmf* memory file and returns the pointer to the allocated buffer. The user will have the responsibility to store *size* bytes (no more, no less) inside the memory region pointed to by the returned pointer. The files size will grow of *size* bytes as a consequence of this operation. The function will return **NULL** if an error happened.

long xdl_mmfile_ptradd(mmfile_t *_mmf_, char *_ptr_, long _size_, unsigned long _flags_);

The function adds a user specified block to the end of the memory file *mmf*. The block first byte is pointed to by *ptr* and its length is *size* bytes. The *flags* parameter can be used to specify attributes of the user memory block. Currently supported attributes are:

XDL_MMB_READONLY Specify that the added memory block must be treated as read-only, and every attempt to write on it should result in a failure of the memory file writing functions.

The purpose of this function is basically to avoid copying memory around, by helping the library to not drain the CPU cache. The function returns *size* in case of success, or -1 in case of error.

void *xdl_mmfile_first(mmfile_t *_mmf_, long *_size_);

The function is used to return the first block of the *mmf* memory file block chain. The *size* parameter will receive the size of the block, while the function will return the pointer the the first byte of the block itself. The function returns **NULL** if the file is empty.

void *xdl_mmfile_next(mmfile_t *_mmf_, long *_size_);

The function is used to return the next block of the *mmf* memory file block chain. The *size* parameter will receive the size of the block, while the function will return the pointer the the first byte of the block itself. The function returns **NULL** if the current block is the last one of the chain.

long xdl_mmfile_size(mmfile_t *_mmf_);

The function returns the size of the specified memory file *mmf*.

int xdl_mmfile_cmp(mmfile_t *_mmf1_, mmfile_t *_mmf2_);

Request to compare two memory files *mmf1* and *mmf2* and returns 0 if files are identical, or a value different from 0 if files are different.

int xdl_mmfile_compact(mmfile_t *_mmfo_, mmfile_t *_mmfc_, long _bsize_, unsigned long _flags_);

Request to create a compact version of the memory file *mmfo* into the (uninitialized) memory file *mmfc*. The *bsize* parameter specify the requested block size and *flags* specify flags to be used to create the new *mmfc* memory file (see **xdl_init_mmfile**()). The function returns 0 if succedded or -1 if an error happened.

int xdl_diff(mmfile_t *_mmf1_, mmfile_t *_mmf2_, xpparam_t const *_xpp_, xdemitconf_t const *_xecfg_, xdemitcb_t *_ecb_);

Request to create the difference between the two text memory files *mmf1* and *mmf2*. The *mmf1* memory files is considered the "old" file while *mmf2* is considered the "new" file. So the function will create a patch file that once applied to *mmf1* will give *mmf2* as result. Files *mmf1* and *mmf2* must be atomic from a line point of view (or, as an extreme, compact), that means that a single test line cannot spread among different memory file blocks. The *xpp* parameter is a pointer to a structure :

```
typedef struct s_xpparam {
        unsigned long flags;
} xpparam_t;
```

that is used to specify parameters to be used by the file differential algorithm. The *flags* field is a combination of the following flags :

XDF_NEED_MINIMAL Requires the minimal edit script to be found by the algorithm (may be slow).

The *xecfg* parameter point to a structure :

```
typedef struct s_xdemitconf {
        long ctxlen;
} xdemitconf_t;
```

that is used to configure the algorithm responsible of the creation the the differential file from an edit script. The *ctxlen* field is used to specify the amount of context to be emitted inside the differential file (the value 3 is suggested for normal operations). The parameter *ecb* is a pointer to a structure :

```
typedef struct s_xdemitcb {
        void *priv;
        int (*outf)(void *, mmbuffer_t *, int);
} xdemitcb_t;
```

that is used by the differential file creation algorithm to emit the created data. The *priv* field is an opaque pointer to a user specified data, while the *outf* field point to a callback function that is called internally to emit algorithm generated data rappresenting the differential file. The first parameter of the callback is the same *priv* field specified inside the **xdemitcb_t** structure. The second parameter point to an array of **mmbuffer_t** (see above for a definition of the structure) whose element count is specified inside the last parameter of the callback itself. The callback will always be called with entire records (lines) and never a record (line) will be emitted using two different callback calls. This is important because if the called will use another memory file to store the result, by creating the target memory file with **XDL_MMF_ATOMIC** will guarantee the "atomicity" of the memory file itself. The function returns 0 if succeeded or -1 if an error occurred.

int xdl_patch(mmfile_t ****mmf*, mmfile_t ****mmfp***, int** *mode***, xdemitcb_t ****ecb***, xdemitcb_t ****rjecb***);**

Request to patch the memory file *mmf* using the patch file stored in *mmfp*. The *mmf* memory file **is not** changed during the operation and can be considered as read only. The *mode* parameter can be one of the following values :

XDL_PATCH_NORMAL Perform standard patching like if the patch memory file *mmfp* has been created using *mmf* as "old" file.

XDL_PATCH_REVERSE Apply the reverse patch. That means that the *mmf* memory file has to be considered as if it was specified as "new" file during the differential operation (**xdl_diff**()). The result of the operation will then be the file content that was used as "old" file during the differential operation.

The following flags can be specified (by or-ing them) to one of the above:

XDL_PATCH_IGNOREBSPACE Ignore the whitespace at the beginning and the end of the line.

The *ecb* will be used by the patch algorithm to create the result file while the *rjecb* will be used to emit all differential chunks that cannot be applied. Like explained above, callbacks are always called with entire records to guarantee atomicity of the resulting output. The function returns 0 if succeeded without performing any fuzzy hunk detection, a positive value if it secceeded with fuzzy hunk detection or -1 if an error occurred during the patch operation.

int xdl_merge3(mmfile_t ****mmfo*, mmfile_t ****mmf1***, mmfile_t ****mmf2***, xdemitcb_t ****ecb***, xdemitcb_t *****rjecb***);**

Merges three files together. The *mmfo* file is the original one, while *mmf1* and *mmf2* are two modified versions of *mmfo*. The function works by creating a differential between *mmfo* and *mmf2* and by applying the resulting patch to *mmf1*. Because of this sequence, *mmf1* changes will be privileged against the ones of *mmf2*. The *ecb* will be used by the patch algorithm to create the result file while the *rjecb* will be used to emit all differential chunks that cannot be applied. Like explained above, callbacks are always called with entire records to guarantee atomicity of the resulting output. The function returns 0 if succeeded or -1 if an error occurred during the patch operation.

int xdl_bdiff(mmfile_t ****mmf1*, mmfile_t ****mmf2***, bdiffparam_t const ****bdp***, xdemitcb_t ****ecb***);**

Request to create the difference between the two text memory files *mmf1* and *mmf2*. The *mmf1* memory files is considered the "old" file while *mmf2* is considered the "new" file. So the function will create a patch file that once applied to *mmf1* will give *mmf2* as result. Files *mmf1* and *mmf2* must be compact to make it easy and faster to perform the difference operation. Functions are available to check for compactness (**xdl_mmfile_iscompact**()) and to make compact a non-compact file (**xdl_mmfile_compact**()). An example of how to create a compact memory file (described inside the test subdirectory) is :

```
int xdlt_load_mmfile(char const *fname, mmfile_t *mf, int binmode) {
        char cc;
        int fd;
        long size, bsize;
        char *blk;

        if (xdl_init_mmfile(mf, XDLT_STD_BLKSIZE, XDL_MMF_ATOMIC) < 0)
                return -1;
        if ((fd = open(fname, O_RDONLY)) == -1) {
```

```
                perror(fname);
                xdl_free_mmfile(mf);
                return -1;
        }
        if ((size = bsize = lseek(fd, 0, SEEK_END)) > 0 && !binmode) {
                if (lseek(fd, -1, SEEK_END) != (off_t) -1 &&
                   read(fd, &cc, 1) && cc != '\n')
                        bsize++;
        }
        lseek(fd, 0, SEEK_SET);
        if (!(blk = (char *) xdl_mmfile_writeallocate(mf, bsize))) {
                xdl_free_mmfile(mf);
                close(fd);
                return -1;
        }
        if (read(fd, blk, (size_t) size) != (size_t) size) {
                perror(fname);
                xdl_free_mmfile(mf);
                close(fd);
                return -1;
        }
        close(fd);
        if (bsize > size)
                blk[size] = '\n';
        return 0;
}
```

The *bdp* parameter points to a structure :

```
typedef struct s_bdiffparam {
        long bsize;
} bdiffparam_t;
```

that is used to pass information to the binary file differential algorithm. The *bsize* parameter specify the size of the block that will be used to decompose *mmf1* during the block classification phase of the algorithm (see MacDonald paper). Suggested values go from 16 to 64, with a preferred power of two characteristic. The *ecb* parameter is used to pass the emission callback to the algorithm responsible of the output file creation. The function returns 0 if succeede or -1 if an error is occurred.

int xdl_bdiff_mb(mmbuffer_t **mmb1*, mmbuffer_t **mmb2*, bdiffparam_t const **bdp*, xdemitcb_t **ecb*);

Same as **xdl_bdiff**() but it works on memory buffer directly. The **xdl_bdiff**() is implemented internally with a **xdl_bdiff_mb**() after having setup the two memory buffers from the passed memory files (that must be compact, as described above). The memory buffer structure is defined as :

```
typedef struct s_mmbuffer {
        char *ptr;
        long size;
} mmbuffer_t;
```

An empty memory buffer is specified by setting the *ptr* member as **NULL** and the *size* member as zero. The reason of having this function is to avoid the memory file preparation, that might involve

copying memory from other sources. Using the **xdl_bdiff**(), the caller can setup the two memory buffer by using, for example, **mmap**(2), and hence avoiding unnecessary memory copies. The other parameters and the return value of the function **xdl_bdiff_mb**() are the same as the ones already described in **xdl_bdiff**().

long xdl_bdiff_tgsize(mmfile_t *_mmfp_);

Given a binary memory file patch, it returns the size that the result file will have once the patch is applied to the target file. It can be used to pre-allocate (or write-allocate) a memory block to store the patch result so that a compact file will be available at the end of the operation. The function returns the requested size, or -1 if an error occurred during the operation.

int xdl_bpatch(mmfile_t *_mmf_, mmfile_t *_mmfp_, xdemitcb_t *_ecb_);

Request to patch the binary memory file *mmf* using the binary patch file stored in *mmfp*. The *mmf* memory file **is not** changed during the operation and can be considered as read only. The binary patch algorithm has no notion of context, so the patch operation cannot be partial (either success or failure). The *ecb* parameter contain the callabck (see above for description) used by the binary patch algorithm to emit the result file. The function returns 0 if succeeded or -1 if an error occurred during the patch operation.

SEE ALSO

Two papers drove the content of this library and these are :

- *File System Support for Delta Compression* by *Joshua P. MacDonald*
- *An O(ND) Difference Algorithm and Its Variations* by *Eugene W. Myers*.

Also usefull information can be looked up inside the **diffutil** GNU package :

http://www.gnu.org/software/diffutils/diffutils.html

LICENSE

AUTHOR

Developed by Davide Libenzi <**davidel@xmailserver.org**>

AVAILABILITY

The latest version of **LibXDiff** can be found at :

http://www.xmailserver.org/xdiff-lib.html

BUGS

There are no known bugs. Bug reports and comments to Davide Libenzi <**davidel@xmailserver.org**>