NAME

archive_write_disk_new, **archive_write_disk_set_options**, **archive_write_disk_set_skip_file**, **archive_write_disk_set_group_lookup**, **archive_write_disk_set_standard_lookup**, **archive_write_disk_set_user_lookup**, **archive_write_header**, **archive_write_data**, **archive_write_data_block**, **archive_write_finish_entry**, **archive_write_close**, **archive_write_finish** **archive_write_free** — functions for creating objects on disk

LIBRARY

Streaming Archive Library (libarchive, -larchive)

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_write_disk_new(void);

int
archive_write_disk_set_options(struct archive *, int flags);

int
archive_write_disk_set_skip_file(struct archive *, dev_t, ino_t);

int
archive_write_disk_set_group_lookup(struct archive *, void *,
    gid_t (*)(void *, const char *gname, gid_t gid),
    void (*cleanup)(void *));

int
archive_write_disk_set_standard_lookup(struct archive *);

int
archive_write_disk_set_user_lookup(struct archive *, void *,
    uid_t (*)(void *, const char *uname, uid_t uid),
    void (*cleanup)(void *));

int
archive_write_header(struct archive *, struct archive_entry *);

la_ssize_t
archive_write_data(struct archive *, const void *, size_t);

la_ssize_t
archive_write_data_block(struct archive *, const void *, size_t size,
    int64_t offset);

int
archive_write_finish_entry(struct archive *);

int
archive_write_close(struct archive *);

int
archive_write_finish(struct archive *);

int
archive_write_free(struct archive *);
```

DESCRIPTION

These functions provide a complete API for creating objects on disk from struct archive_entry descriptions. They are most naturally used when extracting objects from an archive using the **archive_read**() interface. The general process is to read struct archive_entry objects from an archive, then write those objects to a struct archive object created using the **archive_write_disk**() family functions. This interface is deliberately very similar to the **archive_write**() interface used to write objects to a streaming archive.

archive_write_disk_new()
: Allocates and initializes a struct archive object suitable for writing objects to disk.

archive_write_disk_set_skip_file()
: Records the device and inode numbers of a file that should not be overwritten. This is typically used to ensure that an extraction process does not overwrite the archive from which objects are being read. This capability is technically unnecessary but can be a significant performance optimization in practice.

archive_write_disk_set_options()
: The options field consists of a bitwise OR of one or more of the following values:

ARCHIVE_EXTRACT_OWNER
: The user and group IDs should be set on the restored file. By default, the user and group IDs are not restored.

ARCHIVE_EXTRACT_PERM
: Full permissions (including SGID, SUID, and sticky bits) should be restored exactly as specified, without obeying the current umask. Note that SUID and SGID bits can only be restored if the user and group ID of the object on disk are correct. If **ARCHIVE_EXTRACT_OWNER** is not specified, then SUID and SGID bits will only be restored if the default user and group IDs of newly-created objects on disk happen to match those specified in the archive entry. By default, only basic permissions are restored, and umask is obeyed.

ARCHIVE_EXTRACT_TIME
: The timestamps (mtime, ctime, and atime) should be restored. By default, they are ignored. Note that restoring of atime is not currently supported.

ARCHIVE_EXTRACT_NO_OVERWRITE
: Existing files on disk will not be overwritten. By default, existing regular files are truncated and overwritten; existing directories will have their permissions updated; other pre-existing objects are unlinked and recreated from scratch.

ARCHIVE_EXTRACT_UNLINK
: Existing files on disk will be unlinked before any attempt to create them. In some cases, this can prove to be a significant performance improvement. By default, existing files are truncated and rewritten, but the file is not recreated. In particular, the default behavior does not break existing hard links.

ARCHIVE_EXTRACT_ACL
: Attempt to restore ACLs. By default, extended ACLs are ignored.

ARCHIVE_EXTRACT_FFLAGS
: Attempt to restore extended file flags. By default, file flags are ignored.

ARCHIVE_EXTRACT_XATTR
: Attempt to restore POSIX.1e extended attributes. By default, they are ignored.

ARCHIVE_EXTRACT_SECURE_SYMLINKS
: Refuse to extract any object whose final location would be altered by a symlink on disk. This is intended to help guard against a variety of mischief caused by archives that (deliberately or otherwise) extract files outside of the current directory. The default is not to perform this check. If **ARCHIVE_EXTRACT_UNLINK** is specified together with this option, the library will remove any intermediate symlinks it finds and return an error

only if such symlink could not be removed.

ARCHIVE_EXTRACT_SECURE_NODOTDOT
Refuse to extract a path that contains a `..` element anywhere within it. The default is to not refuse such paths. Note that paths ending in `..` always cause an error, regardless of this flag.

ARCHIVE_EXTRACT_SECURE_NOABSOLUTEPATHS
Refuse to extract an absolute path. The default is to not refuse such paths.

ARCHIVE_EXTRACT_SPARSE
Scan data for blocks of NUL bytes and try to recreate them with holes. This results in sparse files, independent of whether the archive format supports or uses them.

ARCHIVE_EXTRACT_CLEAR_NOCHANGE_FFLAGS
Before removing a file system object prior to replacing it, clear platform-specific file flags which might prevent its removal.

archive_write_disk_set_group_lookup(), **archive_write_disk_set_user_lookup**()
The struct archive_entry objects contain both names and ids that can be used to identify users and groups. These names and ids describe the ownership of the file itself and also appear in ACL lists. By default, the library uses the ids and ignores the names, but this can be overridden by registering user and group lookup functions. To register, you must provide a lookup function which accepts both a name and id and returns a suitable id. You may also provide a void * pointer to a private data structure and a cleanup function for that data. The cleanup function will be invoked when the struct archive object is destroyed.

archive_write_disk_set_standard_lookup()
This convenience function installs a standard set of user and group lookup functions. These functions use `getpwnam`(3) and `getgrnam`(3) to convert names to ids, defaulting to the ids if the names cannot be looked up. These functions also implement a simple memory cache to reduce the number of calls to `getpwnam`(3) and `getgrnam`(3).

archive_write_header()
Build and write a header using the data in the provided struct archive_entry structure. See `archive_entry`(3) for information on creating and populating struct archive_entry objects.

archive_write_data()
Write data corresponding to the header just written. Returns number of bytes written or -1 on error.

archive_write_data_block()
Write data corresponding to the header just written. This is like **archive_write_data**() except that it performs a seek on the file being written to the specified offset before writing the data. This is useful when restoring sparse files from archive formats that support sparse files. Returns number of bytes written or -1 on error. (Note: This is currently not supported for archive_write handles, only for archive_write_disk handles.)

archive_write_finish_entry()
Close out the entry just written. Ordinarily, clients never need to call this, as it is called automatically by **archive_write_next_header**() and **archive_write_close**() as needed. However, some file attributes are written to disk only after the file is closed, so this can be necessary if you need to work with the file on disk right away.

archive_write_close()
Set any attributes that could not be set during the initial restore. For example, directory timestamps are not restored initially because restoring a subsequent file would alter that timestamp. Similarly, non-writable directories are initially created with write permissions (so that their contents can be restored). The **archive_write_disk_new** library maintains a list of all such

deferred attributes and sets them when this function is invoked.

archive_write_finish()
This is a deprecated synonym for **archive_write_free**().

archive_write_free()
Invokes **archive_write_close**() if it was not invoked manually, then releases all resources.

More information about the *struct archive* object and the overall design of the library can be found in the libarchive(3) overview. Many of these functions are also documented under archive_write(3).

RETURN VALUES

Most functions return **ARCHIVE_OK** (zero) on success, or one of several non-zero error codes for errors. Specific error codes include: **ARCHIVE_RETRY** for operations that might succeed if retried, **ARCHIVE_WARN** for unusual conditions that do not prevent further operations, and **ARCHIVE_FATAL** for serious errors that make remaining operations impossible.

archive_write_disk_new() returns a pointer to a newly-allocated struct archive object.

archive_write_data() returns a count of the number of bytes actually written, or -1 on error.

ERRORS

Detailed error codes and textual descriptions are available from the **archive_errno**() and **archive_error_string**() functions.

SEE ALSO

archive_read(3), archive_write(3), tar(1), libarchive(3)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3. The **archive_write_disk** interface was added to **libarchive 2.0** and first appeared in FreeBSD 6.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle <kientzle@acm.org>.

BUGS

Directories are actually extracted in two distinct phases. Directories are created during **archive_write_header**(), but final permissions are not set until **archive_write_close**(). This separation is necessary to correctly handle borderline cases such as a non-writable directory containing files, but can cause unexpected results. In particular, directory permissions are not fully restored until the archive is closed. If you use chdir(2) to change the current directory between calls to **archive_read_extract**() or before calling **archive_read_close**(), you may confuse the permission-setting logic with the result that directory permissions are restored incorrectly.

The library attempts to create objects with filenames longer than **PATH_MAX** by creating prefixes of the full path and changing the current directory. Currently, this logic is limited in scope; the fixup pass does not work correctly for such objects and the symlink security check option disables the support for very long pathnames.

Restoring the path aa/../bb does create each intermediate directory. In particular, the directory aa is created as well as the final object bb. In theory, this can be exploited to create an entire directory hierarchy with a single request. Of course, this does not work if the **ARCHIVE_EXTRACT_NODOTDOT** option is specified.

Implicit directories are always created obeying the current umask. Explicit objects are created obeying the current umask unless **ARCHIVE_EXTRACT_PERM** is specified, in which case they current umask is ignored.

SGID and SUID bits are restored only if the correct user and group could be set. If **ARCHIVE_EXTRACT_OWNER** is not specified, then no attempt is made to set the ownership. In this case, SGID and SUID bits are restored only if the user and group of the final object happen to match those specified in the entry.

The "standard" user-id and group-id lookup functions are not the defaults because `getgrnam`(3) and `getpwnam`(3) are sometimes too large for particular applications. The current design allows the application author to use a more compact implementation when appropriate.

There should be a corresponding **archive_read_disk** interface that walks a directory hierarchy and returns archive entry objects.