NAME

libarchive_internals — description of libarchive internal interfaces

OVERVIEW

The **libarchive** library provides a flexible interface for reading and writing streaming archive files such as tar and cpio. Internally, it follows a modular layered design that should make it easy to add new archive and compression formats.

GENERAL ARCHITECTURE

Externally, libarchive exposes most operations through an opaque, object-style interface. The archive_entry(3) objects store information about a single filesystem object. The rest of the library provides facilities to write archive_entry(3) objects to archive files, read them from archive files, and write them to disk. (There are plans to add a facility to read archive_entry(3) objects from disk as well.)

The read and write APIs each have four layers: a public API layer, a format layer that understands the archive file format, a compression layer, and an I/O layer. The I/O layer is completely exposed to clients who can replace it entirely with their own functions.

In order to provide as much consistency as possible for clients, some public functions are virtualized. Eventually, it should be possible for clients to open an archive or disk writer, and then use a single set of code to select and write entries, regardless of the target.

READ ARCHITECTURE

From the outside, clients use the archive_read(3) API to manipulate an **archive** object to read entries and bodies from an archive stream. Internally, the **archive** object is cast to an **archive_read** object, which holds all read-specific data. The API has four layers: The lowest layer is the I/O layer. This layer can be overridden by clients, but most clients use the packaged I/O callbacks provided, for example, by archive_read_open_memory(3), and archive_read_open_fd(3). The compression layer calls the I/O layer to read bytes and decompresses them for the format layer. The format layer unpacks a stream of uncompressed bytes and creates **archive_entry** objects from the incoming data. The API layer tracks overall state (for example, it prevents clients from reading data before reading a header) and invokes the format and compression layer operations through registered function pointers. In particular, the API layer drives the format-detection process: When opening the archive, it reads an initial block of data and offers it to each registered compression handler. The one with the highest bid is initialized with the first block. Similarly, the format handlers are polled to see which handler is the best for each archive. (Prior to 2.4.0, the format bidders were invoked for each entry, but this design hindered error recovery.)

I/O Layer and Client Callbacks

The read API goes to some lengths to be nice to clients. As a result, there are few restrictions on the behavior of the client callbacks.

The client read callback is expected to provide a block of data on each call. A zero-length return does indicate end of file, but otherwise blocks may be as small as one byte or as large as the entire file. In particular, blocks may be of different sizes.

The client skip callback returns the number of bytes actually skipped, which may be much smaller than the skip requested. The only requirement is that the skip not be larger. In particular, clients are allowed to return zero for any skip that they don't want to handle. The skip callback must never be invoked with a negative value.

Keep in mind that not all clients are reading from disk: clients reading from networks may provide different-sized blocks on every request and cannot skip at all; advanced clients may use mmap(2) to read the entire file into memory at once and return the entire file to libarchive as a single block; other clients may begin asynchronous I/O operations for the next block on each request.

Decompression Layer

The decompression layer not only handles decompression, it also buffers data so that the format handlers see a much nicer I/O model. The decompression API is a two stage peek/consume model. A read_ahead request specifies a minimum read amount; the decompression layer must provide a pointer to at least that much data. If more data is immediately available, it should return more: the format layer handles bulk data reads by asking for a minimum of one byte and then copying as much data as is available.

A subsequent call to the **consume**() function advances the read pointer. Note that data returned from a **read_ahead**() call is guaranteed to remain in place until the next call to **read_ahead**(). Intervening calls to **consume**() should not cause the data to move.

Skip requests must always be handled exactly. Decompression handlers that cannot seek forward should not register a skip handler; the API layer fills in a generic skip handler that reads and discards data.

A decompression handler has a specific lifecycle:

Registration/Configuration

When the client invokes the public support function, the decompression handler invokes the internal **__archive_read_register_compression**() function to provide bid and initialization functions. This function returns **NULL** on error or else a pointer to a **struct decompressor_t**. This structure contains a *void * config* slot that can be used for storing any customization information.

Bid The bid function is invoked with a pointer and size of a block of data. The decompressor can access its config data through the *decompressor* element of the **archive_read** object. The bid function is otherwise stateless. In particular, it must not perform any I/O operations.

The value returned by the bid function indicates its suitability for handling this data stream. A bid of zero will ensure that this decompressor is never invoked. Return zero if magic number checks fail. Otherwise, your initial implementation should return the number of bits actually checked. For example, if you verify two full bytes and three bits of another byte, bid 19. Note that the initial block may be very short; be careful to only inspect the data you are given. (The current decompressors require two bytes for correct bidding.)

Initialize The winning bidder will have its init function called. This function should initialize the remaining slots of the *struct decompressor_t* object pointed to by the *decompressor* element of the *archive_read* object. In particular, it should allocate any working data it needs in the *data* slot of that structure. The init function is called with the block of data that was used for tasting. At this point, the decompressor is responsible for all I/O requests to the client callbacks. The decompressor is free to read more data as and when necessary.

Satisfy I/O requests

The format handler will invoke the *read_ahead*, *consume*, and *skip* functions as needed.

Finish The finish method is called only once when the archive is closed. It should release anything stored in the *data* and *config* slots of the *decompressor* object. It should not invoke the client close callback.

Format Layer

The read formats have a similar lifecycle to the decompression handlers:

Registration

Allocate your private data and initialize your pointers.

Bid Formats bid by invoking the **read_ahead**() decompression method but not calling the **consume**() method. This allows each bidder to look ahead in the input stream. Bidders should not look further ahead than necessary, as long look aheads put pressure on the decompression layer to buffer lots of data. Most formats only require a few hundred bytes of look ahead; look aheads of a few kilobytes are reasonable. (The ISO9660 reader sometimes looks ahead by 48k, which should be considered an upper limit.)

Read header
: The header read is usually the most complex part of any format. There are a few strategies worth mentioning: For formats such as tar or cpio, reading and parsing the header is straightforward since headers alternate with data. For formats that store all header data at the beginning of the file, the first header read request may have to read all headers into memory and store that data, sorted by the location of the file data. Subsequent header read requests will skip forward to the beginning of the file data and return the corresponding header.

Read Data
: The read data interface supports sparse files; this requires that each call return a block of data specifying the file offset and size. This may require you to carefully track the location so that you can return accurate file offsets for each read. Remember that the decompressor will return as much data as it has. Generally, you will want to request one byte, examine the return value to see how much data is available, and possibly trim that to the amount you can use. You should invoke consume for each block just before you return it.

Skip All Data
: The skip data call should skip over all file data and trailing padding. This is called automatically by the API layer just before each header read. It is also called in response to the client calling the public **data_skip**() function.

Cleanup
: On cleanup, the format should release all of its allocated memory.

API Layer

XXX to do XXX

WRITE ARCHITECTURE

The write API has a similar set of four layers: an API layer, a format layer, a compression layer, and an I/O layer. The registration here is much simpler because only one format and one compression can be registered at a time.

I/O Layer and Client Callbacks

XXX To be written XXX

Compression Layer

XXX To be written XXX

Format Layer

XXX To be written XXX

API Layer

XXX To be written XXX

WRITE_DISK ARCHITECTURE

The write_disk API is intended to look just like the write API to clients. Since it does not handle multiple formats or compression, it is not layered internally.

GENERAL SERVICES

The **archive_read**, **archive_write**, and **archive_write_disk** objects all contain an initial **archive** object which provides common support for a set of standard services. (Recall that ANSI/ISO C90 guarantees that you can cast freely between a pointer to a structure and a pointer to the first element of that structure.) The **archive** object has a magic value that indicates which API this object is associated with, slots for storing error information, and function pointers for virtualized API functions.

MISCELLANEOUS NOTES

Connecting existing archiving libraries into libarchive is generally quite difficult. In particular, many existing libraries strongly assume that you are reading from a file; they seek forwards and backwards as necessary to locate various pieces of information. In contrast, libarchive never seeks backwards in its input, which sometimes requires very different approaches.

For example, libarchive's ISO9660 support operates very differently from most ISO9660 readers. The libarchive support utilizes a work-queue design that keeps a list of known entries sorted by their location in the input. Whenever libarchive's ISO9660 implementation is asked for the next header, checks this list to find the next item on the disk. Directories are parsed when they are encountered and new items are added to the list. This design relies heavily on the ISO9660 image being optimized so that directories always occur earlier on the disk than the files they describe.

Depending on the specific format, such approaches may not be possible. The ZIP format specification, for example, allows archivers to store key information only at the end of the file. In theory, it is possible to create ZIP archives that cannot be read without seeking. Fortunately, such archives are very rare, and libarchive can read most ZIP archives, though it cannot always extract as much information as a dedicated ZIP program.

SEE ALSO

`archive_entry`(3), `archive_read`(3), `archive_write`(3), `archive_write_disk`(3), `libarchive`(3)

HISTORY

The **`libarchive`** library first appeared in FreeBSD 5.3.

AUTHORS

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