NAME

archive_write — functions for creating archives

LIBRARY

Streaming Archive Library (libarchive, -larchive)

SYNOPSIS

```
#include <archive.h>
```

DESCRIPTION

These functions provide a complete API for creating streaming archive files. The general process is to first create the struct archive object, set any desired options, initialize the archive, append entries, then close the archive and release all resources.

Create archive object

See archive_write_new(3).

To write an archive, you must first obtain an initialized struct archive object from **archive_write_new**().

Enable filters and formats, configure block size and padding

See archive_write_filter(3), archive_write_format(3) and archive_write_blocksize(3).

You can then modify this object for the desired operations with the various **archive_write_set_XXX**() functions. In particular, you will need to invoke appropriate **archive_write_add_XXX**() and **archive_write_set_XXX**() functions to enable the corresponding compression and format support.

Set options

See archive_write_set_options(3).

Open archive

See archive_write_open(3).

Once you have prepared the struct archive object, you call **archive_write_open**() to actually open the archive and prepare it for writing. There are several variants of this function; the most basic expects you to provide pointers to several functions that can provide blocks of bytes from the archive. There are convenience forms that allow you to specify a filename, file descriptor, *FILE* * object, or a block of memory from which to write the archive data.

Produce archive

See archive_write_header(3) and archive_write_data(3).

Individual archive entries are written in a three-step process: You first initialize a struct archive_entry structure with information about the new entry. At a minimum, you should set the pathname of the entry and provide a *struct stat* with a valid *st_mode* field, which specifies the type of object and *st_size* field, which specifies the size of the data portion of the object.

Release resources

See archive_write_free(3).

After all entries have been written, use the **archive_write_free**() function to release all resources.

EXAMPLE

The following sketch illustrates basic usage of the library. In this example, the callback functions are simply wrappers around the standard open(2), write(2), and close(2) system calls.

```
#ifdef __linux__
#define _FILE_OFFSET_BITS 64
#endif
#include <sys/stat.h>
#include <archive.h>
#include <archive_entry.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

struct mydata {
  const char *name;
  int fd;
};

int
myopen(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  mydata->fd = open(mydata->name, O_WRONLY | O_CREAT, 0644);
  if (mydata->fd >= 0)
    return (ARCHIVE_OK);
  else
    return (ARCHIVE_FATAL);
}

la_ssize_t
mywrite(struct archive *a, void *client_data, const void *buff, size_t n)
{
  struct mydata *mydata = client_data;

  return (write(mydata->fd, buff, n));
}

int
myclose(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  if (mydata->fd > 0)
    close(mydata->fd);
  return (0);
}

void
write_archive(const char *outname, const char **filename)
{
```

```
  struct mydata *mydata = malloc(sizeof(struct mydata));
  struct archive *a;
  struct archive_entry *entry;
  struct stat st;
  char buff[8192];
  int len;
  int fd;

  a = archive_write_new();
  mydata->name = outname;
  /* Set archive format and filter according to output file extension.
   * If it fails, set default format. Platform depended function.
   * See supported formats in archive_write_set_format_filter_by_ext.c */
  if (archive_write_set_format_filter_by_ext(a, outname) != ARCHIVE_OK)  {
    archive_write_add_filter_gzip(a);
    archive_write_set_format_ustar(a);
  }
  archive_write_open(a, mydata, myopen, mywrite, myclose);
  while (*filename) {
    stat(*filename, &st);
    entry = archive_entry_new();
    archive_entry_copy_stat(entry, &st);
    archive_entry_set_pathname(entry, *filename);
    archive_write_header(a, entry);
    if ((fd = open(*filename, O_RDONLY)) != -1) {
      len = read(fd, buff, sizeof(buff));
      while (len > 0) {
        archive_write_data(a, buff, len);
        len = read(fd, buff, sizeof(buff));
      }
      close(fd);
    }
    archive_entry_free(entry);
    filename++;
  }
  archive_write_free(a);
}

int main(int argc, const char **argv)
{
  const char *outname;
  argv++;
  outname = *argv++;
  write_archive(outname, argv);
  return 0;
}
```

SEE ALSO

tar(1), libarchive(3), archive_write_set_options(3), cpio(5), mtree(5), tar(5)

HISTORY

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

AUTHORS

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

BUGS

There are many peculiar bugs in historic tar implementations that may cause certain programs to reject archives written by this library. For example, several historic implementations calculated header checksums incorrectly and will thus reject valid archives; GNU tar does not fully support pax interchange format; some old tar implementations required specific field terminations.

The default pax interchange format eliminates most of the historic tar limitations and provides a generic key/value attribute facility for vendor-defined extensions. One oversight in POSIX is the failure to provide a standard attribute for large device numbers. This library uses "SCHILY.devminor" and "SCHILY.devmajor" for device numbers that exceed the range supported by the backwards-compatible ustar header. These keys are compatible with Joerg Schilling's **star** archiver. Other implementations may not recognize these keys and will thus be unable to correctly restore device nodes with large device numbers from archives created by this library.