extfs

Release 1.0.0

Feb 17, 2017

Contents

extfs is a simple, pure C++ implementation of the ext* family of file systems. It provides a simple API to inspect, traverse and modify ext2/3/4 file systems. **extfs** is designed to be included directly in the build process of other projects, that why no binary builds are provided.

CHAPTER 1

User documentation

The API

File System Access

struct `fs::extfs`
The ext* file system.

This abstraction provides a top-level interface to an ext* family file system. It grants access to information like the size of the file system, the space left on the file system as well as the label.

Since 1.0

Public Types

enum `mode`
How to open the file system.

Since 1.0

Values:

`read_only`
Open in read-only mode.

`writeable`
Open in read-write mode.

Public Functions

`extfs`(std::string **const** &*path*, mode **const** *openMode* = mode::read_only)
Open the filesystem at a given path.

Since 1.0

Note This call will also succeed if the file system could not be opened for some reason. To check whether the file system was successfully opened, see *fs::extfs::open()*.

Parameters

- `path`: The path to a device/file containing an ext* file system.
- `openMode`: Whether to open the file system in read_only or writeable mode.

bool **`open`**`()` **const**
Check if the filesystem is open.

Return `true`, iff. the file system is opened, `false` otherwise

Since 1.0

std::string **`label`**`()` **const**
Get the label of the file system.

The ext2/3/4 file systems allow the use of a label for human readable identification of a file system. Because they are user configurable, there are no guarantees on whether or not the label is unique. Thus, a file system should never be identified solely by its label.

Return A std::string containing the file system label. The string might be empty if no label is set.

Since 1.0

bool **`has_label`**`()` **const**
Check if the file system has a label.

ext2/3/4 file systems may or may not have a label. If a label is present, it has a maximum length of 15 character in ISO-Latin-1 encoding.

Return `true`, iff. the file system has a `non-null` label configured, `false` otherwise

Since 1.0

Internal Design

The Superblock

The **superblock** of an ext2/3/4 file system describes the structure and configuration of the file system. This information is used by the implementation to determine the physical and logical structure of the file system. This section describes the structure of the superblock itself.

Definitions

All fields described in this section are stored on the disk in little-endian format, regardless of the system architecture.

> **Warning:** The implementation currently only works in little-endian architectures. If you would like to get involved in implementing big-endian support, please file an issue at the project repository over at Github

The code in *Implementation* makes us of several `using` directives to reduce the amount of typing as well as make the code more readable. The following aliases are declared in `fs/detail/types.hpp`:

using `u32` = std::uint32_t
An unsigned 32-bit integer.

using `s32` = std::int32_t
A signed 32-bit integer.

using `u16` = std::uint16_t
An unsigned 16-bit integer.

using `s16` = std::uint16_t
A signed 16-bit integer.

using `u08` = std::uint8_t
An unsigned 8-bit integer.

using `u32_arr` = std::array<*u32*, Size>
An array of unsigned 32-bit integers

Template Parameters

- `Size`: The size of the array

using `u08_arr` = std::array<*u08*, Size>
An array of unsigned 8-bit integers

Template Parameters

- `Size`: The size of the array

using `chr_arr` = std::array<char, Size>
An array of characters

Template Parameters

- `Size`: The size of the array

Structure

Todo

Describe structure of the superblock

Implementation

struct `fs::detail::superblock`
This structure describes the ext2/3/4 superblock

Since 1.0

Public Types

enum `creator_operating_system`
The operating system that created the file system.

The "standard" utilities to create an ext2/3/4 file system record the operating system they were used on. The values of this enumeration are the "well-known" operating systems, e.g the ones most implementations should understand.

Since 1.0

Values:

linux = 0
Linux.

hurd = 1
HURD.

masix = 2
MASIX.

freebsd = 3
FreeBSD.

lites = 4
Lites.

enum revision_level
The revision level of the file system.

ext2/3/4 currently come in two different revision levels, known as the "Good old" revision and the "Dynamic" revision. The "Good old" format uses fixed inode size and generally lacks some "modern" features, whereas the "Dynamic" format supports, among other things, dynamic inode sizes.

Since 1.0

Values:

good_old = 0
The first version of ext2.

dynamic = 1
The file system supports "modern" features.

enum compatible_feature
The compatible features of ext2/3/4.

ext2/3/4 define a set of so-called compatible features. Even if the implementation does not support these features, it is safe to read and write data from and to the file system. The values of this enumeration reflect the currently "well-known" features.

Note The current implementation does not support any of the "compatible features".

Since 1.0

Values:

directory_preallocation = 1
Blocks for new directories can be preallocated.

imagic_inodes = 2
TODO: Find out what this does.

has_journal = 4
The file system has an ext3 journal.

extended_attribues = 8
The file system supports extended attributes.

resize_inode = 16
The file system can be resized.

directory_indexing = 32
The file system suppors directory indexing.

lazy_block_group_initialization = 64
The file system lazily initializes block groups.

exclude_inode = 128
TODO: Find out what that does.

exclude_bitmaps = 256
The file system has snapshot-related exclude bitmaps.

sparse_superblock_v2 = 512
The file system uses version 2 of the sparse superblock.

enum incompatible_feature
The incompatible features of ext2/3/4.

ext2/3/4 define a set of so-called incompatible features. If the file system makes use of one or more of these features and the implementation does not support the features used, it must refuse to read or write from or to the file system. The values of this enumeration are the currently "well-known" features.

Note The current implementation implementation does not support any of the "incompatible features".

Since 1.0

Values:

compression = 1
The file system uses compression.

filetype = 2
Filetypes are recorded in directory entries.

recover = 4
The fFile system needs recovery.

journal_device = 8
The file system has a separate device for the journal.

meta_block_group = 16
The file system has meta block groups.

extents = 64
The file system uses extents.

large_file_system = 128
The file system supports 2^64 blocks.

multiple_mount_protection = 256
The file system must be protected against being mounted more than once at a time.

flexible_block_groups = 512
The file system uses flexible block groups.

large_extended_attribues_in_inodes = 1024
The file system stores large extended attributes in inodes.

data_in_directories = 4096
The file system stores data directly in directory entries.

metadata_checksum_seed_in_superblock = 8192
The checksum seed for metadata is stored in the superblock.

large_directory = 16384
The file system uses a large directory or 3-level hash tree.

data_in_inode = 32768
The file system stores data directly inside inodes.

encrypted_inodes = 65536
The file system uses encrypted inodes.

enum read_only_compatible_feature
The read-only compatible features of ext2/3/4.

ext2/3/4 define a set of so-called read-only compatible features. An implementation that does not support one or more of these features might still access the file system in a read-only way. The values of this enumeration are the currently "well-known" read-only compatible features.

Note The current implementation implementation does not support any of the "read-only compatible features".

Since 1.0

Values:

sparse_superblock = 1
The file system has a sparse superblock.

large_file = 2
The file system supports large files.

binary_tree_directories = 4
The file system uses sorted binary trees for directories.

huge_file = 8
The file system contains files represented by the number of logical blocks (e.g. HUGE files)

enum compression_algorithm
The compression algorithms of ext2/3/4.

While compression in ext2 was only supported via a patch, later iterations added the compression feature as a "core" component of the file system. The values of this enumeration are the currently "well-known" compression algorithms found in ext2/3/4.

Note The current implementation implementation does not support any of these algorithms.

Note A file system might be using multiple compression algorithms at a time.

Since 1.0

Values:

lempel_ziv = 1
Lempel-Ziv compression.

lempel_ziv_ross_williams_3a = 2
Lempel-Ziv Ross-Williams 3A compression.

gzip = 4
GZIP compression.

bzip2 = 8
BZIP2 compression.

lempel_ziv_oberhumer = 16
Lempel-Ziv-Oberhumer compression.

using cos = std::underlying_type_t<creator_operating_system>
The underlying type of *creator_operating_system*.

Since 1.0

using rlv = std::underlying_type_t<*revision_level*>
The underlying type of *revision_level*.

Since 1.0

using cft = std::underlying_type_t<compatible_feature>
The underlying type of compatible_feature.

Since 1.0

using ift = std::underlying_type_t<incompatible_feature>
The underlying type of incompatible_feature.

Since 1.0

using rft = std::underlying_type_t<read_only_compatible_feature>
The underlying type of read_only_compatible_feature.

Since 1.0

using cpr = std::underlying_type_t<compression_algorithm>
The underlying type of compression_algorithms.

Since 1.0

Public Functions

bool **has**(compatible_feature **const** *feature*) **const**
Check if the file system has the desired "compatible feature".

Return `true` iff. the file system has the feature, `false` otherwise

See *compatible_feature*

Since 1.0

Parameters

- `feature`: The *compatible_feature* to check for

bool **has_all**(std::initializer_list<compatible_feature> **const** *features*) **const**
Check if the file system has all of the desired "compatible features".

Parameters

- `features`: The

  iff. the file system has

  features that were querried,

  otherwise #compatible_feature 1.0

bool **has_any**(std::initializer_list<compatible_feature> **const** *features*) **const**
Check if the file system has at least one of the desired "compatible features".

Parameters

- `features`: The

  iff. the file system has

  least one of the features that were querried,

  otherwise #compatible_feature 1.0

bool **has**(incompatible_feature **const** *feature*) **const**
Check if the file system has the desired "incompatible feature".

Return `true` iff. the file system has the feature

See *incompatible_feature*

Since 1.0

Parameters

- `feature`: The *incompatible_feature* to check for.

bool **has_all**(std::initializer_list<incompatible_feature> **const** *features*) **const**
Check if the file system has all of the desired "incompatible features".

Parameters

- `features`: The

  iff. the file system has

  features that were querried,

  otherwise #compatible_feature 1.0

bool **has_any**(std::initializer_list<incompatible_feature> **const** *features*) **const**
Check if the file system has at least one of the desired "incompatible features".

Parameters

- `features`: The

  iff. the file system has

  least one of the features that were querried,

  otherwise #compatible_feature 1.0

bool **has**(read_only_compatible_feature **const** *feature*) **const**
Check if the file system has the desired "read-only compatible feature".

Return `true` iff. the file system has the feature, `false` otherwise

See *compatible_feature*

Since 1.0

Parameters

- `feature`: The *read_only_compatible_feature* to check for

bool **has_all**(std::initializer_list<read_only_compatible_feature> **const** *features*) **const**
Check if the file system has all of the desired "read-only compatible features".

Parameters

- `features`: The

  iff. the file system has

  features that were querried,

  otherwise #compatible_feature 1.0

bool **has_any**(std::initializer_list<read_only_compatible_feature> **const** *features*) **const**
Check if the file system has at least one of the desired "read-only compatible features".

Parameters

- `features`: The

  iff. the file system has

  least one of the features that were querried,

  otherwise #compatible_feature 1.0

Public Members

u32 **inodes_count**
The total number of inodes in the file system.

u32 **blocks_count**
The total number of blocks in the file system.

u32 **reserved_blocks_count**
The number of blocks reserved for the super user.

u32 **free_blocks_count**
The number of free blocks in the file system.

u32 **free_inodes_count**
The mumber of free inodes in the file system.

u32 **first_data_block_id**
The first block that carries user data in the file system.

u32 **logical_block_size**
The logical size of a block (1024 << logical_block_size)

s32 **logical_fragment_size**
The logical size of a block (1024 << logical_fragment_size)

u32 **blocks_per_group**
The number of blocks per block group.

u32 **fragments_per_group**
The number of fragments per block group.

u32 **inodes_per_group**
The number of inodes per block group.

u32 **last_mount_timestamp**
The unix timestamp when the file system was last mounted.

u32 **last_write_timestamp**
The unix timestamp of the last write operation to the file system.

u16 **mount_count**
The number of times the file system was mounted since the last check.

u16 **maximum_mount_count**
The maximum number of times the file system can be mounted until a full check.

u16 **magic_number**
The magic number identifying the file system type.

s16 **state**
The state of the file system.

s16 **error_behaviour**
The desired behaviour if a file system error occurs.

s16 **revision_level_minor**
The minor revision level of the file system.

u32 **last_check_timestamp**
The unix timestamp of the last check of the file system.

u32 **check_interval**
The unix time interval in which to check the file system.

cos **creator_operating_system_id**
The operation system identifier of the OS that created the file system.

rlv **revision_level**
The revision level of the file system.

u16 **super_user_id**
The user ID of the super user.

u16 **super_user_group_id**
The group ID of the super user group.

u32 **first_inode_id**
The id of the first inode usable for standard files.

u16 **inode_size**
The size of an inode in bytes.

u16 **superblock_group_id**
The ID of the block group hosting this superblock.

cft **compatible_features_bitmap**
The active compatible features.

ift **incompatible_features_bitmap**
The active incompatible feature.

rft **read_only_compatible_features_bitmap**
The active features compatible with read-only mode.

u08_arr<16> **uuid**
The UUID of the file system.

chr_arr<16> **label**
The label of the file system.

chr_arr<64> **last_mount_point**
The location the file system was last mounted on.

cpr **compression_algorithms_bitmap**
The compression algorithms used in the file system.

u08 **file_preallocated_blocks_count**
The number of blocks to preallocate for a file.

u08 **directory_preallocated_blocks_count**
The number of block to preallocate for a directory.

u16 **_padding**
Alignment padding.

u08_arr<16> **journal_superblock_uuid**
The UUID of the superblock containing the journal.

u32 **journal_inode_id**
The ID of the inode hosting the journal.

u32 **journal_device_number**
The device number of the journal.

u32 **last_orphan_inode_id**
The first inode in the list of inodes to delete.

u32_arr<4> **hash_seed**
The seed for the directory hashing algorithm.

u08 **hash_version**
The version of the directory hashing algorithm.

u08_arr<3> **_reserved0**
Alignment padding.

u32 **default_mount_options**
The default mount options for the file system.

u32 **first_meta_block_group_id**
The ID of the first meta block group.

u08_arr<760> **_reserved1**
Padding.

Developer Tools

arraydump

arraydump is a utility to create hexdumps in different forms, suitable for comsumption by a C/C++ compiler. The tool is inspired by the well-known *xxd* utility which is part of *vim*. We created *arraydump* to overcome some weaknesses of *xxd*.

Advantages of arraydump over xxd

arraydump provides the following advantages over *xxd*:

1. Element type selection: *arraydump* allows you to specify the element-type of the array that will be generated. The currently supported types are `std::int8_t` (via *–type int8*), `std::uint8_t` (via *–type uint8*), `char` (via *–type char*), `signed char` (via *–type schar*), and `unsigned char` (via *–type uchar*).
2. Use `std::array<T,S>` instead of C-Style arrays: Since `xxd` was designed to work for **C** projects, it makes use of plain, old, C-Style arrays. `arraydump` has been designed for **C++** projects, and one of the decisions made during development was to use modern facilities in order to promote usage modern **C++**.
3. Support for processing multiple files at once: *arraydump* allows you to transform multiple files at once. You can specify a list of files and a directory for the generated files (via *–output <dir>*). This makes it easy and fast to transform a lot of files at once without having to resort to shell scripting magic.

Disadvantages of arraydump compared to xxd

Of course we live and work in an engineering world, and (almost) no tool comes with advantages alone. The folloing issues need to be considered when using *arraydump*.

1. Young project: *arraydump* is a very young tool. Because of this, it has not seen a lot of use outside the *extfs* project. This means that there are probably bugs that have not yet surfaced and might cause wrong output to be produced. If you find any bugs, please do not hessitate to report them, or even better create a pull request.
2. Written in Python: In contrast to *xxd*, which is written in **C**, *arraydump* is written in **Python**. There is nothing inherently bad about this, it just means that you will need a **Python 3** compatible interpreter on your system to use *arraydump*. You will need to keep that in mind if you, for example, use the tool in your *CI* setup. Additionally, being written in an interpreted language, *arraydump* will probably use more resources for processing than *xxd*.
3. Only **C++** header files can be generated: *xxd* provides several different output formats as well as different modes of operation. *arraydump*, on the other hand, was specifically designed to produce **C++** header files. That is all it can do.

Usage

The output of *arraydump -h* is pretty self-explanatory

```
usage: arraydump [-h] [--output dir] [--columns cols] [--extension ext]
                 [--type {int8,uint8,char,schar,uchar}]
                 file [file ...]

Convert binaries to C++ headers

positional arguments:
  file                  The input file to process
```

```
optional arguments:
  -h, --help            show this help message and exit
  --output dir          The target directory for the generated file(s)
  --columns cols        The number of columns in the output
  --extension ext       The file extension for the generated header
  --type {int8,uint8,char,schar,uchar}
                        The array element type
```

Symbols

B

C

D

E

F

H

I

J

L

M

O

R

S

U