littlefs-python

Release 0.3.0

Jan 10, 2022

Contents

littlefs-python provides a thin wrapper around littlefs, a filesystem targeted for small embedded systems. The wrapper provides a pythonic interface to the filesystem and allows the creation, inspection and modification of the filesystem or individual files. Even if this package uses Cython, the goal is not to provide a high performance implementation. Cython was chosen as an easy method is offered to generate the binding and the littlefs library in one step.

CHAPTER 1

Quick Examples

Let's create a image ready to transfer to a flash memory using the pythonic interface:

```
from littlefs import LittleFS

# Initialize the File System according to your specifications
fs = LittleFS(block_size=512, block_count=256)

# Open a file and write some content
with fs.open('first-file.txt', 'w') as fh:
    fh.write('Some text to begin with\n')

# Dump the filesystem content to a file
with open('FlashMemory.bin', 'wb') as fh:
    fh.write(fs.context.buffer)
```

The same can be done by using the more verbose C-Style API, which closely resembles the steps which must be performed in C:

```
from littlefs import lfs

cfg = lfs.LFSConfig(block_size=512, block_count=256)
fs = lfs.LFSFilesystem()

# Format and mount the filesystem
lfs.format(fs, cfg)
lfs.mount(fs, cfg)

# Open a file and write some content
fh = lfs.file_open(fs, 'first-file.txt', 'w') as fh:
lfs.file_write(fs, fh, b'Some text to begin with\n')
lfs.file_close(fs, fh)

# Dump the filesystem content to a file
with open('FlashMemory.bin', 'wb') as fh:
    fh.write(cfg.user_context.buffer)
```

CHAPTER 2

Installation

This is as simple as it can be:

```
pip install littlefs-python
```

At the moment wheels (which require no build) are provided for the following platforms, on other platforms the source package is used and a compiler is required:

- Linux: Python 3.6 - 3.10 / 32- & 64-bit
- Windows: Python 3.6 - 3.10 / 32- & 64-bit

CHAPTER 3

Development Setup

Start by checking out the source repository of littlefs-python:

```
git clone https://github.com/jrast/littlefs-python.git
```

The source code for littlefs is included as a submodule which must be checked out after the clone:

```
cd <littlefs-python>
git submodule update --init
```

this ensures that the correct version of littlefs is cloned into the littlefs folder. As a next step install the dependencies and install the package:

```
pip install -r requirements.txt
pip install -e .
```

Note: It's highly recommended to install the package in a virtual environment!

3.1 Development Hints

- Test should be run before commiting: *pytest test*
- Mypy is used for typechecking. Run it also on the tests to catch more issues: *mypy src test test/lfs*
- Mypy stubs can be generated with *stubgen src*. This will create a *out* direcotry containing the generated stub files.

CHAPTER 4

Creating a new release

- Make sure the master branch is in the state you want it.
- Create a tag with the new version number
- Wait until all builds are completed. A new release should be created automatically on github.
- Build the source distribution with *python setup.py sdist*
- Download all assets (using *ci/download_release_files.py*)
- Upload to pypi using twine: *twine upload dist/**

CHAPTER 5

Contents

5.1 Usage

littlefs-python offers two interfaces to the underlying littlefs library:

- A C-Style API which exposes all functions from the library using a minimal wrapper, written in Cython, to access the functions.
- A pythonic high-level API which offers convenient functions similiar to the ones known from the `os` standard library module.

Both API's can be mixed and matched if required.

5.1.1 C-Style API

The C-Style API tries to map functions from the C library to python with as little intermediate logic as possible. The possibility to provide customized `read()`, `prog()`, `erase()` and `sync()` functions to littlefs was a main goal for the api.

All methods and relevant classes for this API are available in the *littlefs.lfs* module. The methods where named the same as in the littlfs library, leaving out the *lfs_* prefix. Because direct access to C structs is not possible from python, wrapper classes are provided for the commonly used structs:

- *LFSFilesystem* is a wrapper around the `lfs_t` struct.
- *LFSFile* is a wrapper around the `lfs_file_t` struct.
- *LFSDirectory* is a wrapper around the `lfs_dir_t` struct.
- *LFSConfig* is a wrapper around the `lfs_config_t` struct.

All these wrappers have a `_impl` attribute which contains the actual data. Note that this attribute is not accessible from python. The *LFSConfig* class exposes most of the internal fields from the `_impl` as properties to provide read access to the configuration.

5.1.2 Pythonic API

While the pythonic API is working for basic operations like reading and writing files, creating and listing directories and some other functionality, it's by no means finished. Currently the usage is best explained in the *Examples* section.

5.2 Examples

5.2.1 Preparing a filesystem on the PC

In the following example shows how to prepare an image of a Flash / EEPROM memory. The generated image can then be written to the memory by other tools.

Start by creating a new filesystem:

```
>>> from littlefs import LittleFS
>>> fs = LittleFS(block_size=256, block_count=64)
```

It's important to set the correct `block_size` and `block_count` during the instantiation of the filesystem. The values you set here must match the settings which are later used on the embedded system. The filesystem is automatically formatted and mounted[1] during instantiation. For example, if we look at the first few bytes of the underlying buffer, we can see that the filesystem header was written:

```
>>> fs.context.buffer[:20]
bytearray(b'\x00\x00\x00\x00\xf0\x0f\xff\xf7littlefs/\xe0\x00\x10')
```

We can start right away by creating some files. Lets create a simple file containing some Information about the hardware[2]:

```
>>> with fs.open('hardware.txt', 'w') as fh:
...     fh.write('BoardVersion:1234\n')
...     fh.write('BoardSerial:001122\n')
18
19
```

File- and foldernames are encoded as ASCII. File handles of littlefs can be used as normal file handles, using a context manager ensures that the file is closed as soon as the `with` block is left.

Let's create some more files in a configuration folder:

```
>>> fs.mkdir('/config')
0
>>> with fs.open('config/sensor', 'wb') as fh:
...     fh.write(bytearray([0x01, 0x02, 0x05]))
3
>>> with fs.open('config/actor', 'wb') as fh:
...     fh.write(bytearray([0xAA, 0xBB] * 100))
200
```

As we wan't to place the files in a folder, the folder first needs to be created. The filesystem does not know the concept of a working directory. The working directory is allways assumed to be the root directory, therefore `./config`, `/config` and `config` have all the same meaning, use whatever you like the best.

A final check to see if all required files are on the filesystem before we dump the data to a file:

[1] See *littlefs.lfs.format()* and *littlefs.lfs.mount()* for further details.
[2] Ignore the output of the examples. These are the return values in which we are not interested in almost all cases.

```
>>> fs.listdir('/')
['config', 'hardware.txt']
>>> fs.listdir('/config')
['actor', 'sensor']
```

Everything ok? Ok, lets go and dump the filesystem to a binary file. This file can be written/downloaded to the actual storage.

```
>>> with open('fs.bin', 'wb') as fh:
...     fh.write(fs.context.buffer)
16384
```

5.2.2 Inspecting a filesystem image

Sometimes it's necesary to inspect a filesystem which was previously in use on a embedded system. Once the filesystem is available as an binary image, it's easy to inspect the content using littlefs-python.

In this example we will inspect the image created in the last example. We check if the actor file is still the same as when the image was written. We start again by creating a *LittleFS* instance. However, this time we do not want to mount the filesystem immediateley because we need to load the actual data into the buffer first. After the buffer is initialized with the correct data, we can mount the filesystem.

```
>>> fs = LittleFS(block_size=256, block_count=64, mount=False)
>>> with open('fs.bin', 'rb') as fh:
...     fs.context.buffer = bytearray(fh.read())
>>> fs.mount()
0
```

Let's see whats on the filesystem:

```
>>> fs.listdir('/config')
['actor', 'sensor']
```

Ok, this seems to be fine. Let's check if the *actor* file was modified:

```
>>> with fs.open('/config/actor', 'rb') as fh:
...     data = fh.read()
>>> assert data == bytearray([0xAA, 0xBB] * 100)
```

Great, our memory contains the correct data!

Now it's up to you! Play around with the data, try writing and reading other files, create directories or play around with differnt `block_size` and `block_count` arguments.

5.3 API Documentation

5.3.1 littlefs module

class littlefs.**FileHandle**(*fs*, *fh*)

close()
Flush and close the IO object.

This method has no effect if the file is already closed.

flush()
Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

readable()
Return whether object was opened for reading.

If False, read() will raise OSError.

readall()
Read until EOF, using multiple read() call.

readinto(*b*)

seek(*offset*, *whence=0*)
Change stream position.

Change the stream position to the given byte offset. The offset is interpreted relative to the position indicated by whence. Values for whence are:

- 0 – start of stream (the default); offset should be zero or positive
- 1 – current stream position; offset may be negative
- 2 – end of stream; offset is usually negative

Return the new absolute position.

seekable()
Return whether object supports random access.

If False, seek(), tell() and truncate() will raise OSError. This method may need to do a test seek().

tell()
Return current stream position.

truncate(*size=None*) → int
Truncate file to size bytes.

File pointer is left unchanged. Size defaults to the current IO position as reported by tell(). Returns the new size.

writable()
Return whether object was opened for writing.

If False, write() will raise OSError.

write(*data*)

class littlefs.**LittleFS**(*context: UserContext = None*, ***kwargs*)
Littlefs file system

context
User context of the file system

format() → int
Format the underlying buffer

listdir (*path='.'*) → List[str]
List directory content

List the content of a directory. This function uses *scandir()* internally. Using *scandir()* might be better if you are searching for a specific file or need access to the *littlefs.lfs.LFSStat* of the files.

makedirs (*name: str*, *exist_ok=False*)
Recursive directory creation function.

mkdir (*path: str*) → int
Create a new directory

mount () → int
Mount the underlying buffer

open (*fname: str*, *mode='r'*, *buffering: int = -1*, *encoding: str = None*, *errors: str = None*, *newline: str = None*) → IO
Open a file.

`mode` is an optional string that specifies the mode in which the file is opened and is analogous to the built-in `io.open()` function. Files opened in text mode (default) will take and return *str* objects. Files opened in binary mode will take and return byte-like objects.

Parameters

- **fname** (*str*) – The path to the file to open.
- **mode** (*str*) – Specifies the mode in which the file is opened.
- **buffering** (*int*) – Specifies the buffering policy. Pass *0* to disable buffering in binary mode.
- **encoding** (*str*) – Text encoding to use. (text mode only)
- **errors** (*str*) – Specifies how encoding and decoding errors are to be handled. (text mode only)
- **newline** (*str*) – Controls how universal newlines mode works. (text mode only)

remove (*path: str*) → int
Remove a file or directory

If the path to remove is a directory, the directory must be empty.

Parameters **path** (*str*) – The path to the file or directory to remove.

removedirs (*name*)
Remove directories recursively

This works like *remove()* but if the leaf directory is empty after the successfull removal of `name`, the function tries to recursively remove all parent directories which are also empty.

rename (*src: str*, *dst: str*) → int
Rename a file or directory

rmdir (*path: str*) → int
Remove a directory

This function is an alias for *remove()*

scandir (*path='.'*) → Iterator[LFSStat]
List directory content

stat (*path: str*) → LFSStat
Get the status of a file or directory

unlink(*path: str*) → int
Remove a file or directory

This function is an alias for *remove()*.

walk(*top: str*) → Iterator[Tuple[str, List[str], List[str]]]
Generate the file names in a directory tree

Generate the file and directory names in a directory tree by walking the tree top-down. This functions closely resembels the behaviour of `os.stat()`.

Each iteration yields a tuple containing three elements:

- The root of the currently processed element
- A list of directorys located in the root
- A list of filenames located in the root

5.3.2 littlefs.context module

class littlefs.context.**UserContext**(*buffsize: int*)
Basic User Context Implementation

erase(*cfg: LFSConfig*, *block: int*) → int
Erase a block

Parameters

- **cfg** (*LFSConfig*) – Filesystem configuration object
- **block** (*int*) – Block number to read

prog(*cfg: LFSConfig*, *block: int*, *off: int*, *data: bytes*) → int
program data

Parameters

- **cfg** (*LFSConfig*) – Filesystem configuration object
- **block** (*int*) – Block number to program
- **off** (*int*) – Offset from start of block
- **data** (*bytes*) – Data to write

read(*cfg: LFSConfig*, *block: int*, *off: int*, *size: int*) → bytearray
read data

Parameters

- **cfg** (*LFSConfig*) – Filesystem configuration object
- **block** (*int*) – Block number to read
- **off** (*int*) – Offset from start of block
- **size** (*int*) – Number of bytes to read.

sync(*cfg: LFSConfig*) → int
Sync cached data

Parameters **cfg** (*LFSConfig*) – Filesystem configuration object

5.3.3 littlefs.lfs module

class littlefs.lfs.**LFSStat**
Littlefs File / Directory status

name
Alias for field number 2

size
Alias for field number 1

type
Alias for field number 0

class littlefs.lfs.**LFSConfig**(*context=None*, ***kwargs*)

block_count

block_size

cache_size

lookahead_size

prog_size

read_size

class littlefs.lfs.**LFSDirectory**

class littlefs.lfs.**LFSFile**

flags
Mode flags of an open file

class littlefs.lfs.**LFSFileFlag**
Littlefs file mode flags

append = 2048

creat = 256

excl = 512

rdonly = 1

rdwr = 3

trunc = 1024

wronly = 2

class littlefs.lfs.**LFSFilesystem**

littlefs.lfs.**dir_close**(*LFSFilesystem fs*, *LFSDirectory dh*)

littlefs.lfs.**dir_open**(*LFSFilesystem fs*, *path*)

littlefs.lfs.**dir_read**(*LFSFilesystem fs*, *LFSDirectory dh*)

littlefs.lfs.**dir_rewind**(*LFSFilesystem fs*, *LFSDirectory dh*)

littlefs.lfs.**dir_tell**(*LFSFilesystem fs*, *LFSDirectory dh*)

littlefs.lfs.**file_close**(*LFSFilesystem fs*, *LFSFile fh*)

littlefs.lfs.**file_open**(*LFSFilesystem fs*, *path*, *flags*)

littlefs.lfs.**file_open_cfg**(*self*, *path*, *flags*, *config*)

littlefs.lfs.**file_read**(*LFSFilesystem fs*, *LFSFile fh*, *size*)

littlefs.lfs.**file_rewind**(*LFSFilesystem fs*, *LFSFile fh*)

littlefs.lfs.**file_seek**(*LFSFilesystem fs*, *LFSFile fh*, *off*, *whence*)

littlefs.lfs.**file_size**(*LFSFilesystem fs*, *LFSFile fh*)

littlefs.lfs.**file_sync**(*LFSFilesystem fs*, *LFSFile fh*)

littlefs.lfs.**file_tell**(*LFSFilesystem fs*, *LFSFile fh*)

littlefs.lfs.**file_truncate**(*LFSFilesystem fs*, *LFSFile fh*, *size*)

littlefs.lfs.**file_write**(*LFSFilesystem fs*, *LFSFile fh*, *data*)

littlefs.lfs.**format**(*LFSFilesystem fs*, *LFSConfig cfg*)
Format the filesystem

littlefs.lfs.**fs_size**(*LFSFilesystem fs*)

littlefs.lfs.**getattr**(*LFSFilesystem fs*, *path*, *type*, *buffer*, *size*)

littlefs.lfs.**mkdir**(*LFSFilesystem fs*, *path*)

littlefs.lfs.**mount**(*LFSFilesystem fs*, *LFSConfig cfg*)
Mount the filesystem

littlefs.lfs.**remove**(*LFSFilesystem fs*, *path*)
Remove a file or directory

If removing a direcotry, the directory must be empty.

littlefs.lfs.**removeattr**(*LFSFilesystem fs*, *path*, *type*)

littlefs.lfs.**rename**(*LFSFilesystem fs*, *oldpath*, *newpath*)
Rename or move a file or directory

If the destination exists, it must match the source in type. If the destination is a directory, the directory must be empty.

littlefs.lfs.**setattr**(*LFSFilesystem fs*, *path*, *type*, *buffer*, *size*)

littlefs.lfs.**stat**(*LFSFilesystem fs*, *path*)
Find info about a file or directory

littlefs.lfs.**unmount**(*LFSFilesystem fs*)
Unmount the filesystem

This does nothing beside releasing any allocated resources

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

R

S

T

U

W