FunFS Demo Driver

Introduction

This file introduces the `FunFS` filesystem library, which implements a demo Ext2 filesystem driver in Haskell.

A PDF version of this file can be found in the `doc` subfolder.

Description

`FunFS` (Functional FileSystem) is a functional implementation of the Second Extended Filesystem (Ext2) in Haskell. It features a lazy-evaluated, monadic UNIX-like API which hides the details of the underlying filesystem, allowing replacements to be dropped in at a later time without modifying client code.

Building

Build system

The project is built using `cabal`, the Haskell build system which is distributed with the Haskell Platform.

The Haskell Platform can be downloaded from https://www.haskell.org/platform/.

Dependencies

`FunFS` depends on the following packages:

Package name	Minimum	Maximum
`ghc`	`7.2.1`	
`base`	`4.9`	`4.10`
`binary`	`0.8.3`	`0.9`
`bytestring`	`0.10.8.1`	`0.11`
`mtl`	`2.2`	`2.3`
`QuickCheck`	`2.9.2`	`2.10`
`template-haskell`	`2.11`	`2.12`
`time`	`1.6.0.1`	`1.7`
`transformers`	`0.5.2`	`0.6`
`uuid`	`1.2.2`	`1.3`

Most of these are distributed with the Haskell Platform. The only ones which should need to be installed manually are:

- `quickcheck`
- `uuid`

Due to the dependencies on `ghc` and `template-haskell`, which is a `GHC` extension, other Haskell compilers like `hugs` will probably not work.

Once Haskell is installed, any packages can be installed with `cabal`, the Haskell package manager:

```
$ cabal update
```

followed by

```
$ cabal install <package name>
```

Library and examples

Finally, the library and examples can be built with `cabal`:

```
$ cabal configure
$ cabal build
```

This builds the library and examples.

Running the examples

Before executing the examples we should create an Ext2 filesystem:

```
$ dd if=/dev/zero of="$device" bs=1M count=64
$ mke2fs -r0 "$device"
```

This creates a 64 MB filesystem and formats it as Ext2 revision 0. The file can now be used to run the examples:

- `Example0_Mount`

  The `mount` example mounts a filesystem and prints the contents of its superblock and block group descriptor table.

  ```
  $ dist/build/mount/mount "$device"
  ```

  Its output can be compared to the output of the `dumpe2fs` command on Linux.

- `Example1_ReadDir`

  The `readdir` example mounts a filesystem, searches for a path, and lists the contents of the directory named by the path.

  ```
  $ dist/build/readdir/readdir "$device" "$path"
  ```

  Its output can be compared to `ls -al`.

Note: To run the real `ls` on , we will need to use UNIX loop devices. This will be explained in the next section.

- `Example2_ReadFile`

  The `readfile` example mounts a filesystem, searches for a path, and dumps the contents of the file named by the path.

  ```
  $ dist/build/readfile/readfile "$device" "$path"
  ```

  Its output can be compared with `cat`.

  However, it's important to point out that a fresh Ext2 filesystem does not contain any files, just two directories: `/` and `/lost+found`.

  To test this example we will need to create a file on the Ext2 filesystem. The following commands create a 'loop-back device' using `losetup` and mount a filesystem using `mount`. (This requires superuser access.)

  ```
  $ loop=$(losetup -f "$device")
  $ mount -o ro "$loop" "$mountpoint"
  ```

  NB: The `-o ro` flags to `mount` are very important. Without this, the Ext2 driver will update the superblock to revision 1, which is not supported by `FunFS`.

  A 50 MiB file containing random data is created on the filesystem, which is then unmounted. We also keep a local copy of the file for comparison.

  ```
  $ dd if=/dev/urandom of="$mountpoint/foo" bs=1M count=50
  $ cp "$mountpoint/foo" "bar"
  $ umount "$loop"
  $ losetup -d "$loop"
  ```

Finally, `readfile` is executed to get the contents of the file and its output is compared to the original file using `diff`.

```
$ dist/build/readfile/readfile "$device" "/foo" >foo
$ diff foo bar
```

The return value of `diff` (`$?`) will be 0 if the files are the same. As `foo` and `bar` are binary files, if their contents are different, `diff` will say `"Binary files differ"`.

Testing done

The FunFS examples have been tested on filesystems ranging from `1 MiB` to `17 GiB` in size, formatted with `Ext2` revisions 0 and 1 as well as `Ext3` and `Ext4`, the latter of which was a live filesystem. This process is detailed in the final report.

API Documentation

Documentation for the public interfaces of every module is in the `doc/html` directory. This is automatically-generated HTML documentation pulled directly from source code but is quite detailed and neatly formatted. This can be viewed in the browser at `doc/html/index.html`.

State of Development

- The current development status is such that Ext2 revision 0 filesystems can be parsed and mounted with sanity checks to verify compatibility.
- Files and directories can be opened and read using traditional UNIX-style `/`-delimited paths, although support for Windows-style paths can be added easily. This includes support for the `.` and `..` elements, which refer to the current directory and parent directory respectively.
- There is limited support for user permissions. For example, using the `openDirAs` function from `Data.Filesystem.FunFS.API.Directory` on the `/lost+found` directory with a user ID other than 0 triggers an error.
- The filesystem driver is currently read-only, but most of the logic for writing data to the disk is already present. Additionally, although only Ext2 revision 0 is supported, later revisions as well as Ext3 and 4 filesystems can also be read if certain sanity checks are disabled (specifically, against `sRevLevel` and `sFeaturesIncompat` in `SuperBlock`).
- FunFS uses the older linked-list directory model, but due to the backwards-compatible design of Ext2 revision 1, Ext3 and Ext4, the hash-indexed directories used by those filesystems are also readable. Additionally, although Ext2 revision 0 uses inodes fixed at sizes of 128 bytes, FunFS can read filesystems using larger inodes provided `sInodeSize` is correct in the superblock. There is no hardcoding of filesystem parameters such as the block size -- all of these are taken from the superblock, so any sane values will work.
- There is no support for I/O with 'special' files such as character devices (e.g. terminals), block devices (e.g. disks), inter-process pipes or network sockets. This is because these are part of the POSIX virtual filesystem, for which the operating system is responsible. The driver treats any attempted operations on special files as an error.
- While there is currently support in the API only for mounting filesystems from disks, it would be relatively easy to add support for other sources, such as the network.

Please also see the file ChangeLog.md

Known Bugs

- The library triggers an exception if the filesystem does not end on a block boundary -- in other words, if the size of the filesystem is not a multiple of the block size.

Wish-list

Proper error handling

The current error handling in the driver is messy at best. Some functions indicate failure using `Maybe`. For example, in the API, any attempt to find out the details of a closed handle (such as the size of the file it references, or its current read/write position) results in a return value of `Nothing`. However, most errors trigger uncatchable exceptions using the Haskell standard library function `Prelude.error`, which simply terminates the program after printing a message. This is fine for debugging the logic of the program but would not be acceptable in production code.

A better solution would use error-indicating values similar to the POSIX `errno` interface, but using high-level types with more context, such as the file name, line number and function name of the function triggering the error, or even a trace of the call stack. This would allow calling functions to decide how to handle an error, and if the error is deemed unrecoverable, the error value could be mapped to a standard string as with the C `strerror` function, and printed along with the extra context.

Writing support

The basic logic for writing to the filesystem is present in `Block`, but it is not used and has only been tested minimally with unit tests. Proper writing support would require selecting a block group, allocating blocks and inodes, and updating the block and inode bitmaps appropriately. `Node` would then provide a simplified interface to allow manipulation of the file as a `ByteString`.

This feature would enable creating new filesystems which could then be verified using the `e2fsck` program. It would also enable automated testing, in which a filesystem would be created containing random files, then remounted to see if the files' contents can be read back exactly. At present this is difficult because Linux changes the filesystem version and flags, so the filesystem sanity checks have to be disabled. Since `cabal` does not support