Files as Directories: Some Thoughts on Accessing Structured Data within Files

Raphael Wimmer
University of Regensburg
Regensburg, Germany
raphael.wimmer@ur.de

ABSTRACT

This paper explores the concept of *files as directories* (FAD) as a unified interface to structured data within a file by representing such a file as (virtual) directory and the structured data as subdirectories and subfiles.

Transparent conversion of files and their structured data into directory trees is to be handled by virtual filesystem providers. This allows for arbitrary applications and programming languages to read and write data within supported file types without the need to understand the file format - e.g., in order to modify pixel values within an image file, paragraphs in a text document, or settings in a configuration file.

Advantages of this approach compared to API-mediated file access might include better learnability, modularity, explorability, synchronous access, better integration of proprietary applications, and a few other nice features. While technical issues of the FAD concept have been discussed by developers in the past, no major operating system allows FAD at the moment. In this paper I present concept, advantages, limitations, and use cases of FAD.

CCS CONCEPTS

• **Human-centered computing** → **Command line interfaces**; • **Information systems** → *Directory structures*; • **Software and its engineering** → *File systems management*; *Data types and structures*; Interface definition languages;

KEYWORDS

programming, filesystems, APIs, concept

ACM Reference Format:
Raphael Wimmer. 2018. Files as Directories: Some Thoughts on Accessing Structured Data within Files. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3191697.3214323

1 MOTIVATION

As more and more physical media and workflows are transformed into digital ones, users lose control over them. Whereas physical desktops allow knowledge workers to customize their workspace and workflows to their liking, their digital counterparts constrain the users' configuration options to those explicitly implemented by the developers of operating system and applications.

In order to empower end users to customize and automate their digital workspaces and workflows, it has been suggested repeatedly that 'everyone should learn to code'. However, as Ko et al. [1] show, novice end-user programmers face several learning barriers. Even if a user is able to formulate an algorithmic solution to a problem, they have difficulty in finding the right functions to use, using them, combining them, understanding the system state, and understanding the external behavior.

Furthermore, the choice of programming language also limits the choice of APIs that can be used. For example, a user might want to write a tool that modifies clipboard contents. Programming languages / ecosystems that do not offer an API for the system clipboard would be useless for that user - even if the language itself were powerful and easy to learn.

In many cases, knowledge workers might actually need better end-user *configuration* tools, similar to those in system administration. System administrators can be considered end-user programmers — instead of developing all software for their servers and workstations themselves, they select, combine, and configure existing software. In many cases, system administrators also write glue code in a scripting language to ease configuration or get multiple components to work together. The UNIX approach to system management - storing configuration data in plain-text files and making system state available through device files - combined with its pipes and filters[1] paradigm offers multiple advantages for system administration. For example, users can easily explore configuration files, create backups, or annotate them. Furthermore, nearly all configuration tasks can be performed using any programming language that supports reading and writing files.

However, the system administration approach can not be easily transferred to workspace customization. In most cases, users do not only work with plain-text files but with documents in a variety of file formats. In order to programmatically read or modify such files, a user would need to learn how to use an appropriate API — if it is even available for one of the programming languages they know.

[1] http://doc.cat-v.org/unix/pipes/

In this paper I present *files as directories*[2] (FAD), a modification of the traditional hierarchical file system concept that allows for accessing structured data within a file using a tree of virtual subdirectories and subfiles within the file. Transparent conversion between both representations is handled by modular virtual file systems. The concept allows for users to access and modify contents of arbitrary files using a set of simple file operations (e.g., open/close, read/write) that are supported by most programming languages. A major benefit of FAD is that it allows end-user programmers to

- interactively explore the data they work with,
- use their preferred tools and programming languages, and
- focus on the algorithmic aspects of a problem instead of learning how to use specific APIs.

The main contributions of this paper are description, history, and discussion of the *files as directories* concept. The focus is on the user-facing aspects of the concept. Implementation challenges and semantics are only discussed in passing. So far, no implementation of the FAD concept exists.
The paper is organized as follows: In the next section, *Treating files with structured data as directories*, I describe the basic concept of FAD. In the *Related Work* section, I give an overview of similar approaches and previous discussions of the concept. In the *Usage Examples* section, basic workings and versatility of the proposed approach are demonstrated. Afterwards I discuss *Advantages and Limitations* and present an outlook on *Future Work*.

2 TREATING FILES WITH STRUCTURED DATA AS DIRECTORIES

Files provide an universal interface to data. Operating systems provide APIs and basic tools for file administration. Many programming languages offer support for file reading and writing. Most applications allow accessing files.
I propose to provide a unified interface to structured data within a file by representing such a file as (virtual) directory and the structured data as subdirectories and subfiles. This allows for arbitrary applications to read and write structured data within arbitrary files without the need to understand the file format.
The *files as directories* concept (FAD) proposed in this paper has the following defining properties:

- Some or any files on a computer can also be accessed as if they were directories - such files are called *dir files* in the remainder of the paper.
- The directory tree under a dir file represents one or more views on the structured data within the dir file.
- The files in the directory tree represent individual elements of the structured data. Some of them may also be accessed as dir files themselves.

The FAD concept could also be formulated as *directories as files* (DAF): Some directories can be treated as if they were files. These files contain a representation of the directory contents e.g., as plain text or in a structured file format.

[2]sometimes also called *file as directory* (singular); in this paper, both variants are treated as synonyms.

Considering, for example, a simple spreadsheet stored in a CSV file called `my.csv`, the following operations would be possible if `my.csv` were a dir file:

Listing 1: Accessing dir files

```
$ cat my.csv
animal,legs
lion,4
snail,0

$ ls -la my.csv/
rows/
cols/
.metadata/

$ cat my.csv/rows/0
animal, legs

$ ls my.csv/rows/0/
cols/

$ echo "parrot, 2" > my.csv/rows/3
$ cat my.csv/cols/0
animal
lion
snail
parrot
```

The semantics of the directory structure have been arbitrarily chosen for the examples presented in this paper. In order to use FAD in practice, coherent, intuitive, and robust semantics would need to be developed.

3 RELATED WORK

There already exist a few lightweight, non-generic implementations of the *file as directory* (FAD) concept: archive files (e.g., using the PKZIP file format) are an obvious example of files containing other files and directories. On all major desktop operating systems, the default file managers allow opening archive files as if they were directories.[3] A similar approach is chosen for audio CDs (which are neither files nor directories) in some file managers. When the user 'enters' the CD icon, the file manager shows a list of the audio tracks. These can be copied to another drive, whereby the CD audio is automatically and implicitly converted into PCM WAV, MP3 or other formats on copying. In both of these cases, the representation as a directory tree is provided and implemented by the (GUI) application, not by the operating system itself.
In the *Plan 9* operating system, specialized filesystems present structured data as directory trees. For example, `upasfs`[4] represents emails as directories:

> "Each message in the mailbox becomes a numbered directory in the mailbox directory, and each attachment becomes a numbered directory in the message directory.

[3]e.g., Windows Explorer, macOS Finder, Dolphin (KDE), Nautilus (Gnome), or Midnight Commander (cross-platform). On Linux, *A Virtual File System (avfs)* is a FUSE-based filesystem in userspace which allows for accessing files within an archive or on the network via a special syntax (`/home/user/archive.tar.gz#ugz#utar/path/file`).
[4]http://plan9.bell-labs.com/magic/man2html/4/upasfs

> Since an attachment may itself be a mail message, this structure can recurse ad nauseam."

Plan 9 does not implement FAD, however, as files can not be accessed as directories. The raw email file is simply placed into the message directory together with files containing subject, sender, etc.

Finally, the FAD concept presented here from a user / application perspective has been discussed within the Linux kernel community multiple times with a focus on implementation challenges:

In 2004, Hans Reiser implemented FAD in his Reiser4 filesystem. *Pseudo files*[5] located in a `/metas/` top-level directory could hold metadata in files that also acted as directories (i.e., could be accessed both using the `read` and `readdir` system calls). Kernel developer Christoph Hellwig argued against inclusion of Reiser4 in the Linux kernel due to several fundamental problems that might be caused by the unexpected behavior of files that also appeared as directories.[6]. A long discussion ensued[7]. So far, the Reiser4 filesystem has not been included in the Linux kernel. Jamie Lokier renamed the thread and argued in "The argument for fs assistance in handling archives"[8] that it would be more convenient for the user to explore archive files with the `cd` command, and that treating archives as directories might be more time- and space-efficient on the filesystem level.

In April 2007, Theodore Tso described an inherent limitation of filesystems that also applies to FAD[9]:

> "One of the big problems of using a filesystem as a DB is the system call overheads. If you use huge numbers of tiny files, then each attempt read an atom of information from the DB takes three system calls — an open(), read(), and close(), with all of the overheads in terms of dentry and inode cache."

In May 2007, Miklos Szeredi argued for FAD independently of the Reiser4 debate[10]. He suggested to use FAD for archive files, "accessing streams, resource forks or extended attributes". He also provided a proof-of-concept kernel patch that utilized *avfs*. While a constructive debate ensued, FAD support was not integrated into the Linux kernel.

To my knowledge, FAD is not supported by any other major operating systems. However, FAD shares some properties with other implementations and concepts. The NTFS filesystem for Microsoft Windows allows files to contain multiple *Alternate Data Streams*, however it does not allow these to be subdirectories. HTTP allows for treating URLs with a trailing slash differently than URLs without one and allows for content negotiation. The REST paradigm already implements parts of the semantics of FAD. As most existing applications do not natively support accessing files via HTTP PUT requests or WebDAV, HTTP/REST is not a practical substitute for FAD.

[5]https://web.archive.org/web/20070921094112/http://www.namesys.com:80/v4/pseudo.html
[6]https://lkml.org/lkml/2004/8/24/220
[7]http://yarchive.net/comp/linux/reiser4.html
[8]https://marc.info/?l=reiserfs-devel&m=109406945802488
[9]http://yarchive.net/comp/linux/reiser4.html
[10]https://marc.info/?l=linux-fsdevel&m=117986025907108

4 USAGE EXAMPLES

Commonly suggested uses for FAD are archive files (.zip, .tar.gz, ...) and config files. However, FAD may also be used for working with office documents (text documents, spreadsheets, presentations), XML/HTML files, emails (including MIME-encoded attachments), calendars, multimedia files, or device files representing the state of a system (e.g., connected USB devices). Windows and widgets in a graphical desktop environment might also be represented as directory trees [2,3].

The following examples use a basic UNIX shell syntax for demonstrating two relatively straightforward uses of FAD. In the first example individual elements of a configuration file are accessed.

Listing 2: Accessing structured textual data via subfiles

```
$ cat foo.ini
[Section 1]
val=123
[Section 2]
val=333
val2="tree"

$ ls foo.ini/
Section 1
Section 2

$ cat foo.ini/Section\ 1
val=123

$ cat foo.ini/Section\ 1/
cat: foo.ini/Section\ 1:  Is a directory

$ ls foo.ini/Section\ 1/
val

$ echo "124" > foo.ini/Section\ 1/val

$ cat foo.ini/Section\ 1/val
124
```

In the following example individual pixels of an image file are modified.

Listing 3: Reducing the brightness of an image via its subfiles.

```
$ ls -la foo.jpg/
cols/
rows/
pixels/
formats/
.metadata/

# using a hypothetical script 'subtract.sh'
# that gets value and pixel file passed
# as parameters:
$ for px in foo.jpg/pixels/px*;\
  do subtract.sh 40 "$px"; done
```

FAD also facilitates conversion from one file format to another. Users do not need to know which formats are available or which converters to use. Available formats can be explored e.g., by reading the directory `formats/`. A file can offer multiple representations of its contents. However, not all conversions might be sensible or possible. Invalid conversions can be blocked by returning an error on write access to the subfile:

Listing 4: Converting images from SVG format to BMP format, and replacing the content of one image file with another.

```
$ cp image.svg/formats/image.bmp ./foo.bmp

$ cp foo.bmp image.svg/formats/image.bmp
cp: write error: Invalid argument

$ cp foo.bmp image2.jpg/formats/image2.bmp
```

As each of those subfiles offered by the converting file system in `formats/` may in turn offer other conversion formats, file conversions can also be chained together:

Listing 5: Conversion file systems allow e.g., converting an image into an audio file and playing it.

```
$ playsound foo.jpg/sizes/100x72/formats\
            /foo.bmp/pixels/10:10-20:70\
            /formats/foo.int_array/formats\
            /foo.wav
```

5 POTENTIAL ADVANTAGES

Everything that would be made possible by the FAD concept can already be implemented by using specialized libraries, APIs, and programming languages. However, FAD might offer a few advantages for end-user programmers and software developers.

Learnability. End-user programmers might benefit from a reduced set of methods they need to learn and apply. Instead of learning how to use a set of different APIs, they can focus on algorithmic aspects of the problem.

Explorability. Instead of reading API documentation on how to access certain information in a file, they can interactively explore the available options on the command line or a file manager. If they know which value a certain property has at the moment, they can use a full-text search in order to find the property's location in the subdirectory tree.

Composability. As subfiles can represent even complex data types (such as compressed images, 3D meshes, or audio waveforms) as a tree of simple values, generic tools following the UNIX *pipes and filters* paradigm can be chained and combined to transform and modify this data. Users can also choose the most appropriate programming language for a given sub-task.

Support for 'legacy' applications. FAD allows for integrating 'legacy' applications into workflows. For example, a proprietary application that can only read and write images in TIFF format could be pointed to the TIFF representation of a JPEG file (`foo.jpg/formats/foo.tif`).

OS support. By using FAD for accessing structured data, developers benefit from operating system features, such as access management, mounting files over a network, additional OS-specific management tools

Concurrent access. Two or more users or applications might operate on data in the same file as long as the accessed elements do not overlap or interact with each other. For example, one user might edit an image within a slideshow while another user simultaneously adds text to another slide.

Implementation issues of FAD that limit the usefulness of these features are discussed in the next section.

6 LIMITATIONS AND IMPLEMENTATION ISSUES

As multiple authors on the Linux kernel mailing list have argued, it is not trivial to implement FAD in a robust and consistent way on existing operating systems. In the following I describe some of the challenges that would need to be addressed when implementing FAD.

A major issue is backwards-compatibility with legacy applications that expect an object in the file system to be either a file or a directory - and not both. Furthermore, FAD might reduce I/O performance and increase storage requirements because data in subfiles would need to be extracted from a file and re-integrated into it transparently. Multi-level caching might be needed to mitigate such performance losses.

Sensible semantics and structures of the directory trees for each file format need to be found. In at least some cases, the file/directory metaphor might break or can not replace an API. For example, a complex API call with multiple parameters might be difficult to translate into a directory structure.

If file operations do not succeed, the user might only get a much more ambiguous error message than if they had used a specialized API for accessing the data.

Automatic conversion between file types is limited by conversion losses. For example, it might be sensible to provide access to a bitmap version of a vector graphics image via `bird.svg/formats/bird.bmp`. However, a bitmap image written to this subfile could not be losslessly converted into a vector graphics image.

Finally, care needs to be taken when accessing data with an inherent ordering such as characters, words, and lines in a text file via FAD. While a directory also stores the order of its files, tools and programming languages APIs might implicitly sort these entries alphabetically, removing important information about the file's structure.

Furthermore, there is a lack of examples of specific use cases, where FAD would bring major benefits over existing approaches.

Overall, while I am confident that many of these challenges can be addressed in an actual FAD implementation, trade-offs and design decisions will need to be made that will reduce the simplicity of the proposed concept.

7 FUTURE WORK

In this paper I have presented *files as directories* (FAD), a generic concept for providing access to structured data within files through virtual subdirectories and subfiles within these files. I have presented related work, abstract semantics, and a few simple use cases illustrating this concept. While FAD might offer quite a few advantages for end-user programmers, many implementation details need to be figured out. As next steps, it would be necessary to identify sensible semantics for the directory structures for different file formats. This might be done using *user elicitation* studies and informed by collecting further uses cases. Furthermore, a more

thorough investigation of implementation issues and approaches is needed. An initial prototype might be implemented based on HTTP/REST or FUSE and a custom translation shell.
It might turn out that FAD does not offer any advantages in practice or can not be implemented robustly. In any case, the FAD concept might provide ideas that extend the vocabulary and design space available to developers and programming language designers.

ACKNOWLEDGMENTS

I would like to thank the reviewers and participants of the *Salon des Refusés* for their valuable and friendly feedback. This project is funded by the Bavarian State Ministry of Education, Science and the Arts in the framework of the Centre Digitisation.Bavaria (ZD.B).

REFERENCES

[1] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, 199–206.
[2] Rob Pike. 1991. 8 ½, the plan 9 window system. In *Proc. AUUG '91*, 231–239.
[3] Raphael Wimmer and Fabian Hennecke. 2010. Everything is a window: Utilizing the window manager for multi-touch interaction. In *Workshop "engineering patterns for multi-touch interfaces" in conjunction with ACM EICS 2010.*