View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

A Parts-of-file File System

Yoann Padioleau, Olivier Ridoux

To cite this version:

Yoann Padioleau, Olivier Ridoux. A Parts-of-file File System. USENIX 2005 Annual Technical Conference, General Track, Apr 2005, Anaheim, CA, United States. hal-03214500

HAL Id: hal-03214500

https://hal.archives-ouvertes.fr/hal-03214500

Submitted on 1 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Parts-of-file File System

Yoann Padioleau and Olivier Ridoux
IRISA / University of Rennes 1
Campus universitaire de Beaulieu
35042 RENNES cedex, FRANCE
{padiolea,ridoux}@irisa.fr, http://www.irisa.fr/lande

Abstract

The *Parts-of-file File System* (PofFS) allows read-write accesses to different *views* of a given file or set of files in order to help the user separate and manipulate different concerns. The set of files is considered as a mount point from which views can be selected as read-write files *via* directories. Paths are formulas mentioning properties of a desired view. Each directory contain a file (the view) which contains the parts of the mounted files that satisfy the properties. This service is offered generically at the file system level, and a plug-in interface permits that file formats, or application-specific details are handled by user-defined operators. Special plug-ins called *transducers* can be defined for automatically attaching properties to parts of files. Performances are encouraging; files of 100 000 lines are handled efficiently.

1 Introduction

A typical user working on a digital document performs alternatively one of the three following operations: *searching* for a desired piece of data, *understanding* a piece of data and possibly its relationship with other pieces of the document, and *updating* coherently related pieces. Examples of this situation can be found with textual documents, such as source programs or reports, text databases such as BibTeX files, agenda, or more recently, Web pages and XML documents. Hopefully, those documents have a structure that tools can exploit in order to help the user.

For searching, tools such as class browsers or hypertext tables of contents, and `grep` or the search button of an editor, provide navigation and querying methods. However, these tools suffer an important limitation in that they cannot be combined with each other to make search more efficient. For instance, a text file can be `grep`'ed for a searched string or navigated using a table of contents. However, there is no way to combine the `grep` program and the table of contents program so that one can get the smallest subset of the table of contents that covers the result of a `grep`.

For understanding a document, a commonly accepted practice is to build incomplete but simpler *views* of the document. A popular family of views is obtained by seeing the document at different depths. For instance, a table of contents offers a superficial view, but helps in understanding a document by giving a bird-eye's view on it. At a given depth, many views can also be defined. For instance, showing only the specifications of a program, hiding the debugging code, the comments, or showing only the functions sharing a given variable are possible views on a program file. Each such view helps in understanding one aspect of a program, and also helps in focusing on one task as the user is not visually bothered by irrelevant details. However, usual tools, such as a class browser or tools that support literate programming, provide only a few of these views, whereas all these views are conceptually simple to describe. The user's ability to express what a view should show and hide is often very limited.

For updating a document, views are also helpful. Indeed, an appropriate view can bring together related parts of a document that are distant in the original. For instance, gathering all conclusions of a book may help in updating them coherently. However, tools supporting views seldom support view update because it causes coherence problems between views, and between views and the original document.

The problem with all those tools is that they lack of shared general principles that would make it possible to incorporate new tools, supporting new kinds of navigation, query, views and updates, and that would make it possible to combine them fruitfully.

We can draw a parallel between the management of file contents and the management of file directories. Directories offer one rigid classification of files, in the same way as files offer one rigid organization of data. The possibility to associate several *properties* to a file and then to combine navigation and querying in *virtual directories* has been proposed in the past to help in the management of file directories [2, 3]. We propose in this article to associate several properties to *parts* of a file, and to consider views as *virtual files* built of selected parts to help in the management of file contents.

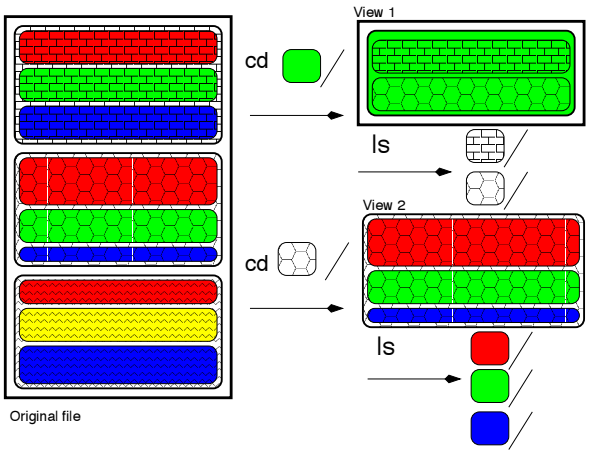

Figure 1: Symbolic representation of a file content

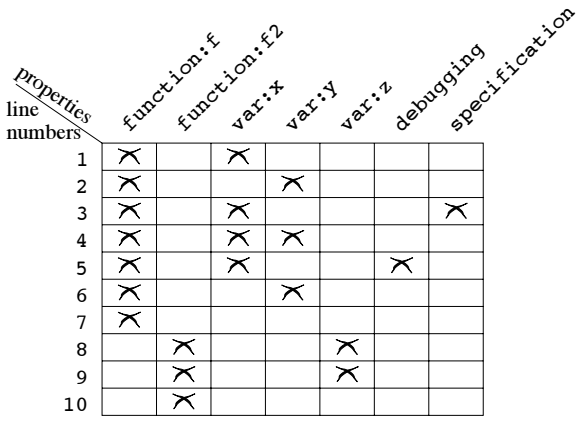

Figure 2: A file context

Figure 1 shows an iconic representation of the problem. The patterns represent the rigid structure of a file, and the grey levels represent different concerns. For instance, different patterns could correspond to different functions in a C program, and grey levels could correspond to heading, local declaration, body, or comments. As usual, concerns are scattered across structural boundaries. Hierarchical navigation must respect these boundaries. However, property-based navigation can focus on a concern across boundaries. Note that brick-pattern and dark-grey form an example of two properties that overlap. So, updating the brick-pattern view may also concern the dark-grey view.

Views must be first-class citizens, and they must be updated without restriction. We propose this new service at the file system level so that its impact is maximum without rewriting applications. So, several applications or users, with different requirements, can read and write the same file under their own visions. This new file system is called *Parts-of-file File System* (PofFS). Since criteria for defining a view are application specific, we propose that the file system operations only offer the generic background mechanisms, and that plug-ins could be defined to handle the application-level details.

2 Principles

We present PofFS as a shell-level demo because this is the simplest textual interface to a file system. However, more modern graphical interfaces like file browsers can also be used. We will say more on this subject at the end of the demo.

Consider a C program file, `foo.c`.

```
[1] % cat -n foo.c
 1  int f(int x) {
 2  int y;
 3  assert(x > 1);
 4  y = x;
 5  fprintf(stderr, "x = %d", x);
 6  return y * 2
 7  }
 8  int f2(int z) {
 9  return z * 4
10  }
[2] % poffsmount foo.c /poffs
```

Command 1 displays the content of `foo.c`, and command 2 mounts `foo.c` on `/poffs`, using *transducers* for attaching properties to parts of file `foo.c`. The transducers are selected automatically using a mechanism similar to MIME types. This makes it easier to manage combinations of transducers without forgetting one. Properties are "this line belongs to the definition of f" (`function`:f), "this line mentions variable x" (`var`:x), "this line contains a trace instruction" (`debugging`), and "this line is an assertion" (`specification`). The attachment of properties to lines can be represented as a *lines* $\times$ *properties* matrix which forms the *file context* (see Figure 2). In real-life, the *lines* $\times$ *properties* matrix can be very large, e.g., 100 000$\times$10 000, but it is also very sparse, e.g., an average of 10 properties per line. Note that indexing is not local to parts. For instance, line 7 has property `function:f` because a declaration of function `f` has been found 6 lines above. A change at line 1 may affect properties of lines 2 to 7.

```
[3] % cd /poffs
[4] % ls
foo.c debugging/ specification/
function:f/ function:f2/ var:x/ var:y/ var:z/
```

Command 4 has two effects. First, it creates a *view* that contains all the parts of the file that correspond to this directory. As this directory is the mount point, the view has the same content as the original file. Second, it computes possible *refinements* to the current directory, and presents them to the user as sub-directories (`function:f/`, `debugging/`, ...).

```
[5] % cd function:f
```

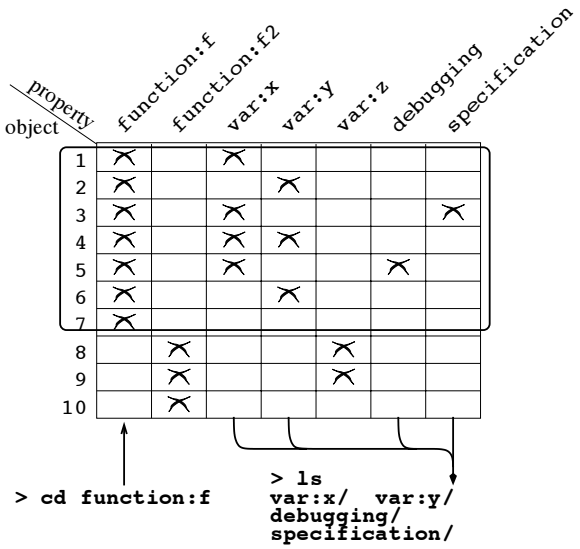

Figure 3: Navigation in a file context

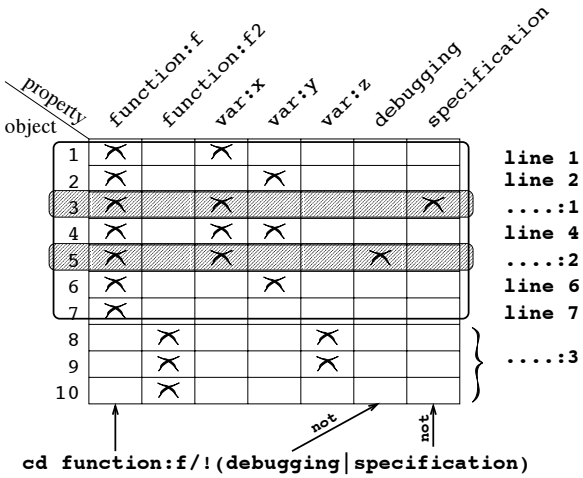

Figure 4: Creation of a view

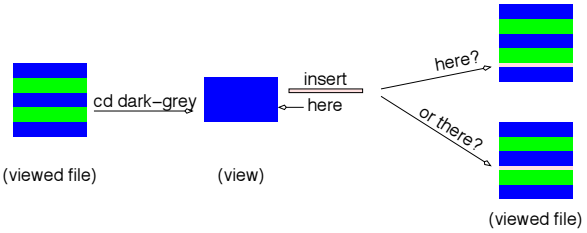

Figure 5: Why do we need marks of absence ?

```
[6] % ls
foo.c debugging/ specification/ var:x/ var:y/
```

Command 5 *refines* the current view by selecting parts of file that have property `function:f`. Command 6 shows how refinements are related to the current query. In directory `poffs/function:f`, property `var:z/` is no longer a refinement. This can be checked in the current view which contains only the code of function `f`, and contains no line related to variable `z` (see Figure 3). Moreover, `function:f/` is no longer a refinement, since it yields exactly the same view as the current one.

```
[7] % cd !(debugging|specification)
```

Command 7 illustrates the possibilities of the querying language. Negation is written `!`, and disjunction is written `|`. Character *slash* can be read as a conjunction. More sophisticated logics than propositional logic can be used by plugging *logic solvers* in the file system. For instance, program functions can be indexed by their types, and the types be compared using a type logic implemented as a pluggable module. So, *valued attributes* can be compared and filtered: e.g., `cd "type:?bool"` selects all functions with a boolean parameter. Similarly, `cd "function:^f.*"` selects all functions whose name starts with an 'f'. In this case a logic of strings (regexp) is used. PofFS also offers mechanisms for grouping resembling refinements; this reduces the size of answers to `ls`. For instance, sub-directories `function:f/` and `function:f2/` can be grouped in a directory `function:/`. Properties can also be grouped by the user in taxonomies, thus permitting to focus on a subset of the properties and making navigation easier.

```
[8] % ls
foo.c var:x/ var:y/
[9] % cat foo.c
int f(int x) {
int y;
.................:1
y = x;
.................:2
return y * 2
}
.................:3
```

Command 8 shows a list of sub-directories reduced to `var:x/` and `var:y/` (check on Figure 4). Command 9 displays the content of the current view. Groups of lines that do not satisfy the current query are replaced by *marks of absence*, e.g., `.....:1`. These marks will make it possible to back-propagate updates to the original file. Figure 5 illustrates the ambiguity of inserting a new piece in a file where missing parts are not marked.

```
[10] % cat foo.c | sed -e s/y/z > foo.c
```

Command 10 demonstrates that views can be *updated*, and can be so by any kind of tool. The effect of this command is to replace all occurrences of `y` by `z` *only in parts that belong to the current view*.

```
[11] % ls
foo.c var:x/ var:z/
```

Command 11 shows that updating a view affects property refinements (compare with results of command 8).

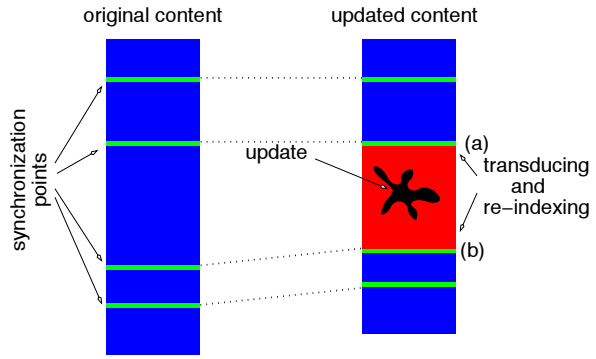

Figure 6: Re-indexing between synchronization points

```
[12] % pwd
/poffs/function:f/!(debugging|specification)/
[13] % cd /poffs
[14] % cat foo.c
int f(int x) {
int z;
assert(x > 1);
z = x;
fprintf(stderr, "x = %d", x);
return z * 2
}
inf f2(int z) {
return z * 4
}
```

Finally, command `14` shows that updating a view affects the other views. PofFS maintains the coherence of all views by back-propagating view updates to the original file, and then to the other views. In the worst case, this could cause a complete re-indexing of the original file. However, it is often the case that a file can be split in several parts that do not depend on each other from the point of view of the properties attached by a transducer. We call *synchronization points* the boundaries of these independent parts. A sophisticated algorithm makes it possible to limit re-indexing to as few independent parts as possible (see Figure 6).

The shell interface must not be taken for the file system. What PofFS actually offers is an implementation of operations `open`, `readdir`, `read`, `write`, etc. This makes it possible to adapt any existing interface that use these operations. For instance, PofFS on-line demo uses an unmodified web file-browser. However, it is sometime better to devise a specialized interface. This is what we have done for an HTML agenda in which properties allow one to select different kinds of events. The web server always displays the current view of selected events simply because a URL contains a PofFS path, and sub-directories are sorted according to their type (date, type of event, ...) and listed in menus.

3 Discussion and conclusion

The service of manipulating file contents is already offered by several kinds of application: e.g., text editors like Emacs, CIA [1], and IDEs (*Integrated Development Environments*) like Eclipse. These applications offer means for querying and navigation, and they also allow to hide parts of file. The novelty of PofFS is to combine fruitfully query, navigation and view update at system level. We have used it in applications like text edition and programming, and also in trace and log analysis. In all cases, performances are encouraging.

The management of file contents at the system level has not evolved much since the first systems with stream files; files are units, querying and navigating work at the file level. Only read and write go inside files. Considering files as flat streams have been a fruitful abstraction to permit the combination of tools *via* pipes, redirection, etc. We have proposed to consider files as possible mount-points, to navigate in them, to extract views, and to update them. This raises the consistent view update problem between the views and the mounted file. We have proposed a mechanism of updatable views that solves this problem efficiently. The *Parts-of-file File System* (PofFS) makes the structure of files virtual, and less tightly related to the physical model of a stream of characters. What PofFS does is to recover structure in files and still permit the fruitful combination of tools.

Our current PofFS prototype handles parts as sets of lines; further works will be to handle parts as sets of character positions. We have only used PofFS with text files. However, nothing in principle prevents from using PofFS with binary files (executable or multimedia) as long as a transducer exists. More experiments are required to assess the feasibility of using PofFS with binary files, in particular with respect to real-time constraints of multimedia usage.

A prototype PofFS and more information on this project can be down-loaded at the following URL:

`http://www.irisa.fr/LIS/PofFS/`

This page makes also accessible a companion paper completing this article by describing the algorithms, benchmarks, and extensions to PofFS. It presents also more precisly the principles and semantic of PofFS, and gives a more complete account on the related works.

References

[1] Y.-F. Chen, M. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction system. *IEEE Transactions on Software Engineering*, 1990.

[2] D.K. Gifford, P. Jouvelot, M.A. Sheldon, and J.W. O'Toole Jr. Semantic file systems. In *ACM Symp. Operating Systems Principles*, 1991.

[3] Y. Padioleau and O. Ridoux. A Logic File System. In *USENIX Annual Technical Conference*, 2003.