Vol.2011-ARC-194 No.10
2011/3/11

セマンティックファイルシステムのフレームワークの提案

宮元裕樹[†1]　酒居敬一[†1]

計算機で利用するアプリケーションの増加や新しいデバイスの登場とともに扱うファイル数が増加している．それに伴って計算機を利用する人がファイルを目的や用途などに応じて分類する手間が爆発的に増加している．しかし，現状の静的な木構造による分類では手間を軽減することは困難である．そこでファイルが持つ意味情報を整理し木構造へとマッピングするセマンティックファイルシステム (SFS) が提案されている．しかし，Linux 系ディストリビューションや Windows 等の OS が SFS を採用した例は少ない．SFS が採用される例が少ない原因のひとつに SFS を有効に機能させるために必要不可欠な整理規則や抽出可能なファイルの種類の追加変更が難しいことに問題があると考えた．本稿では，その問題を改善するためにスクリプト言語 Ruby と FUSE を用いて SFS の機能を容易に追加変更できるフレームワークを提案した．そして実装したファイルシステムが実用的であることを，処理時間および拡張性の 2 点から評価した．

A New Approach of Framework for Semantic Filesystem

Hiroki MIYAMOTO [†1] and Keiichi SAKAI[†1]

The number of files installed on the computer have been increasing according to increase of the application and appearance of new device. The file increase raise an increased cost that the user classify the files by their purposes, usages, *etc.* However, it is difficult to reduce time in the classification by a static conventional tree structure. Therefore, the semantic file system (SFS) have been proposed, it adopts the semantic information extracted from the file to classify the file. But there is no implementation of the semantic files system on the conventional OS such as Linux based distribution, Windows, *etc.* We have pointed out major reason for no implementation, that is difficult to update the classification rules and extract the necessary semantic information among the files. In this report, a script language Ruby and an interface library FUSE are used to solve the problem and implement the function of SFS. We propose the framework that make semantic information change scheme easy. Moreover, we have evaluated the framework that SFS implemented is useful from two view points (the processing time and the extensibility).

1. はじめに

コンピュータで利用するアプリケーションの増加やコンピュータに接続する新しいデバイスの登場とともに扱うファイル数が増加し続けている．それらのファイルがアプリケーションやデバイス固有のものであればよいが，コンピュータを利用する人がファイルをアプリケーション横断的に目的や用途などに応じて分類，整理しようとすると，その手間が爆発的に増加してしまう．しかしながら，現状のファイルシステムではそういった手間を軽減することが難しい．その理由として，現在のファイルシステムではファイルを整理する仕組みとして使われるディレクトリが木構造にほぼ限定されており，その構造が静的であることが挙げられる．もちろん，ディレクトリのハードリンクは原理的に実装できるが，木構造になるようにそういう操作を禁止している．このため，ファイルは利用者が定めたひとつの整理ポリシに則った静的なディレクトリ構造を作って，ファイルの整理をしなければならず，その整理ポリシに束縛され多様なファイル整理できない．また，ファイルへのシンボリックリンクもしくはハードリンクを用いて整理ポリシを柔軟にする方法は存在するが，必要なファイル数だけ利用者がリンクする必要があり，整理する手間の軽減になっていない．

現在のファイルシステムでは，その実装の中にファイル名，作成年月日，更新年月日，最終アクセス年月日，所有者，パーミッションといった意味情報を持たせるしくみがある[3)]．しかし通常ユーザがファイルの識別に利用するのは，そういった情報よりもむしろどのディレクトリに置かれ，どのファイル名であるかというふたつのパラメータである．一方で，ファイルにはアプリケーションが利用するデータをファイルの中に持つのが普通であり，それを抽出して意味情報として扱うことも考えられる．それら複数の意味情報を持つファイルでさえも，名前と場所というふたつのパラメータを除いた他の意味情報はファイルの整理に利用されることは少ない．これには，いわゆるファイラーの類のアプリケーションが，日々新しく考えられるアプリケーションごとに異なる整理ポリシを任意の組み合わせで利用することができず，ディレクトリの中では拡張子などの概念やマジックナンバを導入してある程度の整理はできるにせよ，やはり大域的にはファイルシステム由来の静的なディレクトリ構造によってのみ整理するしかないという現実がある．

†1 高知工科大学
Kochi University of Technology

他には現在の OS では，サービスプログラムとしてファイル整理の手間を軽減する方法を模索している[7]．しかし，サービスプログラムでは対応したアプリケーション以外からは利用が難しいという問題を抱えている．このため，アプリケーションの改変をすることなく全てのアプリケーションから利用可能にするためには，これまでのファイルシステムと同じようにアプリケーションから利用でき，ディレクトリ構造を持つ新しいファイルシステムが必要となる．

そこで，ファイル名以外の意味情報を利用し整理するセマンティックファイルシステムが提案されている[1)2)]．セマンティックファイルシステムでは，ファイルから抽出できる意味情報（作成者，関数名，章名など）を元に木構造を構築する．このため，ファイル名以外の視点からの整理できる．

しかし，セマンティックファイルシステムは一般的に利用されているオペレーティングシステムで採用された例は少ない．その理由として，現状の実装では，目的やファイルの種類にあわせてディレクトリ構造を拡張することが困難であることが考えられる．意味情報の抽出やディレクトリ構造へのマッピングのためには[1)] にあるように，Transducer が鍵となるツールであることがわかるが，日々発表されるアプリケーションやデバイスに関係するファイルの情報に合わせられるような万能なツールを作ることは困難である．そこで，本稿では，ユーザ空間でファイルシステムを実装する FUSE とスクリプト言語である Ruby を用いて，少ないコード量で容易に拡張可能なセマンティックファイルシステムのフレームワークを提案し，その実用性を拡張性と処理時間の両面から検証する．

2. コンピュータ上でのファイル管理方法

2.1 ファイルシステム

ファイルシステムでは，あるデータの集合に対して名前付けをしたものをファイルとして扱っている．ファイルは画像や音楽，テキスト等のデータやコンピュータプログラムを表現，管理するための論理的な最小単位になっている．この論理的な最小単位であるファイルを，ファイルシステムはハードディスクドライブやフラッシュメモリ等の記録媒体に記録し，必要に応じて読み出し，変更等を行っている．また近年のファイルシステムでは，ファイルの一貫性の保持や，ジャーナリングを行っている場合があり，必要に応じてエラー等の訂正もしている．

VFS では **図 1** のように ext2 や ext3，ReiserFS といったファイルシステムの違いや，アクセスするデータがローカルディスク上にあるか Network File System(NFS) のようにネットワーク上にあるかといったファイルシステムによって異なる部分をアプリケーションプログラムから隠蔽し，抽象化して扱えるように作られている．Linux の実装では，アプリケーションからのファイル操作に関するシステムコールは VFS が受け取り，それを各ファイルシステムの実装に渡している．たとえば，VFS 登場以前のセマンティックファイルシステムでは NFS の仕組みを利用し実装する例が見られたが，VFS 登場以降は VFS のインタフェースに合わせて設計ができるようになったため，NFS の仕組みを利用する必要がなくなっている．

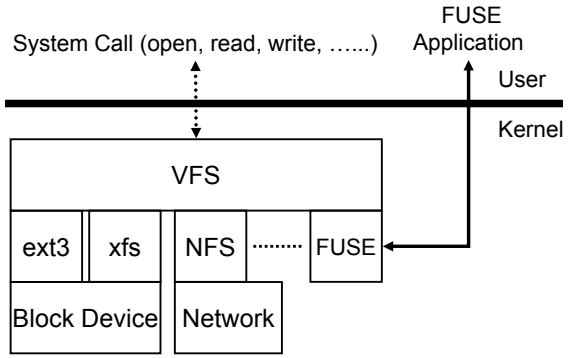

図 1 VFS と FUSE の概念図
Fig. 1 Relationship between VFS and FUSE.

FUSE は VFS の仕組みを利用したカーネルモジュールの一種で，Linux では Kernel 2.6.14 から正式に取り込まれたものである．FUSE では， 図 1 に示すようにアプリケーションから送られてきたシステムコールを VFS を通してユーザ空間上で実行されている FUSE アプリケーションにリダイレクトする仕組みである．これにより，必ずしもファイルシステムがカーネル空間上で実装される必要がなくなり，たとえば WikipediaFS[4)] や NTFS-3G[5)] といったユーザ空間上で実行されるファイルシステムが実現されている．

現状の Linux 系ディストリビューションや FreeBSD などの Unix 系 OS，Windows などでは，ファイルシステム上の木構造ディレクトリでファイルを整理し，アプリケーションから扱っている．このディレクトリ構造は，ext3 や FAT，NTFS といった異なるファイルシステムでも実装方法は異なるが似た構造となっている．

しかしどのファイルシステムにおいてもディレクトリ構造は静的で，たとえば autofs でさえも木を接続・分離するだけなので，仮にファイルに複数の意味を保持させたとしても，

表 1 各種ファイルが持つ属性例
Table 1 Example for Attribute of Several Types of File.

ファイル	属性
C 言語ファイル	作成者
	関数名
Ruby ファイル	作成者
	クラス名
	メソッド名
音楽ファイル	作曲者
	ジャンル
	アルバム名

それらを任意の木構造に随時マッピングすることは困難である．ファイルを移動するにせよリンクを張るにせよファイル数が多くなればそういったマッピングは困難である．そこで，ファイルから抽出した意味情報を利用し整理するセマンティックファイルシステムが提案されている．セマンティックファイルシステムでは，ファイルから抽出可能な意味情報を動的に木構造にマッピングすることで，ファイル名以外の視点から整理できる．

2.2 セマンティックファイルシステム

セマンティックファイルシステムでは，ファイルをファイル名とパス情報以外の方法で整理し木構造ディレクトリとして動的にマッピングをすることができる．そのためにファイルが持つ意味情報を属性として抽出し，抽出された属性を整理ポリシに基づいて整理することによって実現している．このため現在利用されている NTFS や ext3 といったファイルシステムとは異なりファイルから抽出する属性や整理ポリシを追加，変更することによって様々な整理を随時提供することができる．**表 1** では各種ファイルからどのような属性が抽出可能かを示している．

例えば，Javadoc や Doxygen の書式に則ってコメントを埋め込んだソースコードファイルでは，ファイル内に存在する関数名や作成者などを属性として抽出できる．また，構造体名や変数名などを属性として追加したい場合には，ファイルから属性情報を抽出し整理ポリシの作成をすれば新たに SFS 上で扱うこともできる．

このようにセマンティックファイルシステムでは，ファイルの意味情報と整理ポリシからさまざまな整理方法を提供でき，ディレクトリの木構造として動的にマッピングできる．しかし，セマンティックファイルシステムは一般的に利用されている OS で採用された例は少ない．その理由として SFS を有効に機能させるために必要不可欠な整理ポリシや抽出可能なファイルの種類の追加変更といったことが難しいことに原因があると考えた．しかし，整理のための意味情報はアプリケーションの種類に相当する数が存在することから，万能な抽出ツールを構成することは難しい．もちろん，データだけであればアプリケーションに付属するツールなどで取り出すことはできるが，その出力をセマンティックファイルシステムに取り込むにも少なくともツールの出力形式をセマンティックファイルシステムの内部の DB に格納できるように変形するツールが必要である．つまり，アプリケーションごとに抽出ツールが必要でさらに変換ツールも必要となり，この部分はセマンティックファイルシステム本体から分離して追加しやすく実装すればよいということになる．

そこで，本稿では新しい実装によるフレームワークを 第 3 節から改善案として提案する．

2.3 スクリプト言語によるファイルシステムの実装

これまで，ファイルシステムはカーネルやデーモンとして C 言語で実装されてきた．これには，CPU やメモリといったコンピューティングリソースの問題やこれまで記述してきたソースコードの活用，依存するライブラリの最小化，ファイルシステムを提供する API の各プログラミング言語への対応状況，ファイルシステムを実装するためのプログラム空間等が背景にあると考えられる．もちろん，整理ポリシの生成のための抽出と変換ツールについても従来ならば C 言語で書きコンパイルして，できたオブジェクトファイルをセマンティックファイルシステム本体に静的か動的にロードするということになる．

しかし，現在では上記に述べた背景のうちコンピューティングリソースの問題と，ファイルシステムを提供する API，ファイルシステムを実装するためのプログラム空間は既に解消されつつあると考えている．例えばコンピューティングリソースでは，10 年前と比較して CPU は周波数の向上やコア数の増加によって高機能化されており，メモリに関しても 10 倍以上の容量を搭載することが容易になっている．また，API に関しても本研究で利用した FUSE（Filesystem in Userspace）を活用することにより様々なプログラミング言語を用いてファイルシステムをユーザ空間で実現できるようになっている．

一方で近年では Web アプリケーションのように，より簡潔にプログラムを記述できることが求められるようになっているため，Ruby や Python，Java と言った C 言語よりも抽象度が高く簡潔に記述出来るプログラミング言語が広く使われるようになっている[6]．たとえば Tomcat による JSP 実装では，Java をスクリプトレットのように使うことで，Java の豊富なライブラリとあいまって頻雑な文字列処理と複雑なモデル処理を生産性高く記述できる一方で，ある程度速い処理もできる．

そこで本稿では C 言語よりも抽象度が高く Java より簡潔に記述することが出来るプログラミング言語を用いて，最小限の手間で拡張可能なファイルシステムのフレームワークを，

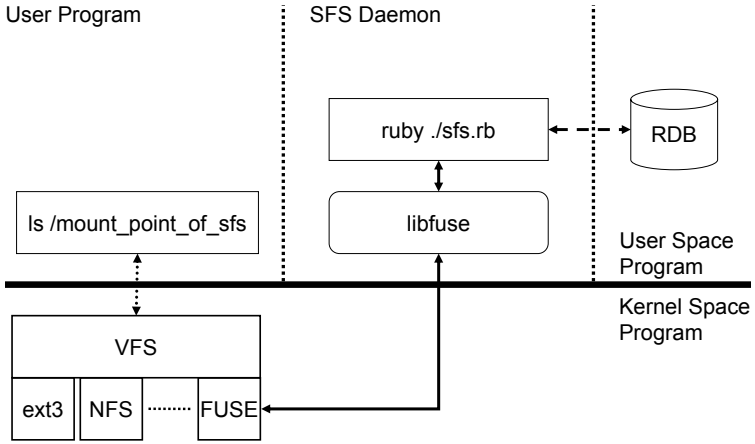

図 2 システム全体図
Fig. 2 System Outline.

高い拡張性を求められるセマンティックファイルシステムとして提案する．

3. フレームワークの概要

本稿で提案するフレームワークを **図 2** に示す．通常，プログラムはカーネル空間とユーザ空間の二つの空間に分かれて存在している．本フレームワークを実現するために実装したプログラムはすべて図 2 のようにユーザ空間で動作している．そして，ユーザプログラムに対してファイルシステムを提供するための API は全て FUSE を通じて提供されるため，カーネルをはじめとする OS のソースコードには一切変更していない．

つぎに，SFS Daemon のブロック図を **図 3** に示す．SFS Daemon は主に SFS Daemon 部と Transducer 部の 2 つの機能に分割されている．一つ目の SFS Daemon 部では，FUSE インタフェースを通じてファイル操作に関わるシステムコールの処理やオリジナルファイルが更新された際に発生する通知を受け取る．二つ目の Transducer 部ではファイルから意味情報の抽出や，抽出された意味情報をセマンティックファイルシステムが利用するデータベースに対して追加変更削除をしている．このように，本フレームワークでは FUSE を用いることで OS を変更することなく，単純なインタフェースを通じてアプリケーションから送られてくるシステムコールを処理している．

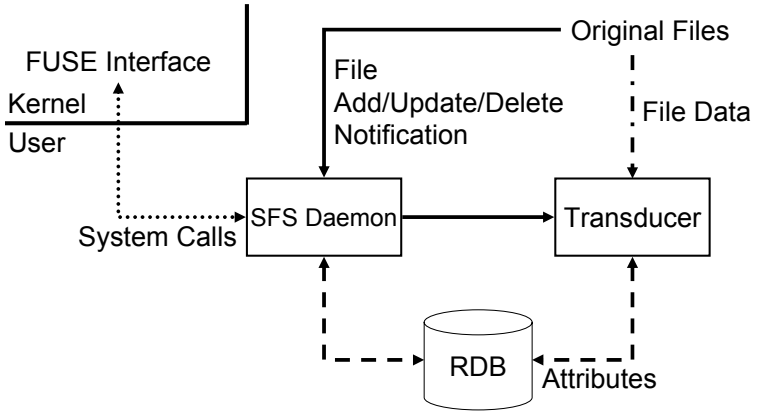

図 3 SFS Daemon のブロック図
Fig. 3 Block Diagram of SFS Daemon.

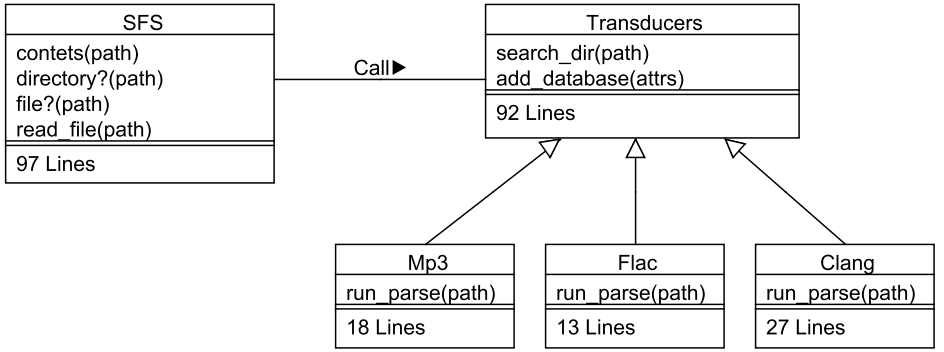

図 4 SFS フレームワークのクラス図
Fig. 4 Class Diagram of SFS Framework.

さらに，実装したプログラムのクラス図を **図 4** に示す．本フレームワークでは，これまでに述べたように SFS Daemon 部と Transducer 部の二つに分かれておりクラスも同様に SFS と Transducers に分かれている．そして，Transducers を継承しファイルの種類ごとに新たなクラスを作成することによって新しい種類のファイルに対応でき，それらのファイルは最小限の行数で記述できる．

Vol.2011-ARC-194 No.10
2011/3/11

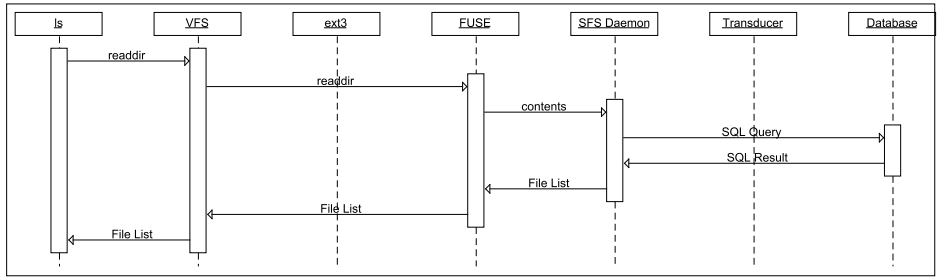

図 5 ls コマンド実行時のシーケンス図
Fig. 5 Sequence Diagram while ls is Running.

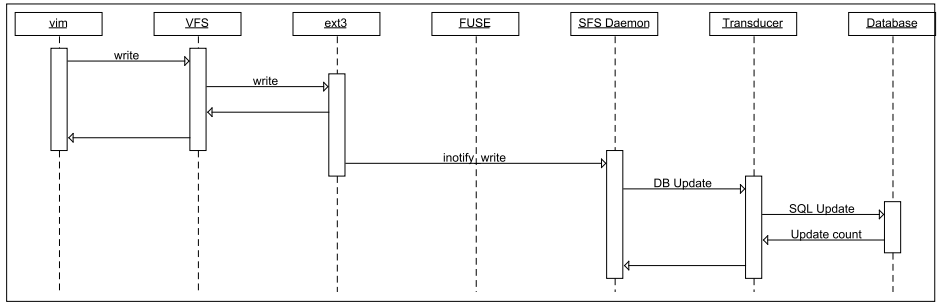

図 6 vim コマンドでファイル書き込み時のシーケンス図
Fig. 6 Sequence Diagram while vim is Writing the File.

3.1 動作シーケンス

本フレームワークでは， 3 節で述べたように，SFS Daemon 部と Transducer 部に分割されている．そのため，各部とそれ以外のソフトウェアが協調して動作する必要がある．本節では，ディレクトリの参照をする場合およびファイルに対して書き込みをする場合という二つの異なる動作をするコマンドを例に挙げ， **図 5** および 6 に動作シーケンスを示す．

まず，ls コマンドを例にした 図 5 では，ls コマンドのようなコマンドはファイルに対して書き込みをしないためオリジナルファイルに変更を加えることはない．このために，VFS が受け取ったシステムコールは FUSE を経由し SFS Daemon にリダイレクトされ，SFS Daemon はパス名をもとにデータベースに対してクエリを送信し，その結果を ls コマンドに返すようになっている．

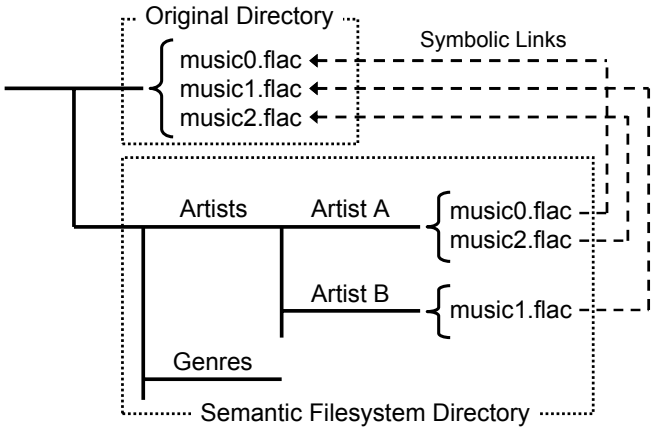

図 7 ディレクトリ構造
Fig. 7 Structure of Directory.

つぎに，vim コマンドを例にした **図 6** では，vim コマンドのようなコマンドはファイルに対して書き込みをするためオリジナルファイルに変更を加える場合がある．この例では，図 6 のようにファイルに対して書き込むと ext3 上に存在するオリジナルファイルが変更される．その後カーネルによりさきほどの変更に伴って inotify が SFS Daemon に通知され，SFS Daemon は変更されたファイルの意味情報を Transducer 部を利用し更新する．

3.2 ディレクトリ構造

ディレクトリ構造を **図 7** に示す．図 7 では，3 つの音楽ファイルを例に上げセマンティックファイルシステムの整理対象のディレクトリを Original Directory とし，整理後のディレクトリを Semantic Filesystem Directory としている．これらは別のファイルシステムとしてマウントする．セマンティックファイルシステムでは，Original Directory に存在する 3 つの音楽ファイルから属性を抽出し，図 7 に図示されている Artists，Genres のように Semantic Filesystem Directory 下に属性順に整理している．Semantic Filesystem Directory 下のディレクトリは動的に生成した仮想的な存在であるが，Semantic Filesystem Directory 下に存在するファイルは全て 図 7 の破線が示すファイルへのシンボリックリンクとなっている．つまりファイルはシンボリックリンクを仮想的に生成しているのみで，実体は Original Directory 下にのみ存在するようにして，本来のファイルと区別している．

表 2　システムコールのマッピングテーブル
Table 2　System Call Mapping.

システムコール	FUSE API	Ruby Binding
open	open	read_file
read	read	↑
write	write	write_to
readdir	readdir	directory?
utime	utime	touch
mkdir	mkdir	mkdir
rmdir	rmdir	rmdir
rename	rename	delete, write_to

3.3　マッピング

セマンティックファイルシステムでは，NTFS や ext3 等のファイルシステムとは異なる振る舞いをする場合がある．そのため既存のオペレーティングシステム上でセマンティックファイルシステムを動作させる場合に，コマンドやシステムコールのマッピングをフレームワーク内で行う必要がある．順に各レイヤーでのマッピングについて述べる．

システムコール

FUSE では，ユーザプログラムからカーネルに対しての `open`，`read`，`write` といったシステムコールを VFS 経由しユーザ空間で動作するプログラムに対してリダイレクトしている．これには，FUSE のプログラムを実行する際に，`fuse_operations` 構造体とよばれる構造体を FUSE に対して受け渡すことによってシステムコールが発行された時に呼び出される関数を設定している．また，`fuse_operations` では，`open`，`read`，`write` といったシステムコールと一対一で対応付けしている．

さらに，Ruby で FUSE からの呼出しを処理するためのバインディングが FUSE と Ruby のコード間に存在している．この為，各システムコールと Ruby の各メソッドの対応付けをする必要がある．本研究では，Ruby の FUSE バインディングに FuseFS を用いているため **表 2** のように対応付けしている．

例えば，`open` システムコールでは，表 2 から分かるように FUSE を通じて Ruby の FUSE バインドの `open` 関数に相当する関数を呼び出した後に，Ruby で書かれた `read_file` メソッドが呼び出されるようになっている．そのため Ruby のコードでは `open` と `read` システムコールの機能的な違いはなくなっている．

ユーザコマンド

本フレームワークではユーザコマンドを 2 種類に分類をしている．最初の整理は，セマンティックファイルシステム上に存在するファイル本体に変更を加えるようなユーザコマンドをファイルエディットコマンドと定義する．ファイルエディットコマンドではファイルの内容を変更するがファイルどおしあるいはファイルとディレクトリとの関係を変更しないようなコマンドのことを指し，例としては vim や Emacs 等が挙げられる．2 つ目の分類は，リレーションエディットコマンドと定義する．リレーションエディットコマンドは，ファイルを変更しないが，そのファイルが持つ意味付け等を変更するコマンドのことを指し，例としては cp や link，mv 等のユーザコマンドが対象となる．

本フレームワークでは，ファイルエディットコマンドの動作は変更していない．しかし，リレーションエディットコマンドに制約を設けている．なぜならば，3.3 節及び 3.4 節で述べるパス名の制約およびコンシステンシ問題によって制約を受けるからである．このため，リレーションエディトコマンドのようなセマンティックファイルシステムに対して何らかの変更を加えるようなコマンドはエラーとなる．

パスシンタックス

セマンティックファイルシステムでは，ファイルを検索するために検索対象の属性と値を入力が必要となる．しかし既存のファイルシステムでは，ファイルにアクセスする手段としてパスを利用する方法しか存在しない．その為，既存のオペレーティングシステム上でセマンティックファイルシステムを利用するには属性と値をパス名を利用して指定できるような仕組が必要となる．そこで本稿では，パス名に対して一定の条件を設定することによって，フレームワーク内で属性と値の判別を可能にした．そして，利用するパス名のシンタックスは **図 8** の用に定めている．

パス名の制約

Unix 系 OS では，ファイル名として使用できない文字は'/' および'\0' のみであるため，利用出来る文字は Windows 等と比べ多い．しかし，習慣上用いない方が良い文字が存在する．例えば，'<'，'>'，'"'，'*' 等である．これらの文字はシェル上等で他の意味付けをされて利用されており，ファイル名として用いない方が入力の際に手間が軽減され，さらには誤動作をする危険性を軽減できるためである．

一方でファイルから意味情報を抽出した場合，先に述べたようにパス名に含めることが出来ないもしくは用いない方が良い文字が含まれている場合がある．この場合本フレームワークでは全て表示の際に'_' で置換している．このため，実際に抽出された意味情報とアプリケーション側から見たファイル名が一致しない場合がある．

```
<sfs-path>    ::= /<pn> | <pn>
<pn>          ::= <name> | <attribute>
                  <field-name> | <name>/<pn>
                  <attribute>/<pn>
<attribute>   ::= field: | <field-name>/<value>
<field-name>  ::= <string>:
<value>       ::= <string>
<name>        ::= <string>
```

図 8 パスシンタックス
Fig. 8 Path Syntax.

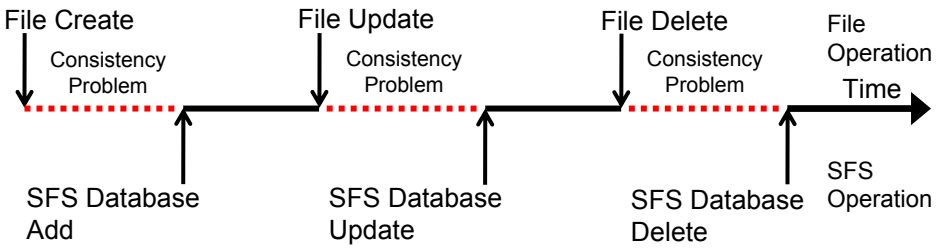

図 9 コンシステンシ問題の概念図
Fig. 9 Consistency Problem.

3.4 コンシステンシ

ファイルシステムにおいてコンシステンシは非常に重要な問題である．例えば，**図 9** が示すように，ユーザや他のアプリケーションによってファイルが追加，変更，削除された場合を想定する．この場合，それぞれの場合おいて早急にセマンティックファイルシステムのデータベースに対して意味情報を追加，変更，削除をしなければコンシステンシの問題が発生し，そのファイルを利用した場合，存在すべきファイルが存在しない，異なる意味情報が表示される，存在しないファイルが存在する等の問題が発生しえる．このためセマンティックファイルシステムでは常にファイルを監視し，これらの変更に対応する必要がある．

このため本研究では，Linux カーネル 2.6.13 から利用可能な inotify と呼ばれる仕組みを利用し，ファイルを監視している．これによって，オリジナルのディレクトリから追加，変更，削除された場合は OS 側から inotify を通じて SFS Daemon に通知されるのを利用しセマンティックファイルシステムのデータベースの追加，更新，削除をしている．

表 3 評価環境
Table 3 Evaluative Environment.

CPU	Intel Core 2 Duo E6600(2.4GHz)
Host Memory	Dual-Channel DDR2-800 4GB
Guest Memory	1GB
Virtual Machine	VMware Player 3.1.3
Host OS	Windows 7 Professional(64bit)
Guest OS	Fedora Linux 14(32bit)
RDBMS	SQLite 3.6.23.1

表 4 ベンチマーク 10 ファイル
Table 4 Benchmark of 10 Files.

Filesystem	second/10k count	second/count
ext3	40.49	$4.049 * 10^{-3}$
SFS	52.26	$5.226 * 10^{-3}$

表 5 ベンチマーク 1920 ファイル
Table 5 Benchmark of 1920 Files.

Filesystem	second/10k count	second / count
ext3	404.77	$4.047 * 10^{-2}$
SFS	614.64	$6.146 * 10^{-2}$

4. 評　　価

本節では，Ruby，FUSE，SQLite を用いて実装したセマンティックファイルシステムの評価について述べる．評価環境を**表 3** に示す．本稿では，Windows7 をホスト OS とし，仮想マシン上に Linux 環境を構築し評価した．

4.1 処 理 時 間

まずは，処理時間が実用的であるかを評価するために，**表 4** 及び **表 5** に測定結果を示す．

この測定では，測定対象のディレクトリを ext3 及びセマンティックファイルシステム上に用意し，10 又は 1920 ファイルの音楽データを配置している．タイムスライスが 1[ms] であるため，測定時間の精度を稼ぐ目的およびディスクキャッシュが十分効いていることを前提にするために，結果は ls コマンドを音楽ファイルを配置したディレクトリに対して 1 万回実行をするのを 3 回繰り返し平均している．

表 6 属性抽出用ソースコード行数
Table 6 Number of Lines for Attribute Extracting Code.

Extensions	Lines
.mp3	18
.flac	13
.c	27

表 4 及び 表 5 から分かるように，ext3 上に直接ファイルを配置した場合と比べ，セマンティックファイルシステム上にファイルを配置した場合でも 1.2〜1.5 倍程度のオーバーヘッドで収まっていることが分かる．また，1 コマンドあたりの実行速度に関しても実行時間の差が $1.177 * 10^{-3}$ 〜 $2.099 * 10^{-2}$ 秒程度となるため実用的な範囲での性能劣化といえる．

4.2 拡 張 性

つぎに，ファイルから属性を抽出する際に，セマンティックファイルシステムではファイルの種類ごとに抽出用のプログラムが必要となる．そのため，より少ないソースコード量で多彩なファイルを対応できることが望ましい．本研究では，ファイル属性の抽出部と実際にデータベースへ追加する部分を分離している．ファイル属性の抽出部の行数をコメントや改行のみの行も含めた形で **表 6** に示す． 表 6 から分かるように 30 行程度で実装できることが分かる．ファイル属性の抽出部では，既存のライブラリや外部コマンドを活用することを優先し実装しため，このような少ないコード量で実装可能という結果になったと考えられる．

しかし，既存のライブラリや外部コマンドが存在しない場合はユーザがファイルから属性を抽出するためのプログラムを全て記述しなければならず本稿の評価結果よりも多くのコードを記述する必要があると考えられる．ただし，Ruby を含むスクリプト言語を用いた場合，C 言語と比べ抽象度が高いため既存ライブラリや外部コマンドが存在しない場合であっても実装規模は小さくなると考えられる．

このことから，ライブラリや外部コマンドが存在する場合は最小限の手間で拡張が出来ることを確認した．また，ライブラリや外部コマンドが存在しない場合でも抽象度が高いスクリプト言語では C 言語での実装に比べ比較的小さい実装規模になると考えられる．

5. おわりに

本研究では，まず，セマンティックファイルシステムが採用される例が少ない原因のひとつに整理ポリシや抽出可能なファイルの種類の追加変更が難しいことに問題があると考え，その問題を改善するためにスクリプト言語 Ruby と FUSE を用いてセマンティックファイルシステムの機能を容易に追加変更可能なフレームワークを提案した．そして，提案したフレームワークをスクリプト言語 Ruby および FUSE を用いてユーザー空間で実装し，その処理時間および拡張性を評価した．その結果，処理時間ではオーバヘッドが 1.2 倍〜1.5 倍程度と既存のファイルシステムである ext3 との差が少なく，ファイルシステム内に置かれるファイル数が 192 倍になったとしても処理時間の増加は 10 倍程度という評価結果を得られた．このことから実際にスクリプト言語 Ruby と FUSE を用いて実用的な処理時間でファイルシステムが動作することを確認した．また，拡張性では，スクリプト言語 Ruby が持つ既存のライブラリや外部コマンドを利用することによってセマンティックファイルシステムの拡張が容易であることが示された．

最後に今後の課題を挙げる．現在のフレームワークでは整理ポリシを Ruby のコードとして記述をしている．しかし整理ポリシは YAML や XML といったプログラミング言語ではない記述法を用いる方がより分かりやすく拡張可能と考える．そのため，YAML や XML といったフォーマットで記述可能な整理ポリシの実装が望まれる．さらに，整理ポリシ自体も現在のフレームワークでは意味情報を一覧にするのみとなっているためより高度な整理ポリシの提案が課題となっている．

参 考 文 献

1) David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O ’ Toole, Jr.: “Semantic File Systems”, ACM Operating Systems Review, pp. 16–25, Oct. 1991.
2) Bryan Mills: *Metadata Driven Filesystem* , `http://bryanmills.net:8086/uploads/metafs/bmills-final.pdf`, Sept. 15, 2010.
3) 高橋浩和，小田逸郎，山幡為佐久: *Linux カーネル 2.6 解読室*, ソフトバンククリエイティブ株式会社, p.253, 2006 年 12 月 25 日.
4) Mathieu Blondel: *WikipediaFS* , `http://wikipediafs.sourceforge.net/`, Jan. 12, 2011.
5) Tuxera Inc.: *NTFS-3G* , `http://www.tuxera.com/community/ntfs-3g-download/`, Jan. 12, 2011.
6) John K. Ousterhout: “Scripting: Higher Level Programming for the 21st Century”, IEEE Computer , Volume 31 Issue No.3, pp. 23–30.
7) The GNOME Project: *Tracker*, `http://projects.gnome.org/tracker/`, Jan. 12, 2011.