



HashFS

Un filesystem basato sull'hashing.

Damian Tosoni, Danilo Salvati

Sicurezza dei sistemi informatici e delle reti

Indice

Abstract.....	pag.3
Motivazioni.....	pag.3
Definizione del problema.....	pag.3
Approccio.....	pag.3
I principali problemi da risolvere.....	pag.4
L'idea.....	pag.5
Le operazioni.....	pag.6
 Tasks del progetto.....	pag.Y
Che cos'è.....	pag.1
Che cos'è.....	pag.1
Che cos'è.....	pag.1
Che cos'è.....	pag.1

Legenda:

NERO Lavoro svolto in comune

BLU Lavoro svolto da Damian Tosoni

VERDE Lavoro svolto da Danilo Salvati

Abstract

MOTIVAZIONI

La scelta di questa tesina è stata motivata da un particolare interesse verso la comprensione dei meccanismi di funzionamento di un Sistema Operativo (anche in relazione alla sicurezza che garantisce, aspetto sempre più rilevante nei sistemi moderni) e, più in generale, verso il mondo Unix.

DEFINIZIONE DEL PROBLEMA

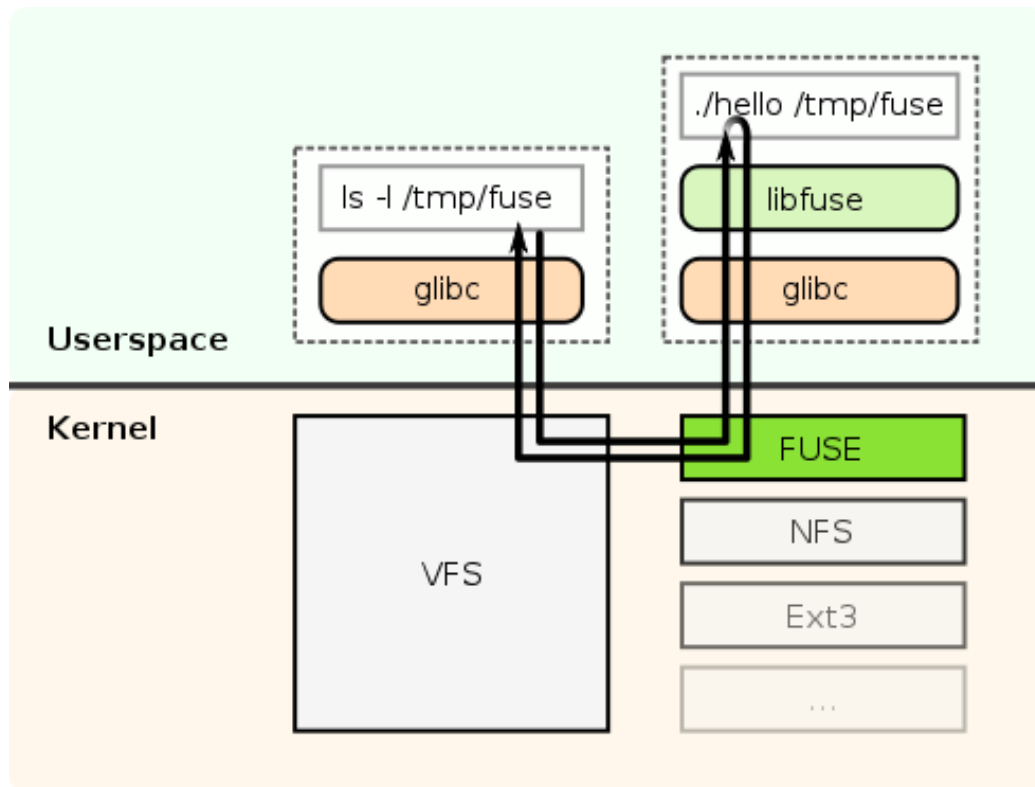
L'obiettivo principale del progetto è realizzare un filesystem che mantenga persistente e aggiornato, per ciascuna directory, un hash della directory stessa (calcolato in modo ricorsivo).

APPROCCIO

Per portare a termine questo obiettivo, ci siamo avvalsi dell'utilizzo di **FUSE**, un acronimo che sta a significare **F**ilesystem in **u**ser**s**pace. È un progetto open source il cui scopo principale è quello di realizzare un *modulo per il kernel Linux* che permetta agli utenti non privilegiati di un sistema di *creare un proprio file system senza la necessità di scrivere codice a livello kernel*. In pratica, quindi, si interpone tra l'utente ed il kernel fungendo da ponte e permettendo all'utente di effettuare richieste al kernel usando però un linguaggio di programmazione più user-friendly, che può essere scelto tra tutti quelli supportati (oltre ai classici C e C++, troviamo Java, Python, Perl, Ruby, OCaml, ...). Per fare ciò, il codice del filesystem è eseguito in User Space.

In dettaglio, i principi di funzionamento sono i seguenti:

- a livello utente, la componente che si occupa di gestire il filesystem FUSE si comporta come un demone che riceve richieste dal kernel attraverso uno speciale device;
- a livello kernel, il modulo si occupa di trasformare le richieste per il filesystem FUSE in richieste al demone.



Un filesystem FUSE è un sistema composto da tre elementi:

1. un modulo del kernel
2. una libreria a livello utente per gestire la comunicazione col modulo del kernel
3. una implementazione del filesystem di interesse

Il progetto FUSE fornisce il modulo del kernel e la libreria/interfaccia per la comunicazione col kernel. L'implementazione della struttura filesystem deve essere fornita dallo sviluppatore.

I principali problemi da risolvere

Durante la fase di progettazione del nostro filesystem, analizzando i vari aspetti, abbiamo redatto un elenco di problemi fondamentali e di decisioni sostanziali da prendere al fine di rendere il filesystem stesso funzionante ed allo stesso tempo il più performante possibile:

1. Dove memorizzare gli hash dei file e delle cartelle?
2. Se si sceglie di memorizzarli in files, è meglio memorizzarli tutti in un unico grande file o fare più files piccoli?

3. Cosa succede se l'utente vuole creare un file avente lo stesso nome del file in cui è memorizzato un hash?
4. Il calcolo dell'hash deve essere ricorsivo?
5. Ogni quanto aggiornare i valori degli hash?
6. Quanto è pesante l'aggiornamento? Quanto scala?
7. Ci sono problemi legati alla concorrenza?

L'idea

Per realizzare il nostro filesystem in modo da risolvere i problemi visti precedentemente, abbiamo preso alcune decisioni.

Prima di tutto, abbiamo deciso di **mettere tutti gli hash** (sia dei files che delle cartelle) **in un unico grande file**; le principali motivazioni che ci hanno spinto a fare questa scelta sono due:

1. ridurre l'overhead nell'apertura e chiusura di file (evitando di dover effettuare tante aperture e chiusure di tanti files piccoli)
2. ridurre il tempo di ricerca e lettura (avendo un unico file, non c'è necessità di cercarlo come invece avremmo dovuto fare se avessimo avuto tanti files piccoli)

Questo file verrà caricato all'avvio del filesystem ed inserito all'interno di una struttura dinamica (una mappa). Ogni riga del file contiene una coppia chiave-valore della mappa, in cui la chiave è il path del file o cartella ed il valore è il suo hash.

Per quanto riguarda il nome di questo file, abbiamo pensato di scegliere un nome talmente particolare da fare in modo che la probabilità che un utente voglia creare un file con quel nome sia praticamente nulla; questo per due motivi:

1. non privare l'utente della possibilità di creare un file con un nome di cui abbia realmente bisogno
2. evitare sovrascritture del file

Inoltre, abbiamo anche pensato di creare il file come file nascosto (facendo iniziare il nome con un ".") e di impedire eventuali operazioni di modifica del file stesso (agendo su varie operazioni quali *rename*, *truncate*, ecc...).

Il nome stabilito è **".hashFSDataFile"**.

In questo modo abbiamo risolto i primi tre problemi.

Il quarto problema che ci eravamo posti era se il calcolo dell'hash dovesse essere effettuato in maniera ricorsiva. La risposta che ci siamo dati è stata "Sì"; questo princi-

palmente perché se modifico un file o una cartella il suo hash cambierà, ma cambieranno anche tutti i valori di hash delle cartelle che si trovano ad un livello di gerarchia più alto (cartella padre, cartella padre della cartella padre, ecc...); sarà quindi necessario richiamare la stessa funzione della modifica dell'hash anche su di esse.

Un ulteriore problema da risolvere era quello della frequenza dell'aggiornamento degli hash. Ogni quanto conviene o è necessario aggiornare tali valori?

Per capire ciò, è necessario avere ben chiaro un concetto: l'hash di un file cambia quando cambia il contenuto di quel file.

Detto ciò, è facile intuire che l'aggiornamento di un hash deve essere effettuato ogniquale volta viene effettuata un'operazione che modifica un file o una cartella; tali operazioni sono, per esempio, *mkdir*, *rmdir*, *unlink*, *rename*, ecc... (per maggiori dettagli, si veda il capitolo relativo ai tasks).

In un primo momento avevamo pensato di aggiornare sia la struttura dinamica che il file ad ogni modifica, al fine di evitare la perdita di dati in caso di crash del sistema operativo. Successivamente, però, ci siamo resi conto che questo approccio poteva essere molto oneroso dal punto di vista computazionale; abbiamo perciò deciso di effettuare una sola scrittura al momento dello smontaggio del filesystem. Per superare ad eventuali crash di sistema, abbiamo implementato un metodo di funzionamento simile a quello dei sistemi operativi. L'idea è che abbiamo un valore booleano scritto su un file (che abbiamo deciso di chiamare “**.hashFSUpToDate**” per gli stessi motivi visti per l'altro file) che ci indica se sono state effettuate modifiche dall'ultima scrittura su file degli hash e che viene quindi resettato ad ogni smontaggio del filesystem; se, durante l'operazione di montaggio del filesystem, questo valore è *True* vuol dire che c'è stato un crash di sistema e quindi si procede ad aggiornare il file.

L'ultimo aspetto da definire era quello relativo ad eventuali problemi di concorrenza. La struttura dinamica è sicuramente affetta da questo tipo di problemi; più file, infatti, potrebbero essere modificati contemporaneamente e quindi le scritture potrebbero essere contemporanee. Per questo motivo, abbiamo implementato un **sistema a semaforo**: quando viene richiesta l'istanza della mappa viene al contempo bloccato un semaforo dimodoché non sia possibile per altri processi richiedere anch'essi l'istanza. Un altro problema potrebbe essere quello di più processi che vogliono scrivere o modificare lo stesso file.

Come linguaggio di programmazione da utilizzare per l'implementazione, abbiamo scelto di utilizzare Python, principalmente per via della maggiore documentazione presente sul web.

Le operazioni

Per implementare il filesystem, siamo partiti da un filesystem “di base” già esistente: XMP.py (<https://stuff.mit.edu/iap/2009/fuse/examples/xmp.py>). La scelta di utilizzare questo filesystem come punto di partenza è stata dettata dal fatto che, per realizzare un qualsiasi filesystem FUSE, è necessario implementare tutte le operazioni che questo dovrà poter eseguire; dato che alcune di queste operazioni sono standard, abbiamo ritenuto opportuno partire da codice preesistente e funzionante ed apportare poi le modifiche necessarie, anche al fine di ottimizzare il tempo a disposizione.

Le operazioni di FUSE che è stato necessario modificare per far sì che il nostro filesystem gestisse correttamente gli hash sono le seguenti:

- *mkdir(self, path, mode)*: create a directory

È necessario modificarla perché si deve mettere nella struttura dinamica e nel file l'hash della cartella ed aggiornare quelli di eventuali cartelle di livelli superiori.

- *rmdir(self, path)*: delete an empty directory

È necessario modificarla perché si deve eliminare dalla struttura dinamica e dal file l'hash della cartella ed aggiornare quelli di eventuali cartelle di livelli superiori.

- *unlink(self, path)*: delete a file

È necessario modificarla perché si deve eliminare dalla struttura dinamica e dal file l'hash del file ed aggiornare quelli di eventuali cartelle di livelli superiori.

- *symlink(self, path, path1)*: create a symbolic link

È necessario modificarla perché si deve mettere nella struttura dinamica e nel file l'hash del link ed aggiornare quelli di eventuali cartelle di livelli superiori.

- *rename(self, path, path1)*: rename a file or folder

È necessario modificarla perché si devono modificare nella struttura dinamica e nel file l'hash del file/cartella e quelli di eventuali cartelle di livelli superiori.

- *mknod(self, path, mode, dev)*: create a file node

È necessario modificarla perché si deve mettere nella struttura dinamica e nel file l'hash del file ed aggiornare quelli di eventuali cartelle di livelli superiori.

- *getxattr(self, path, name, size)*: obtain an extended attribute

È necessario modificarla in modo da far ritornare come extended attribute l'hash del file/cartella

- *listxattr(self, path, size)*: obtain the list of all extended attributes

È necessario modificarla in modo da far ritornare come una lista contenente come extended attributes solo l'hash del file/cartella

- *truncate(path, len, fh)*: cut off at length

È necessario sovrascriverla perché si devono modificare nella struttura dinamica e nel file l'hash del file e quelli di eventuali cartelle di livelli superiori; se si riduce di troppo la dimensione del file, infatti, si elimina del contenuto dal file stesso e quindi l'hash è differente.

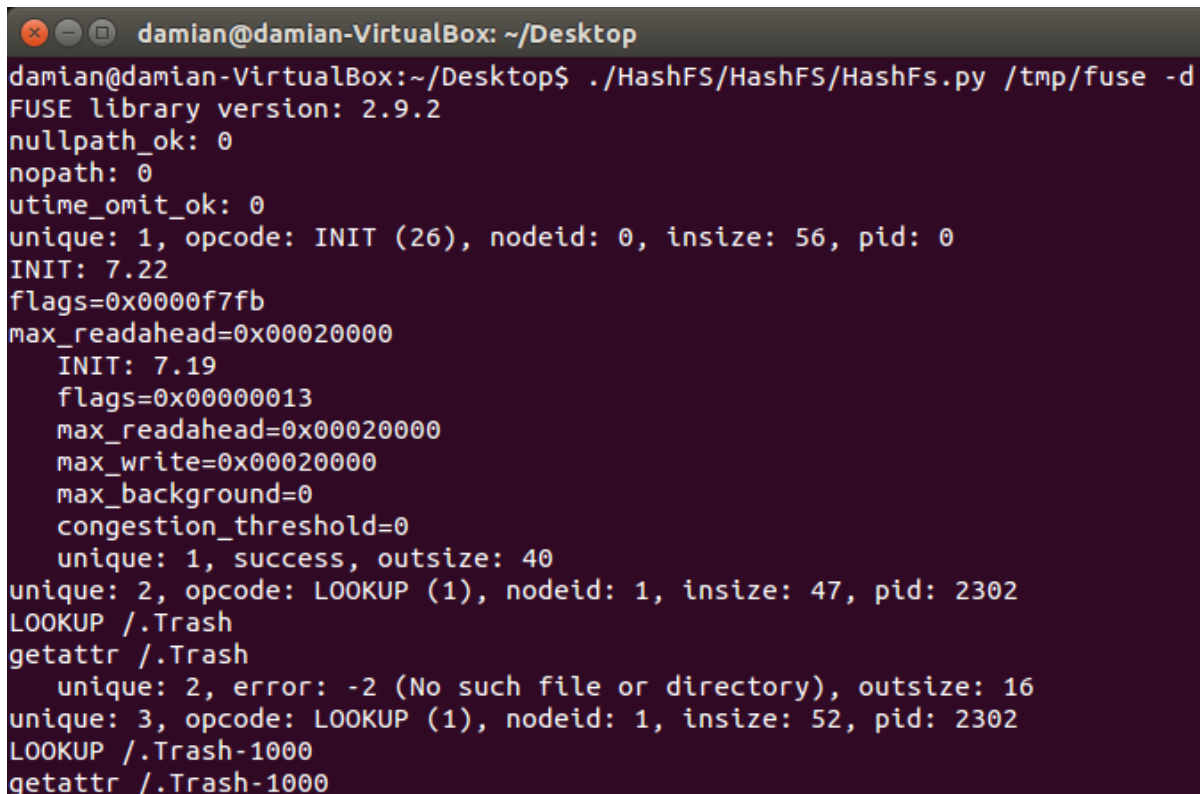
Oltre ad effettuare queste modifiche, è stato anche necessario implementare diverse classi e funzioni di supporto per il calcolo degli hash e la gestione delle strutture di memoria. Per maggiori dettagli, si veda il capitolo successivo.

Tasks del progetto

Risultato

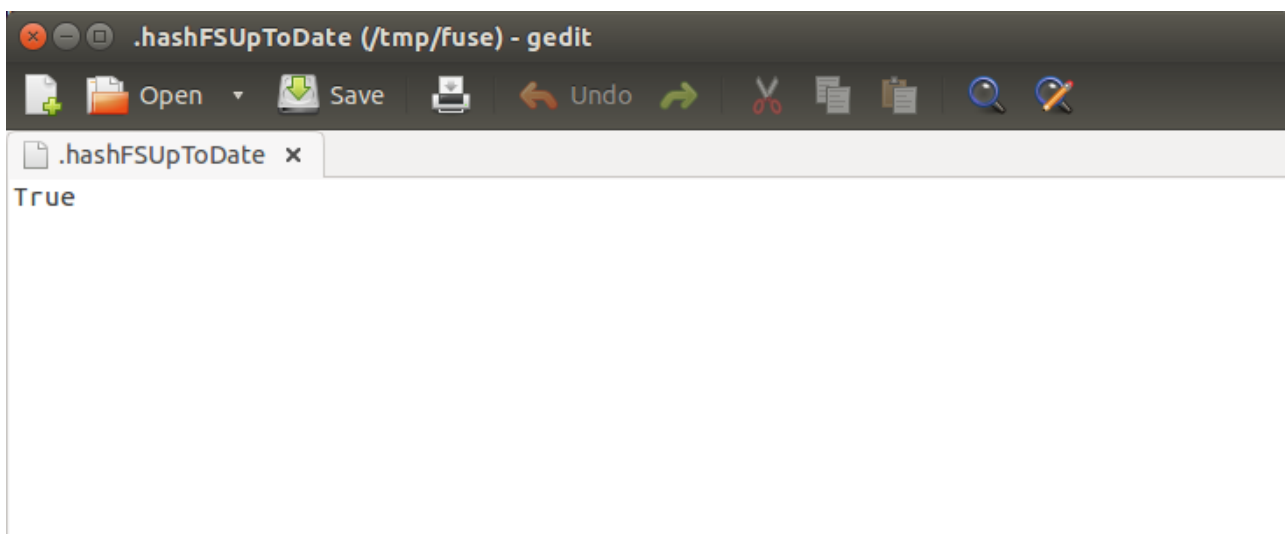
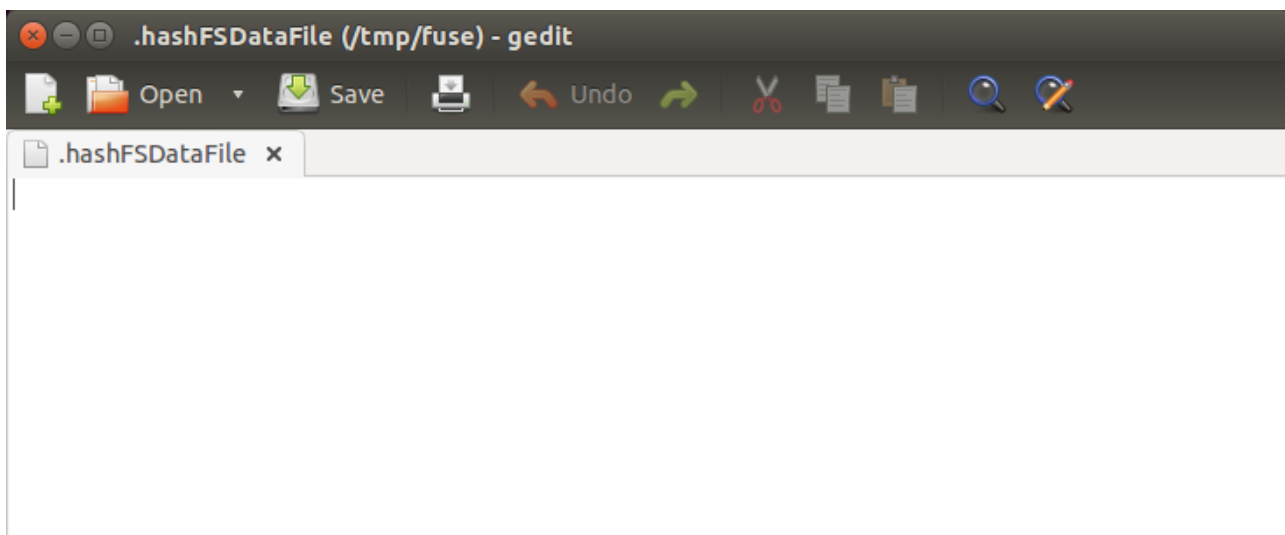
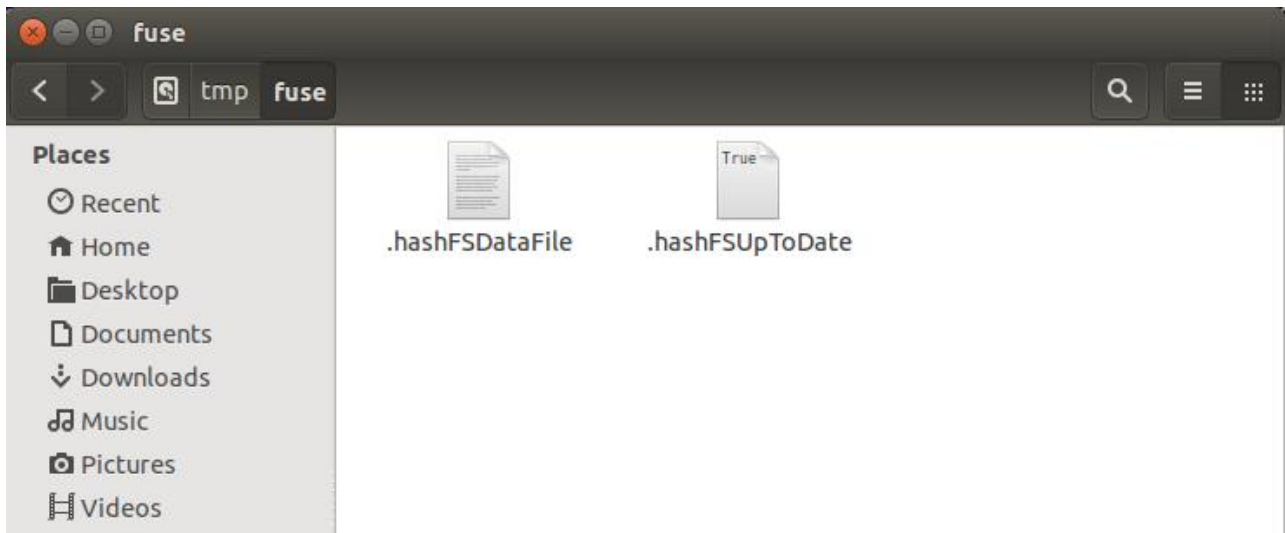
Al termine dell'implementazione, vediamo come si comporta il nostro filesystem.

Prima di tutto, ci spostiamo nella cartella che contiene il file HashFs.py e lo avviamo:

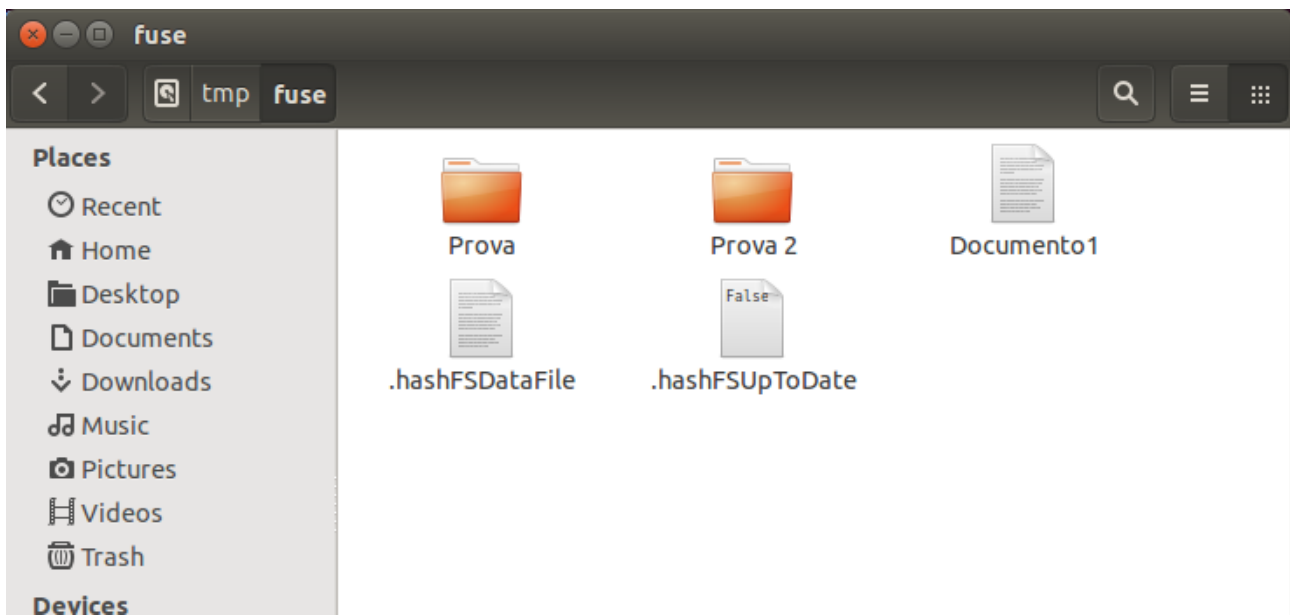


```
damian@damian-VirtualBox: ~/Desktop
damian@damian-VirtualBox:~/Desktop$ ./HashFS/HashFS/HashFs.py /tmp/fuse -d
FUSE library version: 2.9.2
nullpath_ok: 0
nopath: 0
utime_omit_ok: 0
unique: 1, opcode: INIT (26), nodeid: 0, insize: 56, pid: 0
INIT: 7.22
flags=0x0000f7fb
max_readahead=0x00020000
INIT: 7.19
flags=0x00000013
max_readahead=0x00020000
max_write=0x00020000
max_background=0
congestion_threshold=0
unique: 1, success, outsize: 40
unique: 2, opcode: LOOKUP (1), nodeid: 1, insize: 47, pid: 2302
LOOKUP /.Trash
getattr /.Trash
unique: 2, error: -2 (No such file or directory), outsize: 16
unique: 3, opcode: LOOKUP (1), nodeid: 1, insize: 52, pid: 2302
LOOKUP /.Trash-1000
getattr /.Trash-1000
```

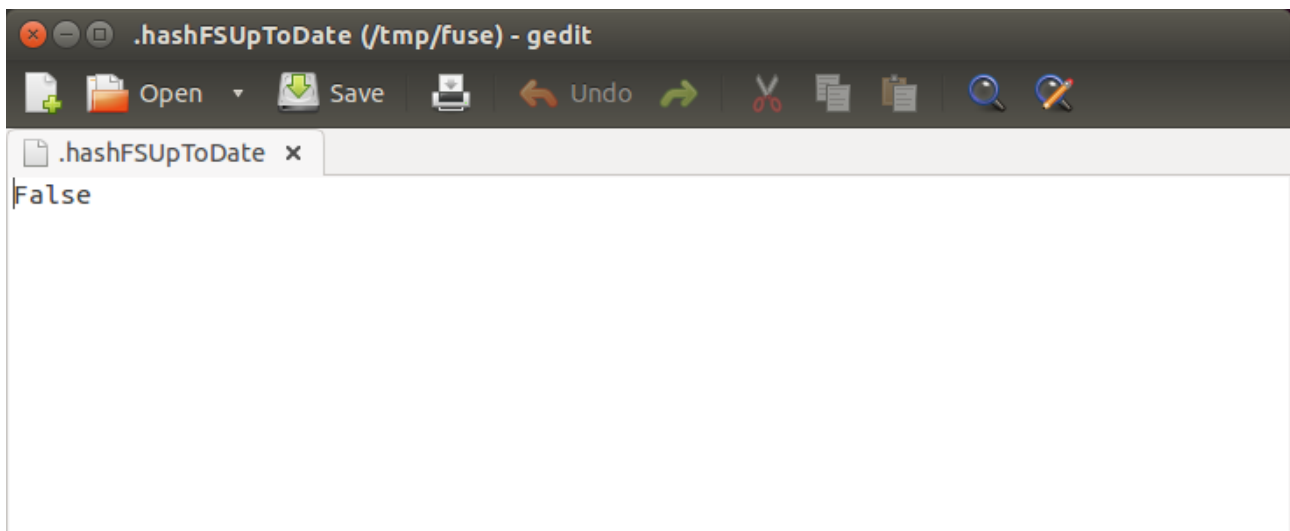
A questo punto andiamo nella cartella specificata come directory di montaggio del filesystem (nel nostro caso `/tmp/fuse`), abilitiamo la visualizzazione dei file nascosti e vediamo che al suo interno ci sono solo due file: quello contenente gli hash (vuoto) e quello contenente il valore booleano (che al momento ha valore *True*):

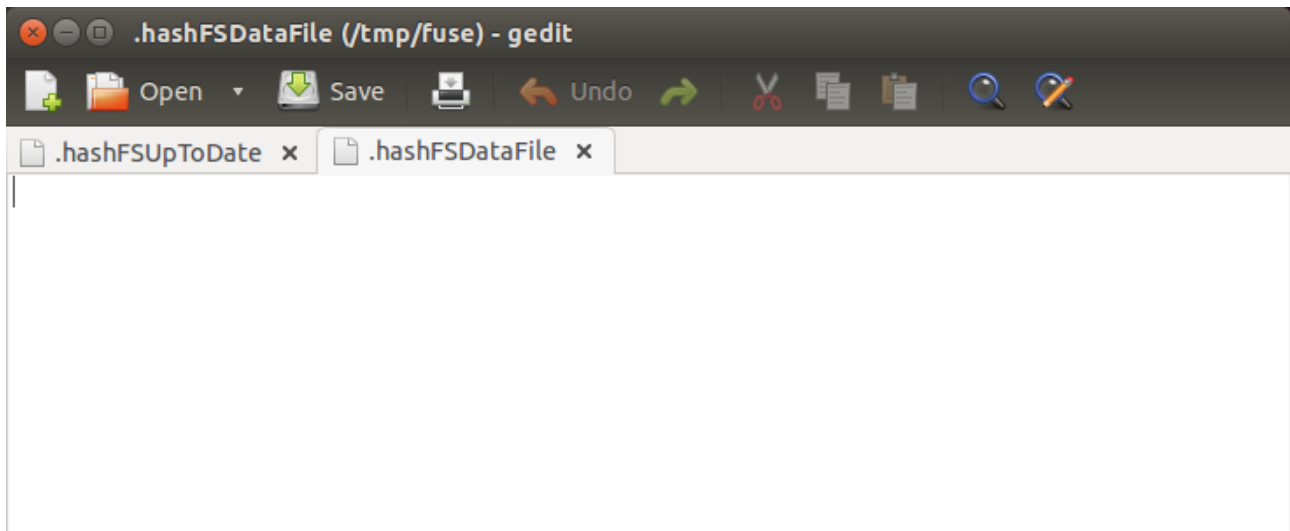


Adesso proviamo a creare qualche file e cartella all'interno del nostro filesystem:

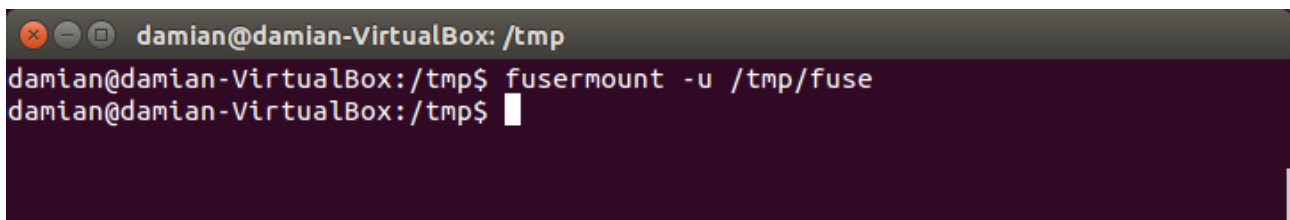


Come possiamo vedere, il valore del booleano è passato a *False* in quanto il file non è più aggiornato (è ancora vuoto):





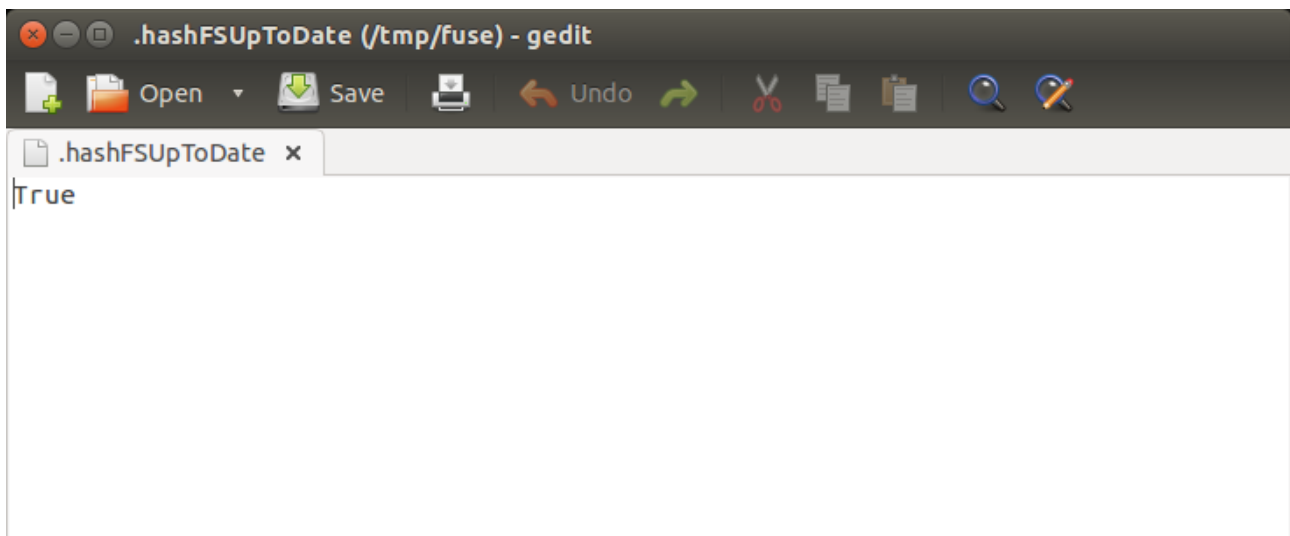
Per come è progettato il nostro filesystem, la scrittura sul file avviene alla chiusura del filesystem; procediamo quindi a smontarlo...



... ed a rimontarlo. Come possiamo vedere il file adesso è stato scritto:



Inoltre, il booleano è stato reimpostato a *True*:



A screenshot of a gedit window titled ".hashFSUpToDate (/tmp/fuse) - gedit". The window has a menu bar with "Open", "Save", "Undo", and other standard editing icons. The main text area shows the file ".hashFSUpToDate" with the content "True".

Aggiungendo altri file e cartelle, smontando e rimontando possiamo notare che il file degli hash viene sempre aggiornato correttamente:



A screenshot of a gedit window titled ".hashFSDataFile (/tmp/fuse) - gedit". The window shows a list of file paths and their corresponding hashes, each on a new line. The paths are located in /home/damian/HashFS/Prova and /home/damian/HashFS/Documento1. The hashes are 64-character hexadecimal strings.

```
/home/damian/HashFS/Prova/A/Index.txt:684d19b38a80ab8fe8423982bf7e90aa  
/home/damian/HashFS/Documento2:e5b39d71be95f5c2354bf2e3a1f3bfd3  
/home/damian/HashFS/Documento1:dc770f4c3a221a515cab939e73aab320  
/home/damian/HashFS/Prova 2/Lista.doc:6ea72c167cc6918d3359497b3f3b0215  
/home/damian/HashFS/Prova/Doc.html:a36d3ade18e31c5e02ef7d1ae1b72b70  
/home/damian/HashFS/Prova 3:1ccf24c1bf86405f93a9e0504a1581c4  
/home/damian/HashFS/Prova 2:0e83310a70b1bf39b6a24f124c63f0e6  
/home/damian/HashFS/Prova/A:5fe8c5f71bd6123a766ed3120d836d67  
/home/damian/HashFS/Prova:1978294410626e279f556d371d07a207
```