



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.3
Risultati.....	pag.4
 Tasks del progetto.....	 pag.5
Raccolta informazioni.....	pag.5
Metodi di valutazione delle prestazioni dei filesystems.....	pag.6
Definizione dell'architettura.....	pag.9
Implementazione del filesystem.....	pag.11
Tasks di Danilo Salvati.....	pag.15
Tasks di Damian Tosoni.....	pag.15
 Risultati.....	 pag.16

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 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** (Filesystem in **userspace**), un progetto open source che permette agli utenti non privilegiati di un sistema di *creare un proprio file system senza la necessità di scrivere codice a livello kernel*.

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?

RISULTATI

Alla fine dell'implementazione siamo riusciti ad ottenere un filesystem completamente funzionante e che gestisce gli hash nella maniera che desideravamo. Le prestazioni sono inferiori rispetto a quelle di un filesystem "classico", ma va tenuto in considerazione che questo dipende dalla mole di operazioni aggiuntive che deve svolgere per mantenere aggiornati gli hash.

Tasks del progetto

RACCOLTA INFORMAZIONI – IN COMUNE

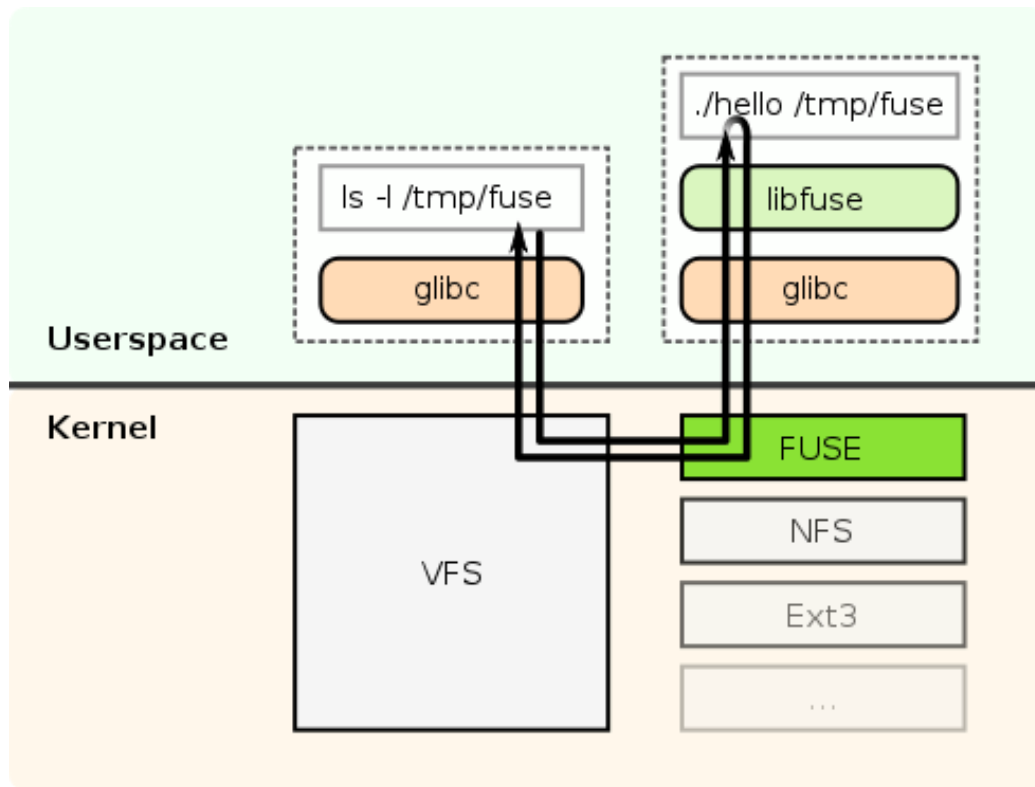
Prima di poter iniziare ad implementare il filesystem, è stato prima di tutto necessario reperire informazioni sullo strumento che saremmo andati ad usare: **FUSE**.

Per documentarci abbiamo utilizzato sia il sito ufficiale (<http://fuse.sourceforge.net/>), sia Wikipedia (http://en.wikipedia.org/wiki/Filesystem_in_Userspace), sia documenti e presentazioni di vari siti internet come, per esempio, quelli dell'Università di Bologna, di Berlino o del New Mexico. Lo strumento fondamentale, però, sono stati senza dubbio i vari esempi di filesystems già implementati, che ci hanno permesso di capire realmente come implementarne uno tutto nostro.

In breve, **FUSE**, acronimo che sta a significare **F**ilesystem in **u**space, 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 del filesystem deve essere fornita dallo sviluppatore.

METODI DI VALUTAZIONE DELLE PRESTAZIONI DEI FILESYSTEMS – IN COMUNE

Abbiamo effettuato svariate ricerche sul web per individuare un test che potesse essere adatto alla valutazione delle prestazioni del nostro filesystem; alla fine la scelta è caduta su **Bonnie++** (<http://www.coker.com.au/bonnie++/>), una nota suite di benchmark che si focalizza sulle velocità di lettura e scrittura dei dati e che quindi è apparsa subito perfetta per le nostre esigenze.

Nel dettaglio, i test eseguiti da Bonnie++ sono i seguenti:

1. Writing a byte at a time

Questo test scrive su un file utilizzando la funzione *putc*. Qui viene sollecitata principalmente la CPU, perché i dati vengono assorbiti dalla cache.

2. Writing intelligently

Questa volta il file viene scritto utilizzando una *write*. Il carico richiesto alla CPU consiste solamente nell'allocazione del file da parte del sistema operativo.

3. Rewriting

Ogni porzione del file pari alla dimensione del buffer viene letta con una *read*, modificata e poi riscritta con una *write* utilizzando anche una *lseek*. Poiché non viene fatta alcuna allocazione dello spazio e l'I/O è localizzato, questo test verifica il corretto funzionamento della cache e la velocità del trasferimento dati.

4. Reading a byte at a time

Il file viene letto utilizzando *getc*. Questo mette sotto sforzo solo *stdio* e l'input sequenziale.

5. Reading intelligently

In questo caso vengono valutate le prestazioni del filesystem per la lettura di file usando la funzione *read*.

6. Random seeks

Questo test avvia numerosi processi che effettuano diverse *lseek* in posizioni casuali del file per poi leggerle con la funzione *read*. Nel 10% dei casi la porzione di file viene modificata e riscritta con una *write*. L'idea dietro l'uso di questi processi, è quella di essere sicuri di avere sempre una *seek* in coda.

7. Create files in sequential order

Questo test crea una grande quantità di file utilizzando delle *create*. Questi hanno nomi che cominciano con sette numeri seguiti da un numero casuale (da zero a dodici) di caratteri alfanumerici.

8. Stat files in sequential order

Richiama *stat* su ogni file.

9. Delete files in sequential order

I file precedentemente creati vengono eliminati con l'operazione *unlink*.

10.Create files in random order

Il processo è simile a quello della creazione in ordine sequenziale, solo che i nomi dei file presentano prima i caratteri casuali e poi quelli fissi.

11.Stat files in random order

Si richiama *stat* sui file appena creati.

12.Delete files in random order

Si procede all'eliminazione dei file in ordine casuale.

In sostanza, i primi sei vogliono emulare tutte quelle attività che risultano essere il collo di bottiglia per applicazioni I/O intensive. Gli ultimi sei, invece, coinvolgono le funzioni *create/stat/unlink* per simulare operazioni che rappresentano il collo di bottiglia per macchine in cui si hanno migliaia di file in una cartella.

Per capire come si comportasse il nostro filesystem, abbiamo deciso di confrontarlo con quello standard di Linux (ext4); abbiamo perciò eseguito il test anche su di esso e successivamente confrontato i risultati.

Questi sono i risultati riportati da Bonnie++ sul nostro filesystem:

Version 1.97		Sequential Output						Sequential Input				Random Seeks		Num Files	Sequential Create						Random Create					
	Size	Per Char		Block		Rewrite		Per Char		Block					Create	Read		Delete	Create	Read		Delete				
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU		/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU		
daniilo-desktop	4M	9	26	+++++	+++	+++++	+++	1092	99	+++++	+++	13694	80	5	32	0	+++++	+++	35	0	31	0	+++++	+++	35	0
	Latency	937ms		2397us		8074us		8058us		2506us		259ms		Latency	103ms		22825us		118ms		82325us		4032us		60821us	

Questi altri, invece, sono i risultati sul filesystem di Linux:

Version 1.97		Sequential Output					Sequential Input					Random Seeks		Num Files	Sequential Create						Random Create					
	Size	Per Char	Block		Rewrite		Per Char	Block		Create	Read				Delete		Create	Read		Delete						
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU		/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU		
daniilo-desktop	4M	303	98	+++++	+++	+++++	+++	1031	99	+++++	+++	+++++	+++	5	+++++	+++	+++++	+++	+++++	+++	+++++	+++	+++++	+++		
	Latency	31805us	66us	72us		8020us		17us	20038us		Latency	424us	1133us		1161us	455us	22us		422us							

La riga più interessante è quella relativa alla latenza. In effetti possiamo notare come vi siano enormi differenze tra i due filesystem (anche di diversi ordini di gran-

dezza). Dobbiamo però ricordare che FUSE si interpone tra l'utente ed il kernel traducendo le istruzioni del primo in equivalenti istruzioni privilegiate. Tale processo implica quindi un consistente overhead con conseguente perdita di efficienza. Inoltre parte della perdita di performance è dovuta anche alla struttura stessa di HashFS. Si può poi notare come questa sia maggiore per le scritture; ciò avviene perché ogni volta che viene modificato un file si procede al ricalcolo dell'hash. Anche la fase di creazione dei file risente di tale problema.

Inoltre, durante l'esecuzione dei test, si sono notati rallentamenti progressivi man mano che si procedeva. Questo perché all'aumentare del numero di file all'interno della directory, aumentava anche il tempo di calcolo per il suo hash, in quanto è necessario analizzare il suo contenuto.

Per quanto detto fino ad adesso, appare chiaro come sia estremamente complicato aumentare le prestazioni mantenendo la medesima tecnologia. Probabilmente l'unica strada percorribile consiste nell'implementazione di questo filesystem a livello del kernel linux, sebbene ciò comporti maggiori oneri per lo sviluppo.

DEFINIZIONE DELL'ARCHITETTURA – IN COMUNE

Per realizzare il nostro filesystem in modo da risolvere i problemi visti nell'Abstract, abbiamo preso le seguenti 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 principalmente 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 sopperire 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 il file contenente gli hash è aggiornato e che viene quindi resettato a *True* ad ogni smontaggio del filesystem; se, durante l’operazione di montaggio del filesystem, questo valore è *False* 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 che sfrutta un semaforo binario**: 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.

IMPLEMENTAZIONE DEL FILESYSTEM – 50% CIASCUNO

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

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.

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.

In dettaglio, quindi, i tasks sono risultati i seguenti:

1. Definizione della funzione **updateDirectoryHash(path, hash_data_structure, hash_calculator)**

Questa funzione ha il compito di eseguire l'aggiornamento dei valori di hash di una cartella e di quelli delle eventuali cartelle genitori (cioè le cartelle di livelli superiori). La prima cosa che si fa è quella di aggiungere il path della directory ed il suo hash al dizionario contenente gli hash; successivamente si fa la stessa cosa per le cartelle di livello più alto (ricavate usando la libreria “os” di python) fino ad arrivare alla cartella di root (esclusa).

2. Definizione dell'operazione **unlink(self, path)**

Questa operazione viene richiamata quando si tenta di eliminare un file. Per prima cosa, controlliamo se il file che si vuole eliminare è uno dei due file necessari per il funzionamento del filesystem (quello contenente gli hash e quello contenente il valore booleano); in questo caso viene restituita un'eccezione ed il file non viene eliminato. Altrimenti, il file viene eliminato, la sua entry nel dizionario contenente gli hash viene rimossa e si procede ad aggiornare le eventuali cartelle di livello più alto.

3. Definizione dell'operazione **rmdir(self, path)**

Questa operazione viene richiamata quando si cerca di eliminare una cartella. L'implementazione prevede perciò che per prima cosa venga eliminata la cartella e, successivamente, la sua entry dal dizionario contenente gli hash. Infine si procede ad aggiornare le eventuali cartelle di livello più alto.

4. Definizione dell'operazione **symlink(self, path, path1)**

Questa operazione viene richiamata quando si vuole creare un link simbolico. Anche in questo caso, prima di tutto si crea il link, si aggiunge una entry al dizionario degli hash contenente il suo path ed il suo valore di hash ed infine si procede ad aggiornare le eventuali cartelle di livello più alto.

5. Definizione dell'operazione **rename(self, path, path1)**

Questa operazione viene richiamata quando si tenta di rinominare un file o una cartella. Per prima cosa, controlliamo se il file che si vuole rinominare è uno dei due file necessari per il funzionamento del filesystem (quello contenente gli hash e quello contenente il valore booleano); in questo caso viene restituita un'eccezione ed il file non viene rinominato. Altrimenti, il file/cartella viene rinominato, la sua entry nel dizionario contenente gli hash viene aggiornata e si procede ad aggiornare anche le eventuali cartelle di livello più alto e/o più basso.

6. Definizione della funzione **__update_child_path(self, new_path, old_path)**

Questa funzione calcola in maniera ricorsiva l'hash dei figli di una cartella rinominata. Prima di tutto si vede se è un file o una cartella (per via delle chiamate ricorsive).

Se è una cartella si vede se è vuota: se non lo è, si aggiornano l'hash dei file al suo interno e gli hash dei livelli inferiori di tutte le cartelle al suo interno ricorsivamente; se lo è, si richiama la funzione `updateDirectoryHash` in modo da aggiornare l'hash della cartella stessa ed anche quelli dei livelli superiori. Se è un file, si aggiunge la sua entry al dizionario degli hash e si aggiorna la cartella padre.

Infine, si rimuove dal dizionario l'entry relativa al vecchio path.

7. Definizione dell'operazione **truncate(self, path, len)**

Questa operazione viene richiamata quando si vuole troncare la dimensione di un file ad un determinato valore. Per prima cosa, controlliamo se il file che si vuole troncare è uno dei due file necessari per il funzionamento del filesystem (quello contenente gli hash e quello contenente il valore booleano); in questo caso viene restituita un'eccezione ed il file non viene troncato. Altrimenti, il file viene troncato, la sua entry nel dizionario contenente gli hash

viene aggiornata e si procede ad aggiornare anche le eventuali cartelle di livello più alto.

8. Definizione dell'operazione **mknod(self, path, mode, dev)**

Questa operazione viene richiamata quando si vuole creare un file. L'implementazione prevede che si crei il nodo richiamando l'operazione di sistema e, successivamente, si aggiunga la sua entry nel dizionario contenente gli hash (path -> valore di hash). Infine si procede ad aggiornare le eventuali cartelle di livello più alto.

9. Definizione dell'operazione **mkdir(self, path, mode)**

Questa operazione viene richiamata quando si vuole creare una cartella. L'implementazione prevede che si crei la cartella richiamando l'operazione di sistema e, successivamente, si aggiunga la sua entry nel dizionario contenente gli hash (path -> valore di hash), procedendo anche ad aggiornare le eventuali cartelle di livello più alto.

10. Definizione delle operazioni **getxattr(self, path, name, size)**, **listxattr(self, path, size)** e **removexattr(self, path, name)**

Questa operazione vengono usate per la gestione degli attributi estesi (nel nostro caso uno solo, l'hash del file o cartella). L'implementazione non è stata modificata se non in due punti: in *getxattr* è stato definito che il valore di ritorno debba essere l'hash del file o cartella, preso dal dizionario degli hash; in *listxattr* è stato impostato che la lista degli attributi contenga solo l'attributo "hash". *removexattr* è stata lasciata solo perché la sua presenza è necessaria per il funzionamento del filesystem (altrimenti restituisce un errore perché non la trova); dato che però non serve per la gestione degli hash, la sua implementazione è vuota.

11. Definizione della funzione **__init__(self, path, flags, *mode)** della classe **HashFSFile**

Questa operazione viene richiamata quando si vuole accedere ad un file. Prima di tutto si controlla se il file esiste: se esiste lo si apre; se non esiste lo si crea, si aggiunge la sua entry nel dizionario contenente gli hash (path -> valore di hash) ed infine si procede ad aggiornare le eventuali cartelle di livello più alto.

12. Definizione della classe **HashDataStructure**

La classe *HashDataStructure* si occupa di memorizzare e manipolare gli hash utilizzati dal filesystem. Dal punto di vista architetturale si tratta di un singleton in cui viene restituito un Dictionary avente come chiave il nome completo della risorsa e come valore il suo hash.

Il metodo fondamentale per ottenere la struttura è **get_data_structure_instance**. Se questa non è stata ancora istanziata allora viene creato un nuovo Dictionary e si controlla se erano stati memorizzati degli hash nel filesystem. In caso affermativo questi vengono ricaricati a partire dal file e viene restituita la struttura. Al contrario, se non vi erano hash salvati, allora si crea il file per la memorizzazione e si restituisce il Dictionary vuoto.

Un caso a parte si può verificare se il filesystem non è stato smontato correttamente. In tale eventualità, infatti, non essendo sicuri che il file contenente gli hash sia corretto, si procede ad un ricalcolo degli hash di tutte le risorse.

Sempre qui viene istanziato (se non era ancora stato creato) e successivamente bloccato un mutex associato alla struttura. Questo serve ad impedire accessi concorrenti da parte del filesystem, ma allo stesso tempo costringe i processi a rimanere in attesa della risorsa se già impegnata. Per questo motivo, è molto importante richiamare il metodo **release_data_structure** per sbloccare il semaforo una volta terminata la propria elaborazione. Un'altra soluzione, specialmente per task che non implicano la scrittura nella struttura, consiste nell'invocare il metodo **get_structure_snapshot**, che restituisce una copia del Dictionary.

In ogni caso per evitare di richiamare continuamente i metodi per la richiesta ed il rilascio della struttura, sono state definite alcune funzioni di utilità che incapsulano tali problematiche:

- **write_data_structure**: permette di salvare su file il contenuto del Dictionary
- **get_file_hash**: ottiene una singola entry della struttura
- **insert_hash**: inserisce o aggiorna una entry della struttura
- **remove_hash**: rimuove un hash dalla struttura

13. Definizione delle classi **HashCalculator** e **HashCalculatorMD5**

La classe *HashCalculator* si occupa di calcolare l'hash di un file o di una cartella. Dal punto di vista del codice, questa è stata organizzata come una classe astratta in cui l'implementazione concreta va a definire quale algoritmo utilizzare per il calcolo. Nel filesystem è stato utilizzato *MD5*, come si può osservare dalla classe *HashCalculatorMD5*.

I due metodi fondamentali sono senza dubbio **calculateFileHash** e **calculateDirectoryHash**.

Come si può facilmente intuire dal nome, il primo calcola l'hash associato ad un file, prendendone in considerazione il nome ed il contenuto; il secondo si rivolge invece alle directory considerando il nome e gli hash dei figli in esse contenute. Per motivi di efficienza, è stato scelto di richiamare il Dictionary di HashDataStructure per calcolare gli hash dei figli, evitando così di effettuare molte chiamate ricorsive.

Nella divisione dei tasks, abbiamo cercato di ripartire il lavoro nella maniera più equa possibile.

Tasks di Danilo Salvati:

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 11 - 12 (funzioni *get_data_structure_instance*, *release_data_structure* ed *insert_hash*)

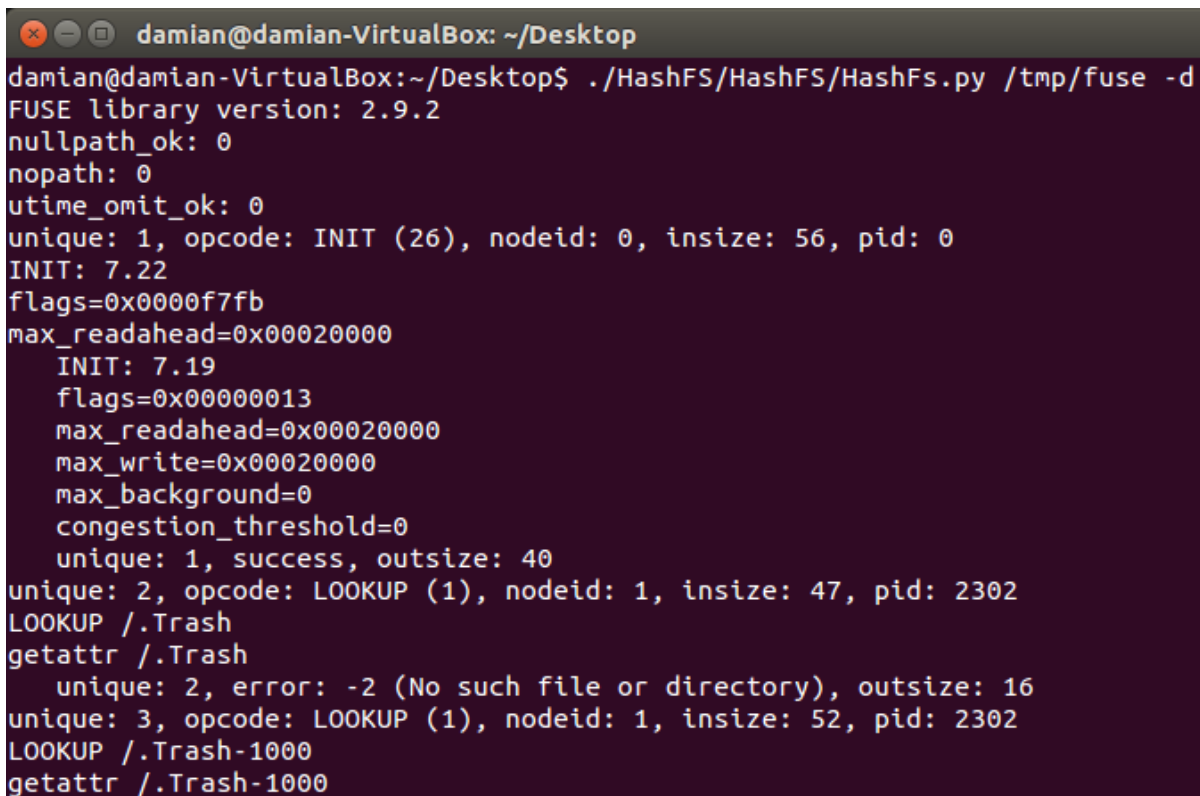
Tasks di Damian Tosoni:

10 - 12 (funzioni *get_structure_snapshot*, *__initialize_data_map*, *__load_data_map_from_file*, *__reloadAllHashes*, *write_data_structure*, *get_file_hash*, *remove_hash*, *__update_boolean_file* e *__read_boolean_file(self)*) - 13 (funzioni *calculateFileHash*, *calculateDirectoryHash*, *HashCalculatorMD5*)

Risultato

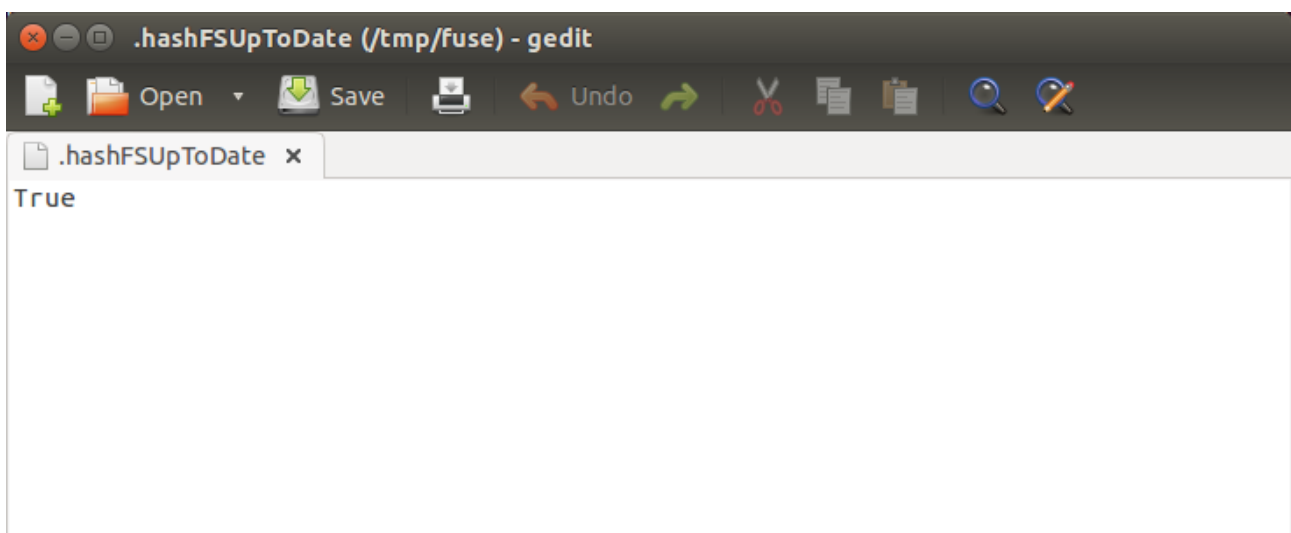
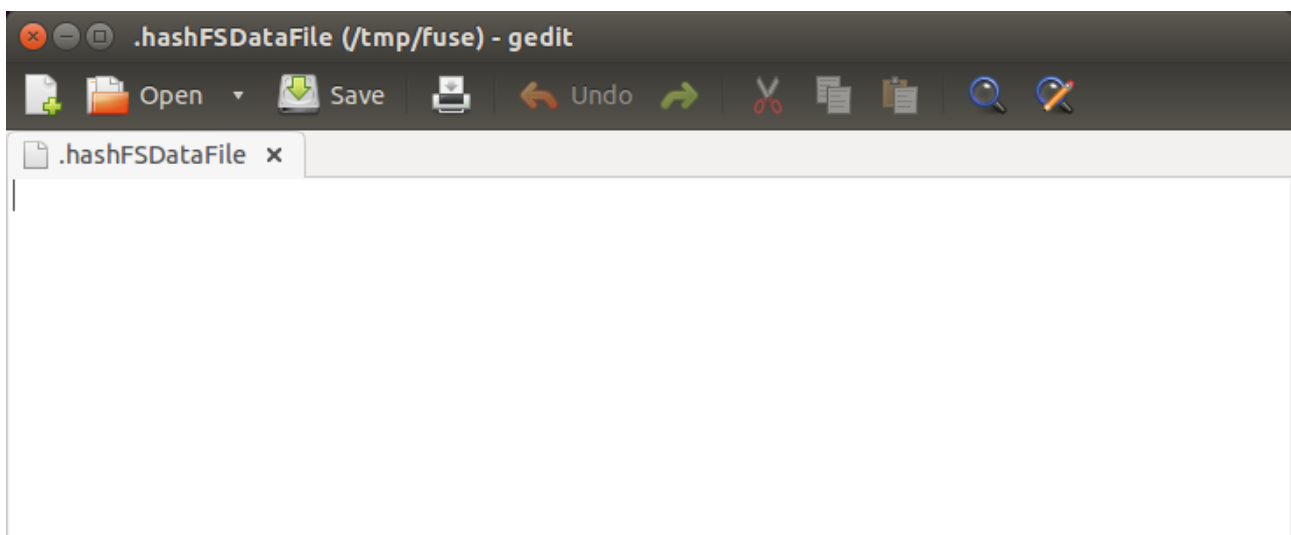
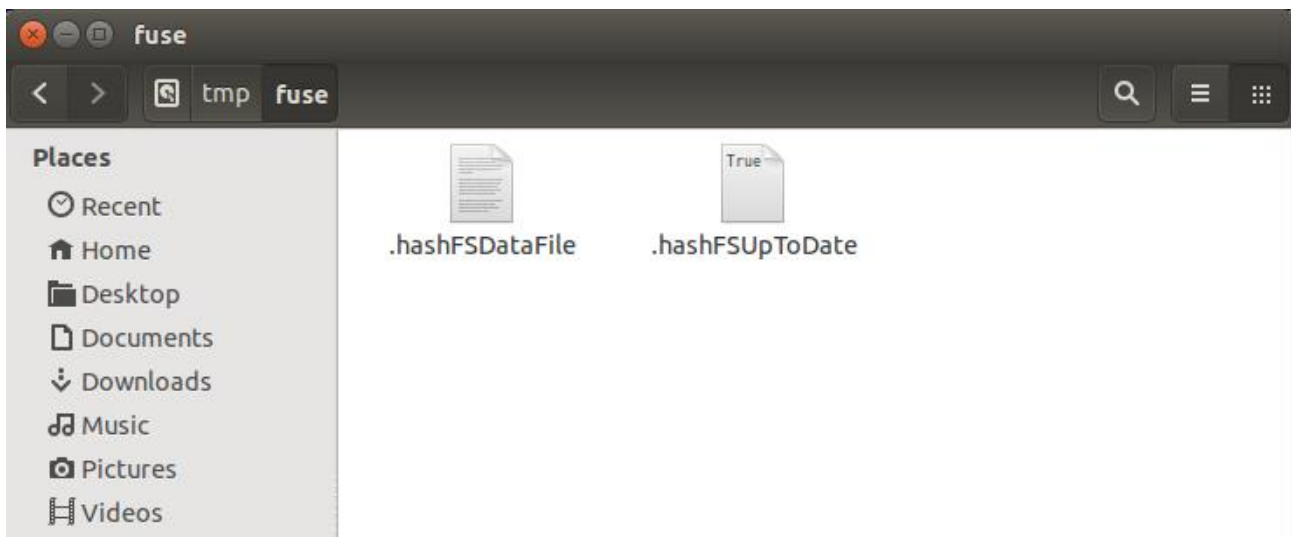
Al termine dell'implementazione, vediamo come si comporta il nostro filesystem.

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

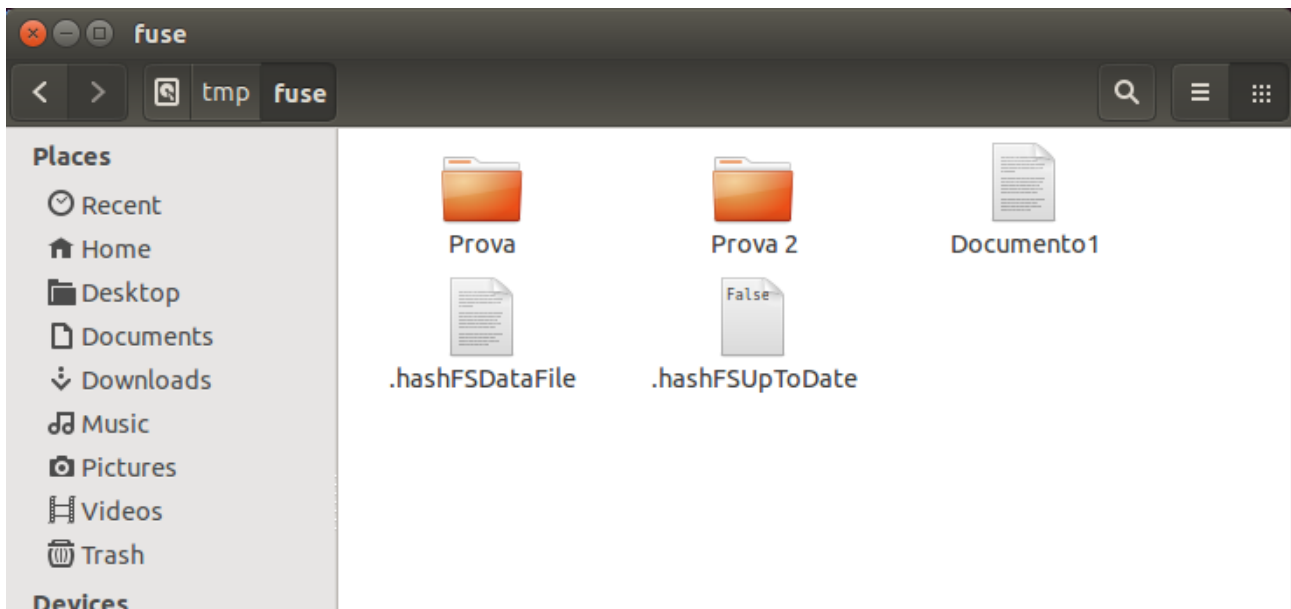


```
damian@damian-VirtualBox: ~/Desktop
damian@damian-VirtualBox:~/Desktop$ ./HashFS/HashFS/HashFs.py /tmp/fuse -d
FUSE library version: 2.9.2
nullpath_ok: 0
nopath: 0
utime_omit_ok: 0
unique: 1, opcode: INIT (26), nodeid: 0, insize: 56, pid: 0
INIT: 7.22
flags=0x0000f7fb
max_readahead=0x00020000
  INIT: 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
```

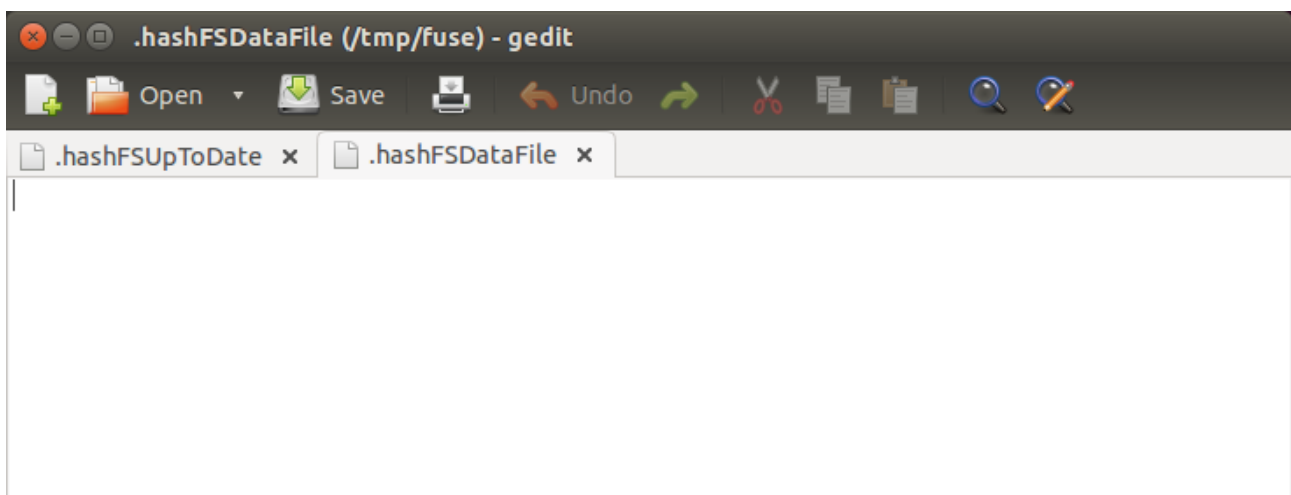
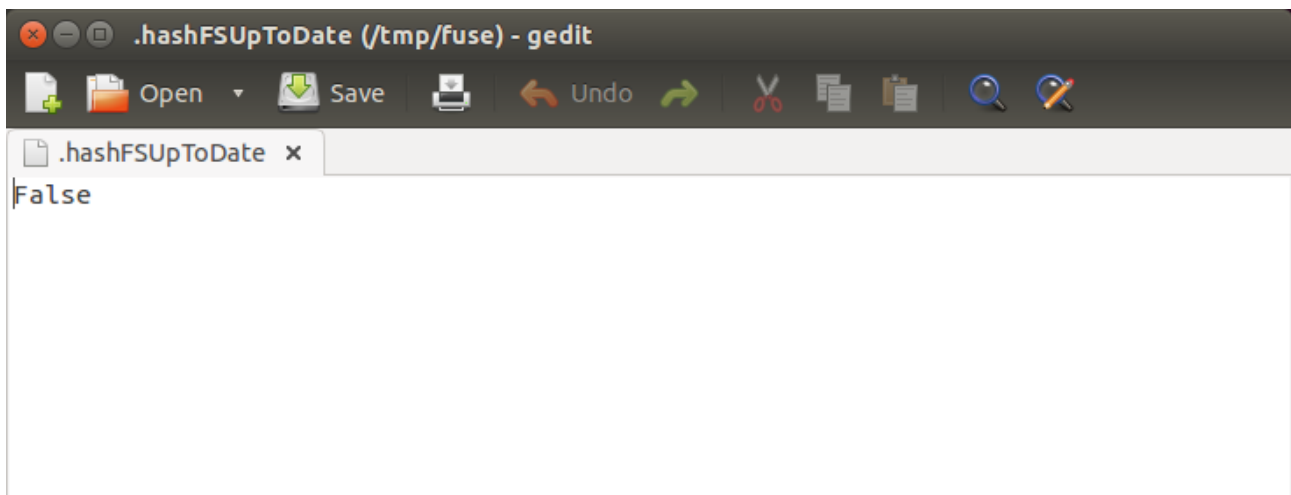
A questo punto andiamo nella cartella specificata come directory di montaggio del filesystem (nel nostro caso `/tmp/fuse`), abilitiamo la visualizzazione dei file nascosti e vediamo che al suo interno ci sono solo due file: quello contenente gli hash (vuoto) e quello contenente il valore booleano (che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

```
damian@damian-VirtualBox: /tmp
damian@damian-VirtualBox:/tmp$ fusermount -u /tmp/fuse
damian@damian-VirtualBox:/tmp$
```

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

```
.hashFSDataFile (/tmp/fuse) - gedit
Open Save Undo
.hashFSDataFile x
/home/damian/HashFS/Documento1:dc770f4c3a221a515cab939e73aab320
/home/damian/HashFS/Prova:d272ce848b4d6bbf7b9e0da44fab700e
/home/damian/HashFS/Prova 2:703c6b0dc463be938827c6a642b18ede
```

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

```
.hashFSUpToDate (/tmp/fuse) - gedit
Open Save Undo
.hashFSUpToDate x
True
```

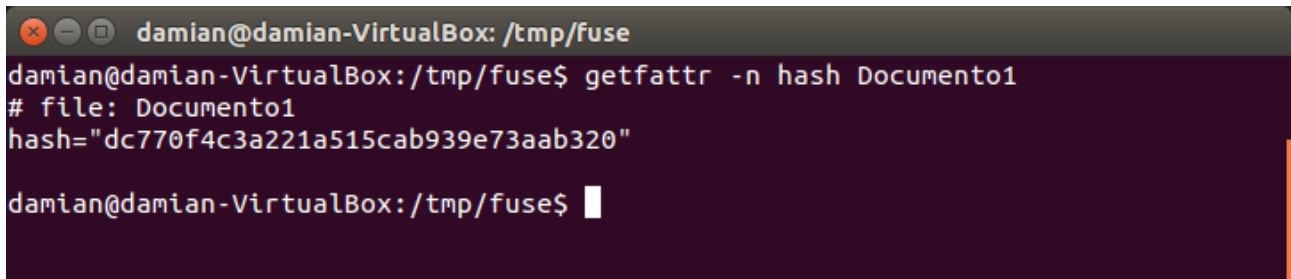
Aggiungendo altri file e cartelle, smontando e rimontando possiamo notare che il file degli hash viene sempre aggiornato correttamente:



The screenshot shows a gedit editor window titled ".hashFSDaDataFile (/tmp/fuse) - gedit". The window contains a list of file paths followed by their corresponding SHA-256 hashes. The files are: Index.txt, Documento2, Documento1, Lista.doc, Doc.html, and three files in the Prova directory (Prova 3, Prova 2, and Prova A).

```
.hashFSDaDataFile x
/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
```

È possibile visualizzare l'hash di un file anche senza aprire il file, utilizzando il seguente comando:



The screenshot shows a terminal window with the prompt "damian@damian-VirtualBox: /tmp/fuse". The user enters the command "getfattr -n hash Documento1", which returns the hash for the file "Documento1".

```
damian@damian-VirtualBox: /tmp/fuse
damian@damian-VirtualBox:/tmp/fuse$ getfattr -n hash Documento1
# file: Documento1
hash="dc770f4c3a221a515cab939e73aab320"

damian@damian-VirtualBox:/tmp/fuse$
```