# HashFS

Un filesystem basato sull'hashing.

Damian Tosoni, Danilo Salvati

Sicurezza dei sistemi informatici e delle reti

# Indice



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

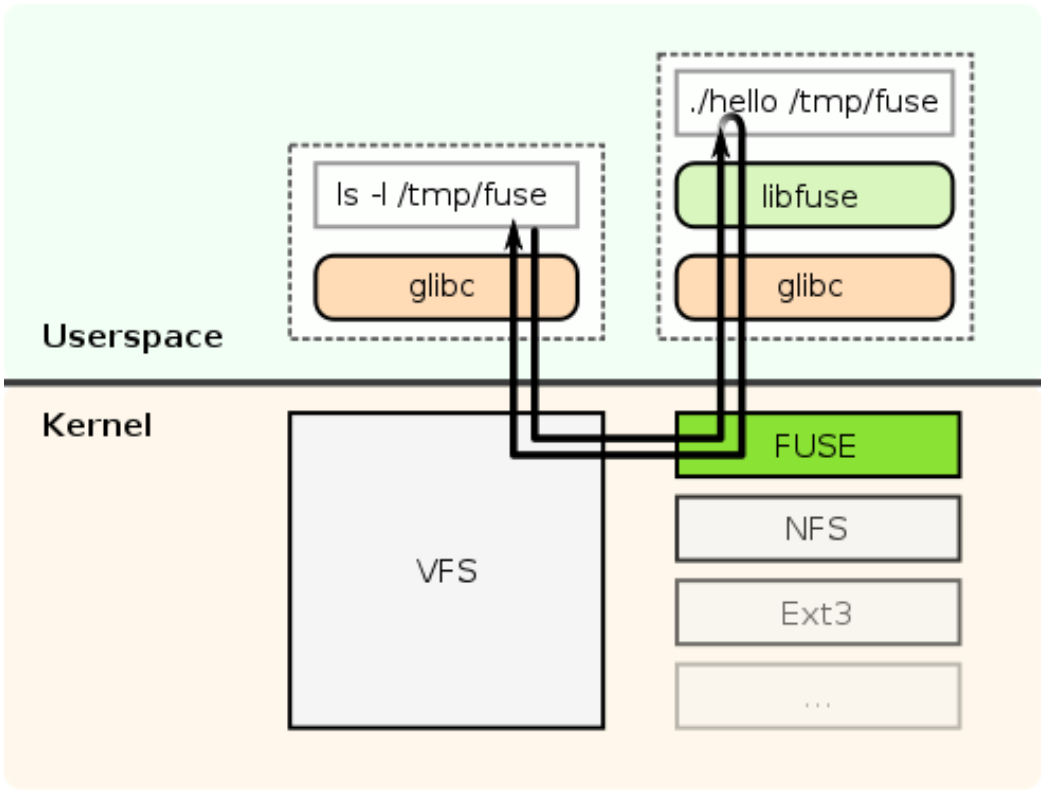
L'obiettivo principale del progetto è realizzare un filesystem che mantenga persistente e aggiornato, per ciascuna directory, un hash della directory stessa (calcolato in modo ricorsivo).

## APPROCCIO

Per portare a termine questo obiettivo, ci siamo avvalsi dell'utilizzo di **FUSE**, un acronimo che sta a significare **F**ilesystem in **use**rspace. 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 richeste 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 aggiorno 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, in un primo momento avevamo pensato di scegliere un nome talmente particolare da fare in modo che la probabilità che un utente volesse creare un file con quel nome fosse praticamente nulla; successivamente, però, abbiamo pensato che qualche hacker avrebbe potuto benissimo mettersi a tentare tutte le combinazioni e quindi alla fine indovinare il nome. Per questo motivo abbiamo deciso di impedire del tutto la creazione di un file con lo stesso nome del nostro (modificando l'operazione *create*) e di creare il file come file nascosto (facendo iniziare il nome con un "."). È stato comunque usato un nome "fantasioso" in modo da non privare l'utente della possibilità di creare un file con un nome di cui avesse realmente bisogno.
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 per un motivo: 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…) quindi sarà 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 ogniqualvolta 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 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:

Filesystem Operations

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

  È necessario modificarla perché si deve mettere nella struttura dinamica e nel file l'hash della cartella.

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

  È necessario modificarla perché si deve eliminare dalla struttura dinamica e dal file l'hash della cartella.

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

  È necessario modificarla perché si deve eliminare dalla struttura dinamica e dal file l'hash del file e modificare quello della cartella che lo conteneva.

- *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.

- *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 quello della cartella che lo contiene.

- *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 e modificare quello della cartella che lo contiene.

File Operations (NON ANCORA FATTE, VANNO FATTE?)

- *rename(old, new)*: move and/or rename a file

  È necessario sovrascriverla perché si devono modificare nella struttura dinamica e nel file l'hash del file e quello della cartella che lo contiene.

- *create(path, flags, mode)*

  È necessario sovrascriverla perché si deve mettere nella struttura dinamica e nel file l'hash del file e modificare quello della cartella che lo contiene.

- *write(path, buf, offset, fh)*

  È necessario sovrascriverla perché si devono aggiornare nella struttura dinamica e nel file l'hash del file e quello della cartella che lo contiene.

- *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 quello della cartella che lo contiene; se si riduce di troppo la dimensione del file, infatti, si elimina del contenuto dal file stesso e quindi l'hash è differente.

# Tasks del progetto