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

Capitolo 1

Lo strumento fondamentale: FUSE

Che cos'è

FUSE è 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.

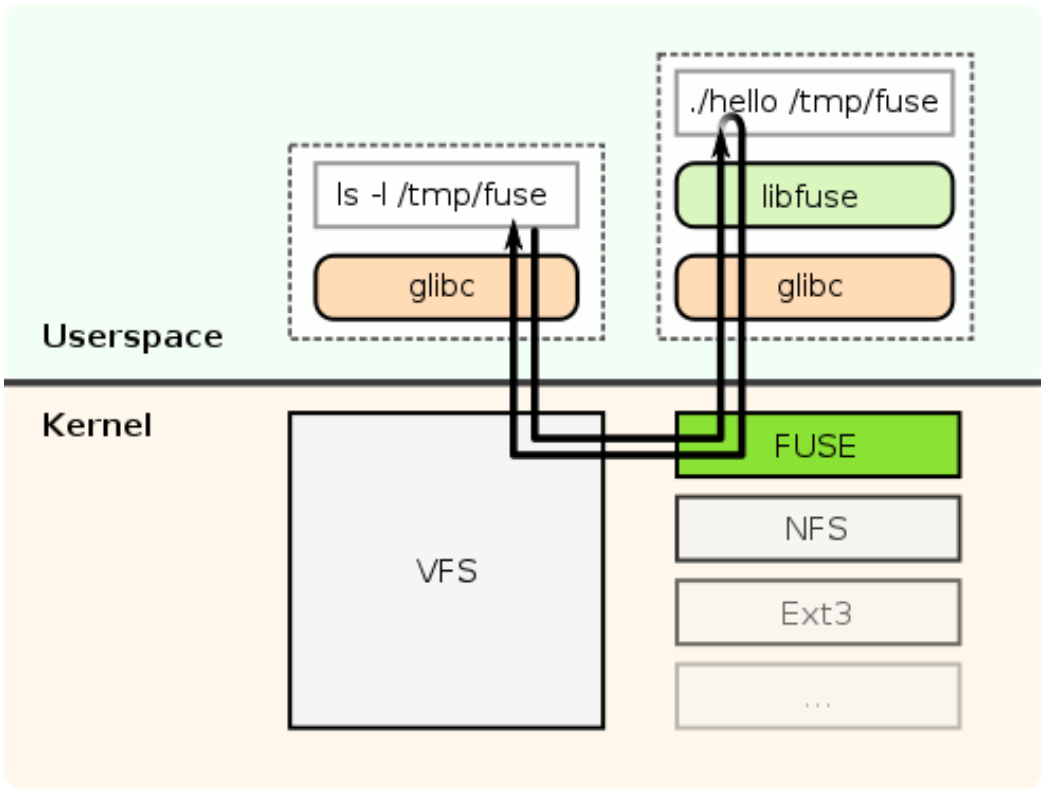
FUSE è particolarmente utile per scrivere *filesystem virtuali*. A differenza dei filesystem tradizionali che si preoccupano principalmente di organizzare e memorizzare i dati su disco, i filesystem virtuali non memorizzano realmente i dati per conto proprio ma come un tramite fra l'utente ed il filesystem reale sottostante.

FUSE è divenuto ufficialmente parte del codice del kernel Linux a partire dalla release 2.6.14, è in grado di supportare la semantica dei più comuni filesystem ed è disponibile per Linux, FreeBSD, OpenSolaris e Mac OS X.

Come funziona

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.

Esempi di filesystems realizzati con FUSE

Esistono decine di filesystems che sono realizzati utilizzando FUSE, più o meno famosi. Tra i più noti ricordiamo:

- **SSHFS**: permette di montare in locale una directory posizionata su un server remoto in cui gira SSH, con il vantaggio di avere una connessione cifrata tramite ssh non intercettabile.

- **EncFS**: un filesystem crittografico che coinvolge due cartelle: la directory di origine, ed il mountpoint. Ogni file nel mountpoint ha un file specifico nella directory di origine che corrisponde ad esso; il file nel mountpoint fornisce la visione non criptata di quello nella directory di origine. I nomi dei file sono criptati nella directory di origine.
- **WikipediaFS**: permette di leggere e modificare gli articoli di Wikipedia (o qualsiasi sito basato su Mediawiki) come se fossero veri file.
- **GmailFS**: permette di montare e usare lo spazio di posta elettronica di GMail come un disco rigido fisico locale.
- **CryptoFS**: un altro filesystem criptato con un principio di funzionamento simile ad EncFS.

Capitolo 3

L’architettura

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?

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 e messo all'interno di una struttura dinamica (una mappa). Ogni riga del file contiene una coppia chiave-valore della mappa.
Ci siamo posti anche la questione degli accessi concorrenti nel caso più file venissero aperti contemporaneamente; dopo un'accurata analisi, siamo giunti alla conclusione che il problema è inesistente, infatti la scrittura avviene su chiavi diverse della mappa.

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.

(RISOLVERE IL QUARTO PROBLEMA)

Un ulteriore problema da risolvere è 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 le seguenti: *mkdir(path, mode)*, *rmdir(path)*, *unlink(path)*, *rename(old, new)*, *create(path, flags, mode)*, *write(path, buf, offset, fh)*, *truncate(path, len, fh)* (per maggiori dettagli, si veda il paragrafo successivo).
Vengono aggiornati sia la struttura dinamica che il file, al fine di evitare la perdita di dati in caso di crash del sistema operativo.

(RISOLVERE IL SESTO PROBLEMA (?))

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 da sovrascrivere

In questo paragrafo, vediamo le operazioni di FUSE che è stato necessario sovrascrivere per far sì che il nostro filesystem gestisse correttamente gli hash.

Directory Operations

- *readdir(path)*: yield directory entries for each file in the directory

  L'equivalente di "ls". È necessario sovrascriverla perché oltre ai nomi dei files e delle cartelle si devono mostrare anche i relativi valori di hash.

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

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

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

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

File Operations

- *unlink(path)*: delete a file

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

- *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.