VUFUSE

Virtual File System in User Space

Presentata da: **Leonardo Frioli**

Relatore: **Prof. Renzo Davoli**

Alma Mater Studiorum - Università di Bologna
Scuola di Scienze
Corso di Laurea in Informatica

Anno accademico 2017-18
Sessione unica

Indice

Elenco delle figure

Introduzione

"*Virtual File System in User Space*" è un sottotitolo che si può definire "parlante" o "molto espressivo": in poche parole riesce a racchiudere i quattro argomenti chiave discussi in questo documento. Se scomposto nelle sue parti si ottiene: "virtualità", "file system" e "user space". Il filo che percorre tutta la trattazione è dato proprio dalla parola "filesystem" la quale, ovviamente, fa riferimento a una delle più conosciute e potenti astrazioni fornite dal kernel. Non si è, però, davanti ad un libro scolastico o universitario che ha lo scopo di illustrare il funzionamento di questo meccanismo, ma piuttosto si è di fronte ad un approfondimento sui file system inquadrati in un ambito specifico e non comune. Vengono in aiuto, nella definizione del contesto, le altre due "parti" del sottotitolo: "virtual" e "user space", ad indicare l'accezione data alla trattazione sui file system. Se si pensasse di dover dire quale delle due parole ha più importanza, probabilmente si sceglierebbe la prima, ma, come si vedrà più avanti, la loro rilevanza è alla pari e i due concetti sono *molto* legati tra di loro. Il quarto argomento a cui si accennava poco prima, non è nient'altro che l'unione di quelli appena discussi: si parlerà sì di file system, sì di file system virtuali e virtualità, sì di file system in user space, ma il corpo della trattazione sarà proprio dato dall'integrazione di tutti questi concetti e dalla dimostrazione di come possano convivere insieme.

Lo scopo sarà quello di argomentare come sia possibile avere dei file system a livello utente, ma in un ambiente virtuale. La trattazione cercherà di arrivare per gradi a questo punto passando prima per una descrizione dei singoli concetti appena visti, così che il lettore abbia un quadro completo sull'argomento preso in esame. Si parlerà della necessità di avere file system a livello utente, ovvero senza dover possedere un qualche tipo di privilegio; si discuterà della virtualità e di alcuni strumenti attualmente disponibili per lavorare in ambiente virtuale[1] e, infine, si entrerà nel vivo della argomentazione dimostrando come l'unione dei concetti prima visti trovi un effettivo riscontro nella realtà.

[1] Si fa riferimento ad un tipo di virtualizzazione parziale, dato che software come **qemu** o **virtualbox** non sono di interesse ai fini della trattazione.

Capitolo 1

Stato dell'arte

Nel presente capitolo saranno esaminati alcuni concetti fondamentali ai fini della trattazione tra cui quelli di file system, file system in user space, virtualizzazione e file system virtuali. Verranno affrontate varie problematiche riguardanti questi argomenti e si discuterà di alcune soluzioni e strumenti attualmente disponibili all'utente. Alla fine sarà chiaro che, nonostante molto sia stato fatto, c'è ancora spazio per l'innovazione.

1.1 File system

Si può affermare con sicurezza che il filesystem rappresenta un punto cardine per tutti quei sistemi operativi Unix-like. In un ambiente dove "tutto è file", il meccanismo di gestione e nomina dei file stessi ricopre un ruolo indispensabile e insostituibile. Se un processo deve accedere a un file può farlo solamente grazie al suo path e quindi solo attraverso il filesystem: tutti i processi *vedono* i file grazie a questa loro *comune* astrazione. Ogni file rientra nella struttura gerarchica e arboriforme fornita da questo meccanismo e ha un *pathname che lo identifica in modo assoluto.*

L'operazione principale che riguarda i file system è la syscall **mount**[1] che si occupa di sovrapporre o attaccare una gerarchia di file in un punto del filesystem.

```
$ sudo mount -o option -t filesystemtype device mnt/
```

La **mount** cambia la *visione* dei file presenti nel sotto-ramo del mountpoint (*mnt/* nell'esempio): se prima di questa operazione si ha la certezza di accedere a un determinato file *mnt/a*, successivamente ad essa non si potrà più essere sicuri di trovare lo stesso file percorrendo il medesimo pathname.

[1] Nonostante la traduzione in italiano non suoni molto bene, per riferirsi alle operazioni di mounting saranno usate espressioni come "fare mount","montare" e "eseguire una mount". Sfruttiamo la nota anche per sottolineare che nel testo saranno inevitabilmente presenti termini in lingua inglese che difficilmente trovano una corretta traduzione in italiano. La versione in inglese sarà quanto più possibile utilizzata, ma, laddove siano necessari sinonimi o giri di parole per evitare ripetizioni, si utilizzerà anche una terminologia italiana.

Non c'è da sorprendersi, quindi, della necessità dei privilegi da super utente per effettuare questo comando. L'utente non può fare mount di un file system in un generico punto. Mettendosi nei panni di un utente malevolo si potrebbe eseguire una **mount** su directory che contengono file essenziali per il corretto funzionamento e la sicurezza del sistema operativo, andandoli, in un certo senso, a sovrascrivere.

1.2 File system in user space

Relegare l'uso della **mount** all'amministratore porta con se pregi e difetti. I privilegi da super user sono necessari perché il codice dell'utility presa in esame è eseguito in *kernel mode* e i file system montati possono essere solo quelli supportati dal kernel stesso.

Sorgono ora alcune domande: Se l'utente avesse una sua immagine di un file system e volesse montarla su una directory di cui ha i permessi di accesso in scrittura, come potrebbe fare non essendo root? Se uno sviluppatore inventasse una nuova tipologia di file system e avesse necessità di montarlo, testarlo, sperimentarlo e, se si rivela utile, usarlo e distribuirlo, come potrebbe fare?

Risulta evidente la necessità di avere un modo per montare file system anche a livello utente, il che significa evitando (almeno in parte) l'esecuzione di codice in kernel mode. Montando in user mode si aumenta la *fault tolerance*: il malfunzionamento di un file system in kernel space può condurre a un potenziale *kernel panic*; in user space, può,nel peggiore dei casi, comportare l'uccisione del processo che gestisce il file system stesso. Un altro vantaggio è dato dalla maggiore flessibilità che ottiene l'utente: in molti casi non è giustificata la necessità di essere *root* per poter modificare delle immagini di file system su cui si hanno i diritti di accesso; inoltre eseguire comandi con i privilegi di amministratore è molto rischioso poiché basta sbagliare la sintassi o non essere pienamente consapevoli di cosa si stia facendo per compromettere il sistema. Agendo in user mode si ha maggiore libertà di sbagliare dato che si ha la sicurezza di non poter danneggiare file sui quali non si hanno diritti di scrittura.

1.2.1 FUSE: Filesystem in USErspace

La risposta al problema dei file system a livello utente è data da FUSE. Questo progetto, iniziato più di dieci anni fa ad opera di Miklos Szeredi [4], è riuscito a dare agli utenti la possibilità di creare e gestire i propri file system senza dover eseguire, se non in minima parte, codice del kernel.

FUSE è composto da un modulo kernel e da una libreria software **libfuse**. Il modulo è ormai standardizzato e da tempo è parte del kernel Linux, ma anche di altri sistemi operativi come FreeBSD e macOS etc... La libreria, invece, è in costante sviluppo e fornisce ai programmatori il supporto per implementare i propri file system. Le due parti sono in stretto legame tra di loro poiché sono collegate da un API (Application Programming Interface) di basso livello che ne permette la comunicazione. Dall'altra parte il modulo interagisce con il **Virtual**

Filesystem Switch (VFS) del kernel, mentre la libreria espone un'interfaccia di alto livello (presente nel file **fuse.h**) accessibile dai programmi a livello utente e necessaria per lo sviluppo dei sottomoduli.

Lo scopo principale di FUSE è di fornire un modo semplice per implementare, montare e usare i propri file system senza dover avere ulteriori privilegi se non quelli di accesso sull'immagine del file system. Un'immediata conseguenza di FUSE è stato lo sviluppo di molti sottomoduli (alcuni saranno analizzati più avanti) che danno all'utente la possibilità di eseguire la **mount** in un ambiente più sicuro senza però stravolgere la semantica del comando. L'operazione di mounting con FUSE rimane sintatticamente simile ad una normale **mount** e permette di attaccare una gerarchia di file a un mountpoint su cui l'utente ha i privilegi di accesso in scrittura: non è ovviamente consentito eseguire l'utility su cartelle che contengono file di sistema e sulle quali ha accesso solo il *super user* perché altrimenti si correrebbero gli stessi rischi di una normale **mount**. Dal punto di vista della semantica, c'è pochissima differenza tra la **mount** "normale" e quella con FUSE dal momento che in entrambi i casi è l'intero sistema ad essere avvertito dell'operazione. Eseguire una system call sul sottoramo montato in un modo o nell'altro è uguale dal punto di vista del processo, cambia solamente il modulo kernel interrogato dal VFS.

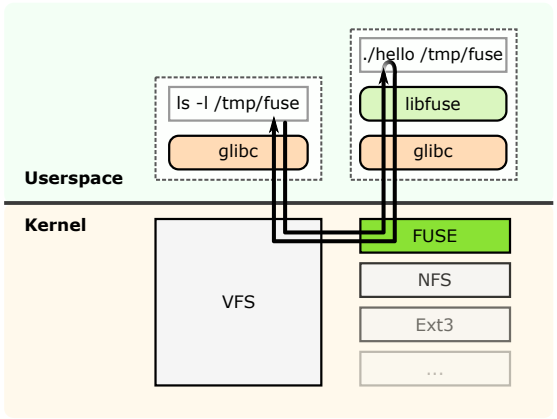

Figura 1.1: FUSE helloworld (da Wikipedia)

E' comodo, come al solito, partire da un *helloworld* [Figura 1.1] ovvero da un esempio base. Si supponga, in un primo momento, di avere il file system FUSE già implementato e montato in */tmp/fuse* e si esegua l'utility **ls**:

```
$ ls -l /tmp/fuse
```

Il comando è principalmente composto da tre system call: **open**, **getdents** e **close**. Le funzioni chiamate sono quelle della **glibc** che andrà ad eseguire il codice del VFS in kernel space. Il **Virtual Filesystem Switch**, accorgendosi che la cartella su cui vengono eseguite le operazioni è una di quelle coinvolte

nella **mount** di FUSE, passerà le chiamate al relativo modulo, il quale si occupa di "interrogare" il processo *./hello* e ottenere i risultati relativi ad ogni system call.

Discutendo ora l'implementazione di *hello* sarà più chiaro il significato del verbo "interrogare" prima utilizzato. Il file in questione fornisce una implementazione per ognuna delle system call necessarie al funzionamento del file system montato ed espone le loro interfacce al modulo kernel grazie a un'apposita struttura fornita dalla **libfuse**. Il programma, infine, si occuperà di eseguire la **mount**. L'operazione appena citata è quella più critica, ma è svolta interamente dalla libreria di FUSE e il programmatore non dovrà far nient'altro che passarle le opportune strutture dati (tra cui quella delle syscall) e le informazioni sulla **mount** stessa (il mountpoint in particolare). La chiamata necessaria per avvertire il modulo FUSE della **mount**(fornita dalla libreria), non porta alla terminazione del processo chiamante bensì lo lascia in un loop attendendo di ricevere le "interrogazioni" a cui si accennava prima. Quando sarà necessaria una syscall sul file system FUSE montato, vi risponderà il processo *./hello* eseguendo la sua implementazione della chiamata a sistema.

1.2.2 Esempi di file system basati su FUSE

La bellezza di FUSE sta nelle infinite potenzialità e nella semplicità di sviluppo e distribuzione del codice dei sottomoduli. Anche nel kernel è possibile aggiungere diverse tipologie di file system, ma il procedimento risulta essere molto più complesso della scrittura di un sottomodulo per FUSE. Espandere il kernel, così che possa supportare una nuova tipologia di file system, richiede che il codice aggiunto sia robusto, sicuro, tollerante ai guasti, ampiamente testato etc... . Un sottomodulo per FUSE necessita solo di essere scritto e testato, al resto ci pensano la libreria e il modulo kernel. Un sottomodulo FUSE è un programma che permette all'utente l'interazione con un particolare tipo di file system. Come si è visto con l'*helloworld* della sottosezione precedente, la struttura di tutti i moduli è simile : essi devono ridefinire le system call legate ai file ed eseguire la **mount** attraverso la **libfuse**. C'è, però, la possibilità di creare varie tipologie di file system, oltre a permettere di utilizzare quelli più comuni. I sottomoduli FUSE sono moltissimi[2] e la maggior parte fornisce delle funzionalità aggiuntive rispetto a quelle date dai file system più noti. Ecco alcuni tra i molti file system che sfruttano la **libfuse**. I primi sono sottomoduli che permettono di montare file system conosciuti, ma non per questo inutili; gli altri estendono le possibilità di utilizzo del file system stesso.

fuse-ext2 Probabilmente uno dei più popolari ed utilizzati, permette di montare file system di tipo ext2. Viene molto usato da utenti Apple che hanno necessità di interfacciare le loro macchine con dispositivi provenienti da ambiente Unix.

[2] Un elenco esaustivo può essere trovato all'indirizzo https://github.com/libfuse/libfuse/wiki/Filesystems

fusefat Utile per il **mount** di file system fat. Punto cardine per il suo funzionamento è la **libfat** che serve proprio ad interagire con questo tipo di file system. Si trova ancora in una fase di sviluppo e necessita di lavoro per giungere ad una versione stabile.

fuseiso9660 Semplicissimo modulo per montare in lettura file system di tipo iso.

encfs Permette di crittografare delle directory ed accedervi in chiaro attraverso l'operazione di **mount**. Mentre gli altri sottomoduli esaminati riguardano file system di uso comune, **encfs** mostra come sia possibile, grazie a FUSE, la creazione di nuove e diverse tipologie di file system.

sshfs Consente di montare e interagire con directory remote utilizzando SFTP (Secure File Transfert Protocol) attraverso l'uso di un client.

loggedfs Può tracciare ogni operazione eseguita sul file system.

1.3 Virtual Square V^2

Saranno introdotte, in questa sezione, le idee che stanno alla base del progetto Virtual Square e i vari strumenti da esso forniti. Si avrà così una più chiara panoramica sullo stato d'arte attuale riguardante i file system virtuali a livello utente. Le citazioni e gli argomenti qui presenti fanno riferimento al manuale ufficiale di Virtual Square [1].

Il progetto nasce con l'obbiettivo di fornire agli utenti "un contenitore di strumenti sulla virtualità". Parafrasando il libro, definiamo *virtuale* un'astrazione di qualcosa che può essere *effettivamente* usata al posto della cosa stessa[3]. Per esempio un file system virtuale fornisce ai programmi la stessa interfaccia di quello reale così che i processi possano utilizzarlo in egual modo senza notare la differenza.

Il lavoro, iniziato nel 2004, ha portato ormai allo sviluppo di molti strumenti legati al mondo della virtualità. Sono due le principali aree di interesse coperte da Virtual Square: la prima riguarda l'ambito delle reti e del networking[4], la seconda si concentra sulla virtualizzazione a livello di processo. Volendo anticipare la prossima sezione si dirà che è proprio la seconda area più di interesse per il documento in quanto fortemente legata ai file system virtuali.

1.3.1 ViewOS

ViewOS (Operating System with a View) copre una delle due aree di interesse del progetto Virtual Square. Il termine è usato per indicare qualcosa di molto

[3] Riguardo la definizione di *virtuale* e una più esaustiva spiegazione del suo significato si consiglia la lettura del manuale stesso

[4] Virtual Distributed Ethernet (VDE)

vasto: ViewOS non è solo il raggruppamento di alcuni strumenti software, ma è anche un'insieme di idee e scelte progettuali. Viene usata la parola 'Operating System' poiché gli strumenti forniti coprono tutti gli ambiti di un sistema operativo: rete, file, file system, device, userid e groupid etc... Il fine ultimo è sempre quello di estendere le possibilità di uso che un utente ha, ma anche di implementare un software che non trova eguali nell'ambito della virtualià sul piano delle funzionalità che offre.

L'idea alla base di ViewOS è di dare ai processi visioni differenti del mondo in cui si trovano (da ciò deriva la presenza di 'View' nel nome). "Ogni processo condivide la stessa visione uniforme (network, device, file system ...) del sistema in cui è eseguito[5]". Gli unici modi per dare a due processi visioni diverse dell'ambiente di esecuzione sono: o tenerli su due macchine differenti, o eseguirli in una macchina virtuale, soluzioni che risultato dispendiose e inefficienti. ViewOS fornisce gli strumenti per liberare i processi dalla *general view assumption*, in altre parole la comune[6] visione del mondo che hanno: percorrendo lo *stesso* path si accede al medesimo file, *stessa* percezione della rete, *stessa* visione dei device etc... .

La soluzione proposta va nella direzione di una virtualizzazione parziale, ovvero non viene completamente virtualizzato tutto l'ambiente in cui si trova il processo (come accade per le più comuni macchine virtuali), ma si virtualizza solo ciò che è necessario e solo ciò di cui il processo ha bisogno. Dato che l'interazione tra il sistema (garante della *general view assumption*) e i processi avviene attraverso le system call, saranno proprio le chiamate a sistema ad essere virtualizzate. La diretta conseguenza di questa idea sta nell'implementazione di una **System Call Virtual Machine** (SCVM), una macchina virtuale che ha il compito di intercettare, contestualizzare e virtualizzare le system call. ViewOS non punta alla creazione di una generica SCVM, ma ricerca un qualcosa che soddisfi i requisiti di *modularità* e di *usabilità a livello utente* che sono tra i punti cardine della filosofia alla base di Virtual Square. Questi due concetti saranno ripresi una volta visto come è stata data risposta all'esigenza di avere processi con viste differenti.

1.3.2 Umview

Nel paragrafo precedente si era detto, anticipando questa parte, che l'idea di ViewOS ha portato all'implementazione di una SCVM. Il manuale di Virtual Square[1] definisce **umview** come una "macchina virtuale parziale (ParVM) ovvero una System Call Virtual Machine che fornisce lo stesso insieme di system call dato dal kernel ospitante". Nella sezione corrente si cercherà di mostrare la macrostruttura di questo tool soffermandosi sulle parti più interessanti al fine della trattazione.

L'intero progetto ViewOS è costituito da un core o hypervisor e da vari moduli: il termine **umview** indica propriamente la parte core mentre i moduli,

[5] Si fa riferimento al Capitolo 7 del manuale di Virtual Square[1]
[6] inteso come condivisa, comune ad entrambi.

nominati in modo simile (**umfuse**, **umdev**...), non costituiscono la parte centrale ma la vanno a completare e arricchire. E' possibile paragonare la struttura di ViewOS con quella di FUSE. Il core (**umview**) ha uno scopo simile a quello della **libfuse**/modulo FUSE del kernel: fornire il supporto per l'implementazione e l'esecuzione dei moduli. Come si vedrà più avanti l'hypervisor, se usato da solo, apparentemente non fa nulla, ma è grazie ai moduli che l'utente riesce a sfruttarne le funzionalità. Anche in FUSE accade che il sottomodulo sia essenziale per fornire all'utente la possibilità di interagire con varie tipologie di file system.

La struttura data a ViewOS porta in evidenza l'idea di modularità che sta dietro al progetto di Virtual Square: all'utente viene fornito uno strumento base (ovvero **umview**) che può espandere sia aggiungendo moduli già implementati, sia implementandone di suoi. Oltre ad essere modulare, **umview** è interamente implementato in user space (il prefisso 'um' sta proprio per *user mode*) quindi non è necessario alcun privilegio da amministratore per utilizzarlo.

Il core funge da ParVM in quanto attraverso il meccanismo della *ptrace* ha il compito di intercettare le system call. L'hypervisor si limita a catturare le chiamate a sistema e decidere se passarle ad un modulo specifico o farle eseguire al kernel (quindi non virtualizzarle). Se **umview** viene usato da solo le chiamate a sistema saranno comunque prese, analizzate e contestualizzate ma, non essendoci alcun modulo a cui farle gestire, il core deciderà di passarle al kernel non compiendo alcun tipo di virtualizzazione effettiva. I moduli dovranno, invece, ridefinire le system call che vogliono virtualizzare e registrarsi presso il core così che possano essere chiamati quando viene intercettata una tra le system call da essi re-implementate. In questo modo saranno i moduli a eseguire e rispondere alle system call, facendo il lavoro che di solito spetta al kernel.

Ci sono vari modi per far sapere al core quali chiamate il modulo vuole che siano virtualizzate. Il metodo più semplice è dire esplicitamente all'hypervisor le singole system call di cui il modulo si occuperà, ma è anche possibile informalo di voler gestire gruppi di chiamate correlate. Si prenderà in esame proprio il secondo caso dato che la virtualizzazione del file system, e quindi delle system call ad esso legate, è di maggiore interesse.

Giunti a questo punto dovrebbe essere chiara l'importanza del meccanismo di nomina dei file e il ruolo svolto dal file system. Non risulterà strano, quindi, venire a sapere che per poter cambiare la visione che i processi hanno è proprio coinvolto il filesystem e, non a caso, l'operazione di **mount**. Con la ridefinizione e virtualizzazione della **mount** da parte di un modulo, è possibile indicare al core di demandare la gestione delle system call riguardanti il ramo montato al modulo stesso. Così facendo, i processi che si trovano nella ParVM avranno una diversa visione di quella porzione di file system rispetto ai processi che non sono virtualizzati.

E' possibile ora chiarire il significato di virtualizzazione : i processi, "tracciati" dal core, sono ignari di tutto il meccanismo che sta dietro la gestione delle loro system call. La virtualizzazione è effettiva dal momento che il processo non si accorge di trovarsi in ambiente virtuale e perché mantiene inalterata l'interfaccia delle system call: il processo esegue le stesse chiamate a sistema

che farebbe nella realtà e riceve risultati coerenti alla funzione invocata. Viene da se anche il significato di *Partial VM*: l'hypervisor non crea un ambiente che sostituisce completamente il kernel, ma è specifico per *alcune* system call (quelle ridefinite dai moduli) di *alcuni* processi (quelli "seguiti" dalla *ptrace*).

Per completare[7] questa parte è bene menzionare anche **kmview**, dove 'km' sta per kernel mode, il quale è identico a **umview** per le funzionalità proposte e le idee che vi sono alla base; ciò per cui differisce è la necessità di installare un modulo kernel per consentire un aumento di performance rispetto alla versione in user mode.

1.3.3 Umfuse

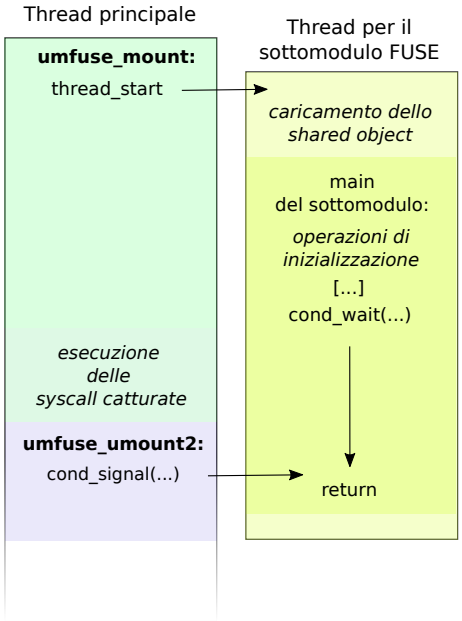

Figura 1.2: **umfuse_mount** e **umfuse_umount2**[8]

In questa parte sarà analizzato uno dei moduli più rilevanti per **umview**, ovvero **umfuse**(user mode fuse). Inizierà ad essere più chiaro come i concetti di virtualità, file system e user space possano convivere nello stesso strumento

[7] Una descrizione esaustiva del funzionamento di **umview** non è il compito di questo elaborato, ma , come ormai più volte citato, rimandiamo al manuale ad esso dedicato[1]. Altri dettagli riguardo **umview** potranno comunque essere ritrovati nel corso della trattazione. Essendo, quella discussa, la prima versione di ViewOS e dovendo prendere in esame anche quella più attuale, inevitabilmente saranno presenti paragoni tra la vecchia implementazione e la nuova, in cui vari meccanismi e scelte progettuali saranno messi a confronto e quindi descritti.

[8] Lo schema è una semplificazione della figura 10.5 di pagina 192 del manuale di Virtual Square[1].

software. **Umfuse**, infatti, è il modulo che permette la virtualizzazione delle chiamate legate al file system, ma soprattutto è quel modulo che fa da ponte tra **umview** e FUSE. **Umfuse** esprime al meglio l'idea alla base di ViewOS dando effettivamente la possibilità di avere processi che lavorano in ambienti differenti, ovvero permettendo ad un processo di montare e lavorare con un file system in maniera trasparente agli altri. Oltre ad essere un modulo chiave per ViewOs, lo è anche per questa trattazione poiché riesce a portare la virtualità nell'ambito dei file system a livello utente, ponendosi come esempio (*proof of concept*) della possibile unione di questi tre concetti.

Umfuse è compatibile con i vari moduli FUSE, poiché mantiene inalterata l'interfaccia della **libfuse**. C'è un unica differenza: per usare i moduli FUSE da soli è necessario compilarli come eseguibili, mentre per utilizzarli assieme a **umfuse** devono essere degli *shared object (.so)*. La compilazione del sottomodulo come file .so consente di caricarlo dinamicamente e quindi di avere la possibilità di ridefinire alcune delle funzioni implementate dalla **libfuse** ed esposte tramite l'API di alto livello presente in **fuse.h**. Se si mantiene la stessa definizione di funzione, durante in linking dinamico, quando si cercherà la procedura con un determinato nome, verrà scelta quella "più recente". Ciò consente, come si vedrà tra poco, di chiamare al posto di alcune funzioni di FUSE delle procedure implementate in **umfuse**.

Procedendo con la descrizione del funzionamento del modulo sarà ancora più semplice capire la relazione con FUSE e aiuterà anche a chiarire aspetti di **umview**.

```
$ umview bash
$ um_add_service umfuse
$ mount -o rw+ -t umfuseext2 ext2.img mnt/
# ora le system call che coinvoglono mnt/ sono passate al modulo
```

Il modulo **umfuse**, riceve il compito di gestire la **mount** e, successivamente, tutte le system call legate al sotto-albero che ha *mnt/* come radice. Durante l'esecuzione della **umfuse_mount**[9] viene caricato lo *shared object*(.so) relativo al modulo FUSE specificato con l'opzione '*-t*' (**umfuseext2** nell'esempio) e gli vengono fatte eseguire tutte le operazioni di inizializzazione [Figura 1.2]. In **umfuse** sono ridefinite le funzioni della **libfuse** che gestiscono il mounting e la comunicazione con il kernel, così che, sruttando il linking dinamico, il sottomodulo FUSE non chiami l'implementazione di quelle funzioni fornita dalla libreria, ma esegua le procedure ridefinite nel modulo. Il processo, quindi, non viene sospeso aspettando le richieste del modulo kernel, ma viene fatto attendere su una *pthread_cond_t*, per poi essere sbloccato con la funzione **umfuse_umount2**[10]. Nel periodo che trascorre fra la **mount** e la **umount**, le system call relative al sotto-ramo di *mnt/* vengono catturate dal core, passate a **umfuse** e rigirate al sottomodulo FUSE attraverso la stessa interfaccia che quest'ultimo espone

[9] Nome dato nel file **umfuse.c** alla funzione che ha il compito di virtualizzare l'operazione di **mount**.

[10] Vale lo stesso discorso della nota precedente: quella citata è la funzione che gestisce la system call **umount** catturata dall'hypervisor.

al kernel. E' quindi il sottomodulo, grazie a **umfuse**, a rispondere al processo invece che il sistema operativo.

1.3.4 VuOS

Finora si è parlato del progetto Virtual Square, dell'idea dietro ViewOS e della sua implementazione attraverso l'hypervisor **umview** e i vari moduli (**umfuse** in particolare). Con ViewOS si è dimostrato che è possibile dare ai processi viste differenti e che un modo per farlo sta nella virtualizzazione del file system e della **mount**. Gli strumenti descritti sono stati rilasciati ormai alcuni anni fa ed è quindi nata la necessità di rinnovarli e di riscriverli così da avere un nuovo ViewOS, ora chiamato VuOS (in inglese la sillaba 'vu' si pronuncia in modo simile alla parola 'view'). Il nuovo ViewOS assomiglia al vecchio per la struttura e le idee alla base[11], ma si pone in una veste aggiornata e con alcuni miglioramenti da analizzare. Con VuOS si da un interesse ancora maggiore alle scelte progettuali perché, a differenza di prima, si parte da una base solida (**umview** e i moduli) con la consapevolezza che quel tipo di virtualizzazione parziale e gestione delle syscall è possibile.

1.3.5 Umvu

E' facilmente intuibile capire che **umvu** è la nuova versione di **umview**. Il concetto rimane sempre lo stesso: una ParVM implementata come una SCVM attraverso un componente core e vari moduli. Tutte le scelte progettuali fatte nella prima versione sono state messe in discussione. Ciò ha portato a lasciare, quasi invariate, alcune delle strutture dati e procedure utilizzate nella vecchia versione, ma a cambiare tutti quei meccanismi che erano migliorabili.

Una delle più grandi differenze riguarda il core stesso. In **umview** l'intercettazione delle system call avveniva attraverso la *ptrace* ed era gestita da un *main event loop* unico per tutti i processi virtualizzati. Ora con **umvu**, oltre al meccanismo della *ptrace*, si sfrutta anche la potenza della libreria *pthread* e del *multithreading*. Ognuno dei processi virtualizzati ha un suo hypervisor (o per meglio dire "angelo custode"), che si occupa di catturare le system call fatte dal processo stesso e passarle ai moduli.

Un altro cambiamento sta nel rapporto tra il core e i moduli. Non solo l'interfaccia esposta dal core è migliore e più semplice, ma anche l'importanza data ai moduli è stata accentuata. Anche con **umvu**, il componente core può poco da solo, ma deve essere completato attraverso l'inserimento di moduli. Il supporto fornito dall'hypervisor ai moduli è notevolmente migliorato poiché si cerca di avere una gestione quanto più uniforme per evitare che gli sviluppatori debbano implementare qualcosa al di fuori dello scopo del modulo.

Il modello che si vuole perseguire con VuOS rispecchia in parte quello del microkernel: avere un core che fornisca il miglior supporto possibile ai moduli, ma

[11] I concetti appena visti nelle sezioni precedenti non saranno quì riscritti perché non avrebbe senso ripetere due volte le stesse argomentazioni. Saranno mostrati solo quali sono i cambiamenti utili e interessanti per la trattazione.

che allo stesso tempo sia snello e non si faccia carico di implementare funzionalità che potrebbero essere espresse dai moduli stessi. Un buon esempio riguarda la virtualizzazione dello *user id* e del *group id* che passa dall'essere gestita nel core di **umview**, con vari escamotage poco eleganti, all'essere amministrata da un modulo specifico (**unrealuidgid**[12]).

In via definitiva si può affermare che **umvu** prende il suo predecessore come forte esempio per giungere a una versione più robusta, strutturata ed efficiente.

1.3.6 Vumodules

Si entra ora in una questione del tutto aperta. Mentre **umview** è stato rinnovato e la sua nuova versione è stata rilasciata ed è stabile, lo stesso non si può dire per i moduli. **Umvu** è stato pubblicato da poco, pertanto i moduli della vecchia versione non trovano ancora il loro corrispettivo nella nuova.

Al momento sono presenti moduli di test del core (per citarne alcuni: **mountreal**, **unreal**) e un paio di moduli veri e propri (**unrealuidgid** e **vunet**). Il testing del core avviene attraverso dei moduli particolari che reimplemntano le system call, però senza dare alcun tipo di virtualizzazione. Il modulo agisce invocando le normali chiamate a sistema, così che, alla fine, tutte le operazioni fatte vengono eseguite realmente, ma con la differenza che c'è stata prima l'interposizione del core e del modulo di test.

L'aggiornamento dei vecchi moduli alla nuova versione risulta necessario per vari motivi: sfruttare le nuove funzionalità esposte dal core e stare al passo con gli altri progetti correlati a VuOS. L'esempio più importante è dato da FUSE stesso che con il passare del tempo è stato aggiornato e sviluppato.

1.4 What's next?

In questo capitolo si è parlato dell'importanza dei file system; della estrema utilità di FUSE e di Virtual Square che con ViewOS/VuOS si pone l'obiettivo di avere dei processi staccati da quella che è chiamata *general view assumption*. Si è visto, inoltre, che un punto di incontro tra tutti questi concetti è dato da **umfuse** il quale lega assieme **umview** con gli svariati sottomoduli per FUSE.

Gli utenti hanno pertanto a disposizione gli strumenti sia per montare file system senza privilegi, sia per cambiare la vista dei processi montando virtualmente file system di tipo FUSE (attraverso **umview** e **umfuse)**. E quindi? Cosa c'è poi?

Sembrerebbe non mancare nulla, se non fosse che sia FUSE, sia ViewOS hanno subito aggiornamenti e modifiche, e, se le strade dei due progetti si erano incrociate con **umfuse**, ora si sono di nuovo allontanate. Molto forte e sentita è quindi la mancanza del successore di **umfuse**, ovvero di un modulo per il nuovo **umvu** che si interfacci con le più recenti versioni di FUSE.

[12] Per essere precisi il modulo si occupa anche della gestione del *saved user id* e del *file system user id*.

Capitolo 2

Contributo originale

Si è giunti al cuore della trattazione, punto in cui si cercherà di dare una risposta alle necessità evidenziate nel capitolo precedente e si presenterà quella che al momento è la soluzione più attuale nell'ambito dei file system virtuali in user space. Tutto quello di cui si è parlato fino ad ora sarà di fondamentale importanza per comprendere questo capitolo e alcuni degli strumenti appena visti troveranno qui una spiegazione più dettagliata del loro funzionamento. Il centro del capitolo sarà una discussione sulle scelte progettuali effettuate e le difficoltà incontrate nell'implementazione di **vufuse**, ovvero la versione di **umfuse** che si basa sul nuovo core **umvu**. Nella descrizione del modulo inevitabile sarà il confronto con la versione precedente; verranno spiegati meglio alcuni meccanismi presenti in **umvu**, ma anche in **ViewOS**, permettendo di capire il perché di alcune scelte prese.

Infine si parlerà di un piccolo sottomodulo di FUSE, **vufusereal**, creato appositamente per **vufuse** al fine di agevolare il debugging di questo modulo e separare le complessità strutturali.

2.1 Vufuse

Se **umvu** è la nuova versione di **umview**, allora è inutile dire che **vufuse** è la versione aggiornata di **umfuse**. Proseguendo con il paragone, anche **vufuse**, come **umvu**, mantiene gli stessi scopi e funzionalità della vecchia versione: dare all'utente uno strumento che gli consenta di montare virtualmente (nell'ambiente definito da **umvu**) file system di tipo FUSE. L'implementazione di **vufuse** si basa fortemente su quella del suo predecessore. Nonostante si cerchi di migliorare alcune tecniche usate in **umfuse**, il nuovo modulo comunque sfrutta meccanismi e scelte progettuali di quello vecchio conservandone lo scopo: legare e interfacciare VuOS con FUSE, e FUSE con VuOS.

Già con **umfuse** si aveva la prova della possibilità di far convivere assieme i concetti di "virtualità", "file system" e "user space", ora con **vufuse** si da all'utente uno strumento efficace per lavorare con le proprie immagini di file

system senza aver vincoli di qualche genere. Si ricordi inoltre la volontà di dare ai processi visioni differenti dell'ambiente, la quale è ereditata da ViewOS e la cui implementazione è realizzata sopratutto tramite questo modulo.

Osservazione **Vufuse**, come accadeva per **umfuse**, è in forte relazione con i sottomoduli di FUSE, alcuni dei quali citati nella sezione 1.2.2. Gli esempi che saranno riportati riguarderanno soprattutto **fuse-ext2** (caricato come *shared object* con il nome **vufuseext2**) dato che è uno dei sottomoduli più importanti e più completi tra quelli visti. I test e gli esperimenti sono comunque stati fatti anche con **fuseiso9660**, **fusefat** e con **vufusereal** (trattato più avanti). Alla fine risulterà chiaro comunque che la strutturazione del modulo permette a **vufuse** di interfacciarsi anche con altri sottomoduli di FUSE, come accadeva per il suo predecessore.

2.1.1 Progettazione e scelte implementative

E' bene presentare subito come **vufuse** si collochi tra **umvu** e FUSE mostrando la sua forte funzione di collegamento tra le due interfacce [Figura 2.1].

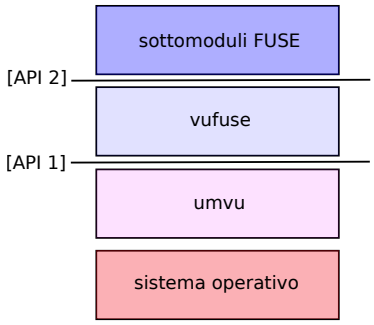

Figura 2.1: Collocazione del modulo vufuse

Il modulo preso sotto esame dovrà quindi conformarsi al nuovo **umvu** e ai sottomoduli di FUSE. Le due interfacce sono così da intendersi: [API 1] è l'insieme di funzioni messe a disposizione dal core attraverso l'header file **vumodule.h**; [API 2] è l'interfaccia data dai sottomoduli FUSE e determinata dalla **libfuse** (che è in costante aggiornamento). L'approccio all'implementazione ha visto una prima fase in cui si è cercato di conformare il modulo all'interfaccia di **umvu**([API 1]), conservando la stessa versione dei sottomoduli FUSE che usava **umfuse**; in un secondo momento è stata presa in esame la **libfuse** e i suoi eventuali cambiamenti, in modo da rendere **vufuse** compatibile con una delle ultime versioni di questa libreria (la 2.9).

Dopo aver ragionato su come affrontare l'implementazione del modulo, si è discusso riguardo quali meccanismi di **umview** andassero mantenuti e quali altri modificati.

Vu_vufuse_mount

La struttura della colonna portante del modulo, ovvero la funzione **vu_vufuse_mount**, è stata fortemente condizionata dalla sua versione precedente (**umfuse_mount**). Entrambe condividono, infatti, la stessa tecnica per la gestione dei sottomoduli FUSE. All'atto della **mount**, processata dal modulo, verrà creato un nuovo thread sul quale si esegue il processo FUSE che ha il compito di gestire il file system. Una volta completata la fase di inizializzazione delle varie strutture dati, il processo rimarrà in sospeso su una *mutex*[1] per poi essere risvegliato dalla **umount**. Durante la fase di "sospensione", come avveniva nella vecchia versione, tutte le system call dirette al ramo del *mountpoint* sono catturate dal core e passate al nuovo modulo che chiamerà le funzioni relative al file system montato esposte dal sottomodulo FUSE.

Vufuse mount flags

Differentemente a quanto visto per il momento, alcune scelte fatte in **umview** sono state scartate o, comunque, rivalutate e cambiate. Mano a mano che si proseguirà nella trattazione, i cambiamenti verranno spiegati e commentati. Per ora si analizzerà il tema delle flag rese disponibili per la **mount** di **vufuse** e in particolare la flag FUSE_HUMAN.

Con **umfuse** era possibile eseguire l'utility **mount** con delle flag aggiuntive rispetto a quelle che possono essere comunemente usate. Si poteva, in un certo senso, personalizzare l'uso di **umfuse**. Per la nuova versione si è pensato di semplificare e diminuire il numero delle opzioni possibili. In questo modo si mantiene il codice più pulito, ma si lascia comunque aperta la possibilità di aggiungere nuove flag successivamente qualora si senta il bisogno di dare all'utente più libertà di personalizzazione. Vedremo due esempi di opzioni che erano presenti e ora sono state rimosse. Nel primo caso, la flag non necessiterà più di essere aggiunta; nel secondo caso, si pensa che al momento non serva, ma potrà, eventualmente, venir re-inserita.

FUSE_DEBUG Usata per abilitare le scritte di debugging, ora risulta inutile dato che il nuovo core fornisce un supporto diverso e più generale al debug.

```
printkdebug(F,"Messaggio di debugging");
```

Nel codice di **vufuse** sono presenti stampe di debug nella forma appena vista e la loro gestione è demandata a **vudebug**, un'utility di **umvu**.

```
$ vudebug -F
```

[1] Si fa riferimento alla *pthread_cond_t* accennata spiegando lo stesso meccanismo in **umfuse** (sottosezione 1.3.3).

Quando ci si trova in ambiente **umvu** è possibile abilitare il debugging attraverso questo comando più generale, in quanto basta cambiare l'opzione di invocazione per avere una stampa di controllo differente: nel nostro caso '**-F**' è stata scelta per riferirsi a **vufuse**, ma, per esempio, con '**-s**' si stampano in output tutte le system call catturate.

FUSE_HUMAN In **umview** l'utente è di default *root* della sua macchina virtuale, quindi non vengono fatti i controlli sull'accessibilità dei file virtuali, ma tutto è concesso in quell'ambiente. C'è, però, la possibilità di attivare una HUMAN MODE[2], che abilita i vari controlli.

Con **vufuse**, si è discusso se mantenere o no questa possibilità ed è stato momentaneamente scelto di toglierla, lasciando l'utente nella modalità in cui "tutto può". Se l'opzione fosse stata mantenuta, si sarebbe corso il rischio di avere un'implementazione poco chiara. Molto importante per la scrittura del codice e delle funzioni è avere una separazione dei compiti e implementare delle procedure che non si occupino di troppe cose. Eliminando la flag si evita che le funzioni che ridefiniscono le syscall debbano, oltre a gestire la compatibilità con i sottomoduli FUSE, anche occuparsi del controllo degli accessi. Si evita, inoltre, di avere molto codice ripetuto non solo all'interno dello stesso modulo, ma ance tra moduli differenti.

Si pensa alla possibilità di reintrodurre una forma di controllo sugli accessi ma attraverso l'uso di un modulo specifico simile ad **unrealuidgid**. Questa scelta è ancora da valutare, ma andrebbe a togliere ai moduli il peso di dover effettuare i vari controlli e fornirebbe all'utente un modo più chiaro e pratico per utilizzare la "human mode".

Unlink system call

Un altro dei meccanismi di **umfuse** sui quali si è discusso, riguarda la gestione della system call **unlink**. Prima di passare a vedere nel dettaglio le scelte prese in merito, è necessario leggere una parte della pagina di manuale della system call citata : "la **unlink**() elimina un nome dal file system. Se quel nome era l'ultimo link a un file *e nessun processo aveva il file aperto*, il file è eliminato e lo spazio, che occupava, reso disponibile al riuso. Se quel nome era l'ultimo link a un file, *ma un qualche processo ha ancora il file aperto*, il file sarà cancellato solo dopo che l'ultimo file descriptor, che si riferisce a lui, verrà chiuso."

FUSE per rispettare questa semantica fa in modo che, se viene eseguita una **unlink** su un file aperto da qualche processo, il file venga rinominato in modo opportuno. Così facendo, se qualche altro processo cerca di accedere al file con il suo nome originale non lo troverà, ma chi lo ha aperto potrà comunque continuare ad utilizzarlo. In **umfuse**, per rispettare la semantica della **unlink** si utilizza un meccanismo ispirato a quello di FUSE. Ogni volta che un processo apre un file, viene inserito, in una hash table, un elemento che si riferisce a

[2] E' possibile definire la modalità di default come "divina", dato che permette di fare tutto, e l'altra come "umana" o "terrena".

quel file e ne mantiene il nome da utilizzare per accedervi attraverso le varie operazioni di read e write (o comuque system call che necessitino del pathname di quel file). Qualora ci trovassimo nel caso della **unlink** preso sotto esame, si verifica cercando nella hash table se qualche processo ha quel file aperto e, in caso di successo, l'operazione di rinomina è fatta cambiando il campo relativo al path nell'elemento della tabella.

Si è molto ragionato sulla possibilità di cambiare o eliminare il meccanismo preso in esame, ma si è giunti successivamente alla conclusione di inserirlo anche in **vufuse** dato che la tecnica usata rispecchia proprio quella adottata da FUSE stesso. Inizialmente si pensava di rimuovere questo meccanismo dato che sembrava troppo dispendioso l'uso di una ulteriore hash table. Il core, infatti, utilizza internamente varie tabelle hash, tra cui anche una specifica per i file aperti. Legittimo era chiedersi se la stessa tecnica o una migliore potesse essere implementata sfruttando l'hypervisor. Due soluzioni erano plausibili: o lasciare al core il dovere di occuparsi da solo dell'eccezione della **unlink**, oppure fare in modo che il core fornisse il supporto ai moduli per gestire questa particolarità. Dal momento che il problema riguarda principalmente **vufuse** non sarebbe stato del tutto opportuno lasciare la sua gestione al core poiché avrebbe portato all'esposizione di una feature di cui però solo un modulo ha effettivo bisogno. In sostanza ci sarebbero stati più svantaggi che vantaggi nel togliere la tabella. Lasciando la hash table si ha un meccanismo già sperimentato e implementato che non influisce pesantemente nè sulle prestazioni del codice nè sulla sua complessità e leggibilità.

Organizzazione del codice

Concludiamo la sezione parlando di qualcosa di più leggero: la suddivisione e l'organizzazione in file del progetto. Molta importanza è stata data anche a come raggruppare le varie funzioni implementate. Se in **umfuse** si tendeva ad avere grandi file con dentro svariate procedure, con la nuova versione si è cercato di dare maggiore separazione alle funzioni, le quali sono state raggruppate in file diversi a seconda del loro scopo e della loro semantica. Solo per citare un esempio, nella vecchia versione si aveva un unico file **umfuse.c** che raggruppava sia le funzioni per il caricamento del sottomodulo FUSE sia le ridefinizioni delle system call, mentre ora si hanno due file separati, **vufuse.c** e **vufuseop.c**.

2.1.2 Difficoltà incontrate e Soluzioni adottate

Nonostante **vufuse** si basi molto su **umfuse** e ne tragga molto ispirazione, durante la fase di sviluppo sono sorte alcune problematiche che necessitano di essere discusse. Le difficoltà qui presentate riguardano quella che, nella [Figura 2.1] della sezione precedente, è chiamata [API 1], ovvero l'interfaccia esposta dal nuovo core.

Inizializzazione della variabile *static __thread struct vuht_entry_t**

Per discutere di questa problematica è necessario fare un passo indietro e andare a spiegare, più nel dettaglio, come **umvu** si occupa della scelta dei moduli per la gestione delle system call.

Ogni modulo, una volta inserito, andrà a popolare una tabella hash presente nel core[3]. Il modulo può inserire uno o più elementi nella tabella, i quali sono delle strutture dati che si contraddistinguono principalmente per due campi: *void ** obj; *uint8_t* type (dei quali sarà chiaro tra un attimo il significato). L'hypervisor utilizza poi la hash table per capire a quale modulo dover far gestire la system call che ha catturato. Per comprendere il funzionamento di questo meccanismo saranno usati un paio di esempi che coinvolgono **vufuse**.

Esempio 1

```
$ umvu bash
$ vu_insmod vufuse
# nel codice dell'hypervisor comporta l'esecuzione di
# vuht_add(uint8_t type, void *obj, ...):
vuht_add(CHECKMODULE, "vufuse", strlen("vufuse"), service, ...);
```

Come si può vedere nell'esempio 1, l'inserimento del modulo porta all'aggiunta nella hash table di un elemento relativo a **vufuse**.

Esempio 2

```
# continua dall'Esempio 1
$ mount -t vufuseext2 ext2.img mnt/
# nel codice della vu_vufuse_mount si esegue
# vuht_pathadd(uint8_t type, const char *source, \
# const char *path, const char *fstype,...) :
vuht_pathadd(CHECKPATH, "ext2.img", "mnt/", "vufuseext2",...);
```

Appena viene eseguita l'operazione di **mount** il core controlla se è presente un modulo che può gestirla cercando nella hash table. Nel caso preso in esempio la ricerca ha successo[4] in quanto trova una corrispondenza tra la stringa *"vufuseext2"*, nelle flag del comando, e *"vufuse"* presente come *obj* nell'elemento precedentemente inserito. Viene, pertanto, passata la gestione[5] al modulo **vufuse** che nell'eseguire la **vu_vufuse_mount** chiama nuovamente una funzione di inserimento nella hash table. La nuova chiamata è leggermente diversa, ma la semantica non cambia molto dalla precedente **vuht_add** e può essere così

[3] Un meccanismo del tutto analogo può essere ritrovato anche in **umview**. Per non appesantire il discorso si cercherà di riassumerlo in questo paragrafo, ma si consiglia nuovamente la lettura del manuale di Virtual Square [1] per avere una visione più chiara della tecnica utilizzata.

[4] Non ci si soffermerà a discutere come effettivamente viene fatta la ricerca perché porterebbe il discorso ad allungarsi notevolmente e si perderebbe lo scopo principale del paragrafo.

[5] In realtà è sempre il core che agisce, chiamando però le funzioni definite dal modulo e passate come campi della struttura *service* presente nella prima **vuht_add.**

riassunta: viene detto all'hypervisor di demandare l'esecuzione delle system call che coinvolgono il PATH[6] *mnt/* al modulo **vufuse**.

Una volta che il core ha scelto l'elemento della tabella che deve gestire la system call, lo salverà in una variabile a livello di thread così che anche le funzioni del modulo possano accedervi. Durante l'aggiunta dell'elemento il modulo potrebbe, infatti, aver salvato nella variabile delle informazioni di cui necessita in un secondo momento. **Umvu** lavora in *multithreading* e non è consigliabile, eccetto alcuni casi, utilizzare variabili globali poiché forte è il rischio di *race conditions*. Le variabili che dovrebbero essere globali, lo sono ma solo internamente al thread così che un processo virtualizzato non possa compromettere il funzionamento degli altri.

La lunga premessa fatta è necessaria per inquadrare una delle prime difficoltà incontrate, riguardante la corretta inizializzazione della variabile *static __thread struct vuht_entry_t**. Dato che **vufuse** crea un nuovo thread (nel quale esegue il processo FUSE per la gestione del file system montato), si incappava in un *segmentation fault* tentando di accedere alla variabile sopracitata perché non veniva inizializzata. Per correggere l'errore, si è aggiunto, alla struttura usata per la creazione del thread, un campo che contenesse il riferimento all'elemento di tabella hash con cui inizializzare la variabile presa in considerazione. Durante l'esecuzione del nuovo thread è stato possibile settare correttamente la variabile grazie alle funzioni fornite dalla libreria **vumodule.h**.

Vu_vufuse_cleanup

Il prossimo problema da analizzare permette la discussione di un aspetto interessante ai fini del progetto in generale. Non è inconsueto che l'utilizzo di un software faccia emergere la necessità di aggiungere funzionalità ad esso e nasca così la possibilità di migliorarlo. Con **vufuse** e **umvu** si è in un caso analogo perché il modulo sviluppato si basa fortemente sul core e sulle funzionalità da esso esposte. Durante l'implementazione si è visto che l'hypervisor non forniva un metodo sufficientemente consistente per controllare il disinserimento dei moduli nella fase di terminazione del core stesso. Accadeva che se si aggiungeva il modulo **vufuse**, si eseguiva una **mount** e si terminava l'ambiente virtuale prima di effettuare l'**umount**, il processo di FUSE rimanesse in sospeso non garantendo una corretta terminazione del tutto. E' stato quindi necessario rivedere la fase di terminazione del core e dei moduli.

Il nuovo meccanismo adottato non stravolge il modo in cui si rimuovono gli elementi di hash table relativi ai moduli, ma va a completarlo e perfezionarlo. Ogni elemento di hash table ha un campo che "tiene il conto" di quanti sono i thread che lo stanno utilizzando e quando il suo valore diventa 0 l'elemento può essere rimosso. E' però necessario un meccanismo che permetta di eseguire codice del modulo una volta che l'elemento sta per essere eliminato.

[6] Significativo nell'operazione di ricerca del hash table element è il campo *type*, che sta ad indicare come dover interpretare la variabile *obj*: CHECKMODULE indicherà la presenza del nome di un modulo, CHECKPATH di un pathname, CHECKDEVICE di un device etc...

I moduli dovranno ora definire una funzione di **cleanup** che verrà chiamata dal core prima di rimuovere l'hash table element del modulo stesso. La funzione si comporta come una procedura di *call back* permettendo di eseguire codice del modulo per consentirgli di effettuare le varie operazioni di terminazione. Così facendo se viene eseguita correttamente la **umount** non cambia quasi nulla dalla gestione precedente (ad eccezione che ora le operazioni di terminazione vanno implementate nella funzione di **cleanup** piuttosto che nella **umount** stessa), mentre nel caso in cui si termini l'ambiente virtuale il core penserà all'eliminazione degli elementi di hash table chiamando il codice della call back e permettendo anche ai moduli di terminare correttamente. Nel caso di **vufuse** la funzione **vu_vufuse_cleanup** si occupa proprio di sbloccare il processo FUSE e viene eseguita, come appena detto, sia conseguentemente ad una **umount** sia durante la chiusura del core evitando che il processo del sottomodulo rimanga in sospeso.

E' necessaria ancora un'ulteriore spiegazione riguardo la scelta presa. Si è preferito implementare la soluzione interamente nel core per ottenere una gestione uniforme per tutti i moduli, ma anche perché è l'hypervisor ad avere tutte le informazioni necessarie per capire quando rimuovere gli elementi di hash table e chiamare le funzioni di cleanup. Mettere la gestione della terminazione nei moduli stessi avrebbe portato a un livello di complessità più elevato dato che il modulo ha il solo compito di fornire al core le interfacce per le system call ridefinite e non è progettato per disporre di informazioni che non riguardino questo scopo.

Campo *private* nelle vu_vufuse_syscall

E' necessario, ancora una volta, rifarsi all'implementazione di **umvu** e di **umfuse**, prima di parlare della prossima difficoltà incontrata.

Nella versione precedente del modulo che stiamo esaminando, ogni volta che si apriva un file (**umfuse_open**), veniva creata una struttura dati che contenesse le informazioni necessarie al modulo per quel file. Questa variabile era poi inserita in una tabella hash del core e ogni volta doveva essere ripescata da lì per accedervi. In **vufuse** si è cercato di evitare questo tipo di gestione, sostituendola con una più semplice e sfruttando una nuova funzionalità introdotta dal core. Quando l'hypervisor chiama le system call ridefinite dal modulo, oltre a passare gli argomenti standard della syscall relativa, gli fornisce anche un campo opaco: *void* *private, utile proprio per passare informazioni tra una chiamata e l'altra. Nel core per ogni file aperto è creata una struttura contenete alcune informazioni utili all'hypervisor e inserita in un tabella hash. Il valore di *private* è conservato nella struttura relativa al file e può essere settato durante la **open** dal momento che questa funzione riceve un doppio puntatore (void **) a quel campo. Invece di utilizzare una ulteriore hash table risulta più semplice sfruttare quella che è già presente nel core così da evitare due strutture dati identiche. Così facendo, è sufficiente creare la struct sullo heap e far puntare la variabile *private* alla struttura stessa. Le operazioni appena descritte avvengono durante la **vu_vufuse_open**, mentre quando sono chiamate le altre system

call che necessitano di quell'elemento, non dovranno far altro che controllare la validità del campo *private* e accedervi.

2.1.3 Altri aspetti rilevanti per lo sviluppo

Per rendere la trattazione esaustiva, ci sono ancora alcuni aspetti da esaminare. Al momento non si è discusso di come **vufuse** si interfacci con le nuove versioni di FUSE ([API 2] nella [Figura 2.1]) e c'è da parlare nuovamente della **mount**. Si vedrà poi anche come è strutturata una tipica funzione di **vufuse** (prenderemo in esame la **vu_vufuse_lstat**).

FUSE_USE_VERSION

La **libfuse**, su github [4], si trova attualmente alla versione 3.2 e non da molto è avvenuto il passaggio dalla versione 2 alla 3. Come precedentemente detto, **umfuse** si interfaccia con FUSE e con i suoi sottomoduli. La versione della **libfuse** usata in questo caso è la 2.6, mentre **vufuse** al momento si conforma alla versione 2.9 della libreria di FUSE.

Si è scelto di interfacciarsi con la versione 2.9 in quanto più stabile e soprattutto perché gli sviluppatori di sottomoduli per FUSE fanno principalmente riferimento a quella. Per essere precisi, infatti, non solo **vufuse** risente della dipendenza da una qualche versione della **libfuse**, ma sono sopratutto i moduli di FUSE stesso che vi si devono conformare. La scelta della versione è stata successivamente avvalorata dalla semplicità con cui è avvenuto il passaggio dalla 2.6. E' stato necessario solamente cambiare la versione con cui veniva compilato il sottomodulo e in particolare lo *shared object*. La macro che gestisce la versione è proprio la **FUSE_USE_VERSION** che ora assume il valore 29. La **libfuse** è fatta in modo da garantire la retrocompatibilità con le versioni precedenti: è possibile, attraverso il valore dato alla macro, compilare l'eseguibile per il sottomodulo FUSE a una versione della libreria più vecchia di quella installata. Per **vufuse**, ovviamente, non è bastato settare FUSE_USE_VERSION al valore corretto in una qualche parte del codice, ma le operazioni di aggiornamento hanno coinvolto la revisione di alcuni file di compilazione.

Come risultato delle scelte fatte, si ha un modulo **vufuse** che funziona con una versione aggiornata della **libfuse**, senza essere però stati a faticare per rincorrere la libreria. Si è preferito arrivare ad una versione stabile di FUSE per poi poter discutere di come comportarsi in futuro. Ancora è del tutto aperto il problema di capire come fare per stare al passo con le versioni della **libfuse**. La questione non è banale ed è di un certo peso data la rilevanza della libreria.

Vusu

All'inizio si era detto che la **mount** necessita dei privilegi da super utente. Per capire questa sottosezione è necessario entrare più nello specifico del problema.

Quando si esegue l'operazione di **mount** vengono fatti vari controlli sullo *user id* (*group id* etc...) per verificare che siano quelli del super user. Il problema

è dato dal momento in cui sono eseguite le verifiche, le quali non avvengono tutte all'interno del codice della system call **mount**. Dato che **umvu** cattura "solo" le chiamate a sistema non sembrava possibile andare ad evitare i controlli appena descritti. Inizialmente era necessario comunque essere *root* per eseguire una **mount** in ambiente virtuale (ovvero con **umvu** attivo).

In questi casi risulta comodo andare a vedere come il problema è risolto nella versione precedente del core. In **umview** l'hypervisor si occupa di virtualizzare lo *user id*[7] e, quindi, quando vengono effettuati i controlli per la **mount** si "fa finta" di essere l'amministratore[8]. Nel nuovo core si è scelto di non rispecchiare il meccanismo di **umview** dato che è implementato in modo poco elegante, ma si è preferito demandare la virtualizzazione dello *user id* ad un modulo esterno.

Inizialmente per ragioni di praticità e per avere velocemente la possibilità di testare **vufuse** è stato implementato un piccolo modulo che virtualizzasse le chiamate agli id dell'utente (**getuid**, **getgid**,...) e ritornasse sempre 0, come accade quando è il super user ad eseguirle. La soluzione discussa, essendo momentanea, è stata rimpiazzata dall'uso del modulo **unrealuidgid**, predisposto proprio alla gestione degli *user id*, *group id*, insieme ad una utility specifica : **vusu**, la quale funziona come il comando **su**, ma in ambiente virtuale.

```
user:~$ umvu bash
user:~$ vu_insmod vufuse unrealuidgid
user:~$ vusu -p
root:~$
root:~$ mount -t vufuseext2 ext2.img mnt/
```

Vu_vufuse_lstat

Per finire presentiamo la struttura di una tipica funzione che va a ridefinire una system call. Il caso preso in esame è quello della syscall **stat**. Il codice riportato è volutamente non completo poiché l'interesse di questa sottosezione non è spiegare nel dettaglio come viene virtualizzata la **stat**, ma capire il ragionamento che sta dietro la re-implementazione delle system call e come ci si interfaccia con i sottomoduli FUSE.

vu_vufuse_lstat : vufuseop.c

```
int vu_vufuse_lstat(char *pathname, struct vu_stat *buf, int flags,
    int sfd, void *private) {

struct fuse_context *fc=vu_get_ht_private_data();
int rv;

[...]
```

[7] Quà come prima ci si riferisce anche al *group id, saved user id, file system user id.* Cercheremo di evitare di doverli riscrivere ogni volta per non appesantire la frase.

[8] Ovviamente non si è in presenza di un bug di sistema, perché nel caso in cui poi si eseguisse la vera system call **mount**, la sua esecuzione verrebbe impedita dalle verifiche interne.

```
memset(buf, 0, sizeof(struct vu_stat));
rv = fc->fuse->fops.getattr(pathname, buf);

printkdebug(F,"LSTAT path:%s status: %s retvalue:%d", pathname, rv ?
    "ERROR" : "SUCCESS", (rv < 0) ? -rv : 0);

if (rv<0) {
   errno = -rv;
   return -1;
} else {
   [...]
}
return rv;
}
```

La logica di funzionamento è molto lineare, il che rende la comprensione e manutenzione del codice relativamente semplice. Ogni volta che il core cattura una system call della famiglia delle **stat** (**lstat**, **fstat**,...) ne passa la gestione al modulo[9], il quale eseguirà la funzione appena riportata. La procedura in un certo senso fa da *wrapper* alla relativa funzione del sottomodulo di FUSE. In una prima fase si gestiscono i vari parametri di input e si prende la struttura dati salvata nell'elemento di hash table al momento della **mount**. In questa variabile è presente la **getattr** che è la funzione definita dal sottomodulo che opera come la **stat**. Una volta eseguita, si controllano i valori di ritorno, si procede, eventualmente, ad una stampa di debugging e si ripassa il controllo al codice del core.

2.1.4 Confronto con gli altri meccanismi di mounting

Verranno ora presi in considerazione tre esempi di **mount** eseguiti in tre modi diversi: con **vufuse**, con l'utility **mount** e con **fuse-ext2**. Il primo esempio costituirà il caso d'uso più frequente di **vufuse** e verrà brevemente commentato in quanto riunisce alcuni dei concetti prima trattati. Gli altri due esempi serviranno da paragone con il primo caso e permetteranno di mostrare alcuni dei pregi della **mount** con **vufuse**.

Sono necessarie alcune premesse per capire bene quello di cui si parlerà. Si supponga che l'utente *user1* abbia un'immagine di un file system di tipo **ext2** con il nome di *ext2.img* sulla quale ha pieni diritti sia in lettura che in scrittura. E' utile anche supporre che nell'immagine siano presenti dei file contenenti delle informazioni non condivisibili che l'utente ha bisogno di modificare. I file appena citati sono stati creati dall'utente senza però fare troppa attenzione alle regole di accesso dato che solo lui ha i permessi sul quell'immagine: in genere un file creato con una qualche utility come **touch**, **echo**, **vi** ha tutti i permessi in lettura e li ha in scrittura per l'utente e il gruppo (-rw-rw-r–) .

[9]Non ci dovrebbe essere bisogno di dire che il modulo **vufuse** è stato precedentemente caricato, la **mount** è stata eseguita, e il *path* della **stat** riguarda il mountpoint.

Il metodo più semplice per editare il file all'interno dell'immagine è utilizzare **vufuse**.

mount con vufuse

```
$ umvu bash
$ vu_insmod unrealuidgid vufuse
$ vusu -p
$
$ mount -o rw+ -t vufuseext2 ext2.img mountpoint/
$ ls mountpoint/
fileDaEditare
[editing del file]
```

E' possibile notare alcuni degli argomenti prima trattati come la necessità di usare il modulo **unrealuidgid** e l'utility **vusu**. Viene inoltre caricato lo *shared object* **vufuseext2** realativo al sottomodulo **fuse-ext2** con le opzioni 'rw+' per indicare di montare il file system in lettura e scrittura.

E' altrimenti possibile ma meno pratico utilizzare la **mount** classica o quella di **fuse-ext2**.

mount con privilegi da super user

```
$ su
$ mount -o rw -t ext2 ext2.img mountpoint/
[editing del file]
$ umount mountpoint/
```

mount con fuse-ext2

```
$ fuse-ext2 -o rw+ ext2.img mountpoint/
[editing del file]
# umount
$ fusermount -u mountpoint/
```

Si confrontino i tre esempi insieme per vederne pregi e difetti. Il modo peggiore per raggiungere lo scopo di modificare il file è dato sicuramente dalla **mount** con i privilegi d'amministratore, poiché non c'è motivo di aver bisogno di diritti di accesso superiori per compiere l'operazione dato che è fatta su file di proprietà dell'utente e l'azione non porta a danneggiare alcun elemento sensibile del sistema. Un ulteriore svantaggio è dato proprio dall'avere i massimi privilegi, il che comporta dover stare attenti alle azioni che si fanno e a ciò che si esegue: basta sbagliare la sintassi di un comando per andare a modificare/rimuovere un file di sistema piuttosto che il file dell'utente. L'ultimo problema del secondo esempio coinvolge anche il terzo, ovvero la visibilità del file system montato, la quale è estesa anche agli altri utenti. Se il path del mountpoint è accessibile in lettura anche da altri, sia nel caso della mount da super user che in quella con **fuse-ext2**, anche gli altri utilizzatori della macchina potranno accedere e vedere il contenuto di quei file con l'ulteriore svantaggio di rendere l'unmounting

impossibile finché non si cessa di utilizzare quel ramo di file system (EBUSY Device or resource busy). C'è ovviamente da dire che il terzo metodo risulta comunque molto comodo dato che non richiede alcun privilegio a conferma della potenza di FUSE e dei suoi sottomoduli.

Il primo esempio, invece, risulta essere quello con meno difetti. L'unico svantaggio, se così si può definire, è dato dal numero di comandi il quale è superiore rispetto agli altri casi, ma la cui sintassi e semantica rimane pressoché identica. Innumerevoli sono i vantaggi, a partire dalla intera gestione a livello utente fino ad arrivare alla visibilità che ha il file system montato. Non solo altri utenti saranno ignari di quel ramo del file system, ma esso non sarà nemmeno visibile da altri processi al di fuori dell'ambiente virtuale di **umvu**. L'utente può così modificare il *suo* file presente nella *sua* immagine stando sicuro di essere l'unico a poterlo *vedere*. E' sicuramente vero che con un attento controllo degli accessi anche con **fuse-ext2** si può arrivare a qualcosa di simile: basta settare le opportune flag per evitare l'accesso in lettura al path del mountpoint. Bisogna però sottolineare di nuovo che con **vufuse** la visibilità è a livello di processo, quindi nel caso l'utente abbia installato programmi malevoli nemmeno essi potranno accedervi e inoltre non è richiesto nessun controllo attento sull'accessibilità del mountpoint da parte di altri. Infine si noti come per il secondo e terzo caso sia necessario effettuare l'**umount** del fie system, mentre con **vufuse** non è richiesto dato che chiudendo l'ambiente virtuale è eseguito automaticamente.

2.2 Vufusereal

Assieme allo sviluppo di **vufuse** è stato implementato anche un sottomodulo di FUSE specifico per esso. Il compito di **vufusereal** è quello di bypassare lo strato dato dai moduli che usano la **libfuse** mappando ogni system call "virtuale" nella sua implementazione "reale". Si era detto nella sezione 1.2.1 che il sottomodulo FUSE deve esporre delle implementazioni per le system call che saranno chiamate sul file system montato al posto delle funzioni reali. In questo caso però la ridefinizione della system call non fa nient'altro che chiamare quella originale. Ecco un esempio.

op_getattr : ops_vufusereal.c

```
int op_getattr (const char *path, struct stat *stbuf){
    GETPATH(path)

    int rv = lstat(path,stbuf);

    RETURN(rv)
}
```

Tralasciando le operazioni iniziali e finali date dalle due macro, il cuore della funzione è costituito dalla semplice chiamata alla system call **lstat**.

Potrebbe sembrare che **vufusereal** non faccia nulla e sia inutile : in effetti, attraverso di esso è unicamente possibile montare sotto-rami del proprio file

system, un'operazione apparentemente poco utile . In realtà, il suo scopo è di separare la complessità dell'implementazione di **vufuse** da quella dei vari sottomoduli per FUSE. Si supponga che durante l'uso del modulo **vufuse** con **fuse-ext2** si presenti un certo errore. Nella fase di debugging sarà utile andare a capire se quel problema è dovuto all'implementazione di **vufuse** o a quella del sottomodulo. E' allora possibile utilizzare **vufusereal** al posto di **fuse-ext2** per replicare il caso di test e vedere se si ripresenta l'errore. Le possibilità sono due: o l'errore rimane oppure non ricompare. Nel primo caso si avrà la certezza che il bug si trova nell'implementazione di **vufuse**; nel secondo risulterà facile capire che quell'errore non è dovuto al nostro modulo.

Difficoltà incontrate

La struttura del codice è semplice e lineare il che ha reso molto facile la fase di debugging e risoluzione dei vari problemi. E' utile spendere comunque due parole per parlare di un aspetto interessante dell'implementazione di **vufusereal**: il modo in cui ci si interfaccia con l'API fornita da FUSE e la gestione dei valori di ritorno.

Per la fase di inizializzazione del modulo (la funzione **main**) ci si è fortemente ispirati a **fuse-ext2** dato che presenta delle buone funzioni di gestione delle *option*. Nella stessa fase sono presenti anche alcune procedure della **libfuse** per effettuare la **mount** e comunicare con il modulo FUSE del kernel.

L'unica difficoltà affrontata ha riguardato i valori da ritornare. La **libfuse** è molto stringente sotto questo punto di vista e pretende che tutte le syscall ridefinite in caso di fallimento ritornino - *errno*, mentre in caso di successo, alcune devono restituire lo stesso valore della system call reale (i.e. **op_read**, **op_write** devono tornare il numero di byte letti o scritti), altre, come la **op_open**, devono restituire 0. E' stato pertanto necessario controllare ognuna delle definizioni e verificare che il valore tornato fosse corretto e rispettasse le direttive presenti nel file **fuse.h** in cui si trova l'API di alto livello (verso i sottomoduli) della libreria di FUSE.

Capitolo 3

Conclusione e futuri sviluppi

Cerchiamo di ricapitolare e tirare le somme di quanto detto nei capitoli precedenti. Il file system rappresenta una delle astrazioni più importanti in un kernel Unix-like, perciò il codice atto alla sua gestione viene eseguito con i privilegi da super utente e in kernel mode. Nasce, comunque, la necessità di dare all'utente la possibilità di montare i propri file system in user space. La risposta a questo bisogno è FUSE, il quale, attraverso un modulo kernel specifico e una libreria (**libfuse**), da all'utente o allo sviluppatore più libertà nell'uso dei file system. Altro strumento di notevole utilità è ViewOS, una componente del progetto Virtual Square, il quale permette di modificare la normale visione che i processi hanno del loro ambiente circostante. In particolare l'utilizzo di due tool, **umview** e **umfuse**, consente di lavorare virtualmente con file system di tipo FUSE. Con la nuova versione di **umview**, **umvu**, si sente la mancanza di un aggiornamento di **umfuse**, modulo che gestisce il collegamento con la **libfuse**. Da tutte queste premesse nasce **vufuse**.

Il nuovo modulo si basa sicuramente sul suo predecessore, ma lo supera a livello di scelte progettuali e di implementazione. La scrittura di **vuos** non è da riassumere come un banale aggiornamento dalla vecchia alla nuova versione, ma ha permesso di rimettere in discussione molte delle scelte effettuate in passato per capire se certi meccanismi andassero mantenuti oppure migliorati. Uno degli aspetti più interessanti della progettazione e della risoluzione di difficoltà incontrate, probabilmente, è lo stretto legame che si è instaurato tra modulo e core. Mano a mano che si procedeva nello sviluppo e nella progettazione del modulo si è andati a migliorare anche il core così da fornire ai prossimi sviluppatori un ambiente già consolidato e sperimentato. Si ricordi, per esempio, la gestione della terminazione dell'hypervisor e dei moduli, oppure il meccanismo utilizzato per evitare che la **mount** necessiti dei privilegi anche in ambiente virtuale (**vusu**). Entrambi sono i due esempi più importanti di feature che prima non erano presenti nel core ma che ora forniscono notevole supporto ai moduli.

Futuri sviluppi

Il progetto VuOS è molto vasto e, sebbene **vufuse** sia uno dei moduli più rilevanti, sicuramente non è l'unico. Mancano ancora da sviluppare molti dei moduli che erano presenti nella vecchia versione come, per esempio, **umdev** e **umbinfmt**[1]. E' presente anche la possibilità di sviluppare un modulo che controlli gli accessi, come accennato nella sezione 2.1.1 parlando della "human mode". Il modulo si distinguerebbe dagli altri poiché deve essere sempre il primo a venir chiamato dopo la cattura di system call come la **access**, per esempio. Il core, al momento, tiene anche conto dell'ordine di inserimento dei moduli per scegliere a chi dare la gestione della chiamata. Il modulo di cui si sta parlando rappresenterebbe un'eccezione e si dovrebbe fare in modo che sia sempre l'ultimo ad essere stato inserito. Siccome la progettazione viene prima di tutto e non si vogliono introdurre dei casi particolari qualora non siano strettamente necessari, si attende di trovare la migliore soluzione prima di passare ad una qualche implementazione.

Parlando, invece, più nello specifico di **vufuse** si può affermare che si trova ad una versione stabile, ma "nulla è perfetto o, comunque, tutto è perfettibile". Un problema del tutto aperto fa riferimento a quello che viene definito come *nested mount*, ovvero la possibilità di montare un file system in una directory virtuale (in parole povere, eseguire per due volte la mount di **vufuse**, ma la seconda volta in una delle directory del sotto-ramo aggiunto precedentemente). Il meccanismo funziona solo quando l'immagine da montare la seconda volta non si trova in un path virtuale. Il problema è dovuto alla terminazione e uscita dei thread generati dal core. Probabilmente la soluzione non è banale ed è necessaria una profonda riflessione riguardo le chiamate per la gestione dei thread come la **__exit_thread** e la **_clone**.

Infine va menzionato il problema dell'aggiornamento alle varie versioni della **libfuse**(di cui si era già parlato nella sottosezione 2.1.3). Anche in questo caso la questione non è banale dato che la libreria di FUSE è in costante aggiornamento e sviluppo, quindi starle dietro non è semplice. Le soluzioni attualmente possibili sembrano essere due: creare per ognuno dei sottomoduli di FUSE un ulteriore strato software che consenta di interfacciarsi con **vufuse**, mantenendo quest'ultimo inalterato; oppure cercare di aggiornare con più costanza il nuovo modulo alle più recenti versioni della **libfuse**. La questione è ancora da ragionare in modo approfondito, ma, momentaneamente, la seconda soluzione è preferibile dato che introduce un grado di complessità inferiore: è sicuramente più semplice aggiornare un unico modulo piuttosto che dover gestire quel piccolo strato software per *ogni singolo* sottomodulo di FUSE.

[1] Si consiglia per l'ultima volta di riferirsi al manuale citato nella bibliografia[1]

Bibliografia e Sitografia

[1] Renzo Davoli e Michael Goldweber. *Virtual Square. Users,Programmers & Developers Guide.* 2008. URL: `http://www.cs.unibo.it/~renzo/virtualsquare/`.

[2] Gardenghi L., Goldweber M. e Davoli R. "View-OS: A New Unifying Approach Against the Global View Assumption". In: *Lecture Notes in Computer Science.* Vol. 5101: *Computational Science - ICCS 2008.* A cura di Bubak M. et al. Berlin, Heidelberg: Springer, 2008.

[3] Jeffrey B. Layton. *User Space File Systems.* 22 Giu. 2010. URL: `http://www.linux-mag.com/id/7814/`.

[4] *Libfuse master branch on Github.* URL: `https://github.com/libfuse/libfuse`.

[5] Tim Love. *Ansi C for programmers on Unix systems.* URL: `http://www.cs.unibo.it/~renzo/doc/C/AnsiC.pdf`.

[6] Davoli R., Goldweber M. e Gardenghi L. "UMView: View-OS implemented as a system call virtual machine". In: *7th Usenix Symposium on Operating Systems Design and Implementation.* Seattle (WA), November 2006.

[7] Tanenbaum e Woodhill. *Operating System Design and Implementation.* 2nd edition. Prentice Hall, 1997.

[8] *Virtual Sqaure Wiki.* URL: `http://wiki.v2.cs.unibo.it/wiki/index.php?title=Main_Page`.

[9] *Vuos master branch on Github.* URL: `https://github.com/virtualsquare/vuos`.