TP3: Filesystem FUSE

Introducción

AVISO: antes de comenzar, verificar que se tiene instalado **el software necesario**.

En este trabajo implementaremos nuestro propio sistema de archivos (o *filesystem*) para Linux. El sistema de archivos utilizará el mecanismo de FUSE (*Filesystem in USErspace*) provisto por el kernel, que nos permitirá definir en *modo usuario* la implementación de un *filesystem*. Gracias a ello, el mismo tendrá la interfaz VFS y podrá ser accedido con las syscalls y programas habituales (`read` , `open` , `ls` , etc).

La implementación del *filesystem* será enteramente en memoria: tanto archivos como directorios serán representados mediante estructuras que vivirán en memoria RAM. Por esta razón, buscamos un sistema de archivos que apunte a la velocidad de acceso, y no al volumen de datos o a la persistencia (algo similar a `tmpfs`). Aún así, los datos de nuestro *filesystem* **estarán** representados *en disco* por un archivo.

Software necesario

FUSE está compuesto de varios componentes, los principales son:

- un módulo del kernel (que se encarga de hacer la delegación)
- una librería de usuario que se utiliza como framework

Para realizar este trabajo se necesitará un sistema operativo que cuente con el kernel Linux, y que disponga de las librerías de FUSE versión 2.

Pueden instalarse todas las dependencias con el siguiente comando:

```
sudo apt update && sudo apt install pkg-config libfuse2 libfuse-dev
```

El código del esqueleto viene con un *filesystem* FUSE trivial, para poder probar las dependencias. Si las mismas están correctamente instaladas, deberían poder *compilar y ejecutar* el código del esqueleto de la siguiente forma:

- Compilación

```
$ make
gcc -ggdb3 -O2 -Wall -std=c11 -Wno-unused-function -Wvla fisopfs.c  -D_FILE_OFFSET_BITS=64 -
I/usr/include/fuse -lfuse -pthread -o fisopfs
```

- Montando un directorio

```
// Creamos un directorio vacío
$ mkdir prueba
// Ejecutamos nuestro binario y le decimos dónde queremos que monte nuestro filesystem
$ ./fisopfs prueba/
// Verificamos que se haya montado correctamente
$ mount | grep fisopfs
/vagrant/labs/fisopfs/fisopfs on /vagrant/labs/fisopfs/prueba type fuse.fisopfs
(rw,nosuid,nodev,relatime,user_id=1000,group_id=1000)
```

- Pruebas sobre el directorio

```
$ ls -al prueba/
total 0
drwxr-xr-x 2    1717 root       0 Dec 31  1969 .
drwxr-xr-x 1 vagrant vagrant  352 Nov 13 12:41 ..
-rw-r--r-- 1    1818 root    2048 Dec 31  1969 fisop
$ cat prueba/fisop
hola fisopfs!
```

- Limpieza

```
$ sudo umount prueba
```

Filesystem tipo FUSE

La compilación y ejecución de un cliente FUSE es algo distinta. El esqueleto ya está preparado (en el `Makefile`) para compilar incluyendo los flags de compilación necesarios, pero **se recomienda leer este artículo** antes de arrancar, y antes de introducir modificaciones en el `Makefile` . En particular, se utiliza la utilidad `pkg-config` para obtener los flags de compilación adecuados.

En el artículo también podrán encontrar una explicación sobre cómo utilizar la librería de FUSE e implementar sus propias funciones. En el caso del esqueleto, únicamente están implementadas 3 primitivas del sistema de archivos: `getattr`, `readdir` y `read`.

```
static struct fuse_operations operations = {
        .getattr = fisopfs_getattr,
        .readdir = fisopfs_readdir,
        .read = fisopfs_read,
};
```

La documentación oficial de FUSE es muy útil para revisar las firmas de las funciones y los campos de cada struct de la librería. Sin embargo, hay que tener en cuenta que esa documentación es **para la versión 3**, y por lo tanto podría tener algunas diferencias en la API/firma de algunas funciones que utilizaremos en el TP.

La documentación para la versión 2 (la última, 2.9.9) más precisa es el código del *header* (`fuse.h`) en sí, el cual pueden encontrarlo en el repositorio. En el mismo se encuentran todos los *structs*, funciones auxiliares y firmas; junto con comentarios útiles.

> **IMPORTANTE**: Si tienen errores al momento de compilar, o ven alguna discrepancia con la documentación, es posible que estén usando FUSE versión 3. Pueden comprobarlo con `apt list "libfuse*"`

Implementación

Implementaremos `fisopfs`, un *filesystem* de tipo FUSE definido por el usuario. El mismo deberá implementar un subconjunto de las operaciones que soporta FUSE. Las operaciones serán las necesarias para soportar la lista de operaciones que figura a continuación.

Operaciones requeridas

- Es *requisito* que el sistema de archivos soporte las siguientes funcionalidades:
  - Creación de archivos (`touch`, redirección de escritura)
  - Creación de directorios (con `mkdir`)
  - Lectura de directorios, incluyendo los pseudo-directorios `.` y `..` (con `ls`)
  - Lectura de archivos (con `cat`, `more`, `less`, etc)
  - Escritura de archivos (sobre-escritura y append con redirecciones)
  - Acceder a las estadísticas de los archivos (con `stat`)
    - Incluir y mantener fecha de último acceso y última modificación
    - Asumir que todos los archivos son creados por el usuario y grupo actual (ver `getuid(2)` y `getgid(2)`)
  - Borrado de un archivo (con `rm` o `unlink`)
  - Borrado de un directorio (con `rmdir`)
- La creación de directorios debe soportar al menos un nivel de recursión, es decir, directorio raíz y sub-directorio.

Para cada una de las funcionalidades pedidas, se espera que la misma se pueda corroborar montando el sistema de archivos e interactuando con el mismo a través de una terminal `bash`, de la misma forma que se haría con cualquier otro *filesystem*.

> **AYUDA**: Una buena forma de saber qué *operaciones* hacen falta es implementar las mismas vacías con `printf` de debug; y montar el *filesystem* en primer plano: `./fisopfs -f ...`. Esto permitirá loggear todas las operaciones/syscalls que generan los procesos al interactuar con el sistema de archivos.

En caso de error o funcionalidad no implementada, el sistema de archivos debe escribirlo por pantalla (basta con un `printf` a `stderr`, o con el prefijo `[debug]`) y devolver el error apropiado. Pueden tomar los errores definidos en `errno.h` (ver `errno(3)`) como inspiración, viendo qué errores arrojan otros sistemas de archivos. Algunas opciones útiles son: `ENOENT`, `ENOTDIR`, `EIO`, `EINVAL`, `EBIG` y `ENOMEM`, entre otros.

Representación del sistema de archivos

El sistema de archivo implementado debe existir en memoria RAM durante su operación. La estructura en memoria que se utilice para lograr tal funcionalidad es enteramente a diseño del grupo. Deberá explicarse claramente, con ayuda de diagramas, en el informe del trabajo (i.e. archivo `fisopfs.md`); las decisiones tomadas y el razonamiento detrás de las mismas.

La primera parte, consistirá en el diseño de las estructuras que almacenarán toda la información, y cómo se accederán en cada una de las operaciones. Para luego implementarlas en la segunda parte del trabajo práctico.

Documentación de diseño

- Se deben explicar los distintos aspectos de diseño:
  - Las estructuras en memoria que almacenarán los archivos, directorios y sus metadatos
  - Cómo el sistema de archivos encuentra un archivo específico dado un *path*
  - Todo tipo de estructuras auxiliares utilizadas
  - El formato de serialización del sistema de archivos en disco (ver siguiente sección)
  - Cualquier otra decisión/información que crean relevante

AYUDA: Se recomienda utilizar una estructura *flat* para los directorios, limitando el *filesystem* a un único nivel de recursión en los directorios. O bien implementar un esquema con *inodes* similar a los sistemas de archivo *Unix-like*, y permitir múltiples niveles de directorios. De todas formas, es recomendable definir un límite en la longitud del *path* o en la cantidad de directorios anidados soportados.

AYUDA 2: Definir un tamaño máximo para el sistema de archivos, y arrojar `ENOSPC` (*No space left on device*), o similar, si nos excedemos del mismo. Se recomienda utilizar arreglos de tamaño estático para tal fin, para simplificar el manejo de memoria.

Persistencia en disco

Si bien el sistema de archivos puede vivir enteramente en RAM durante su operación, también será necesario persistirlo a disco al desmontarlo y recuperarlo de disco al montarlo.

El *filesystem entero* se representará como un único archivo en disco, con la extensión `.fisopfs`; y en el mismo se serializará toda la estructura del *filesystem*. Al montar el *filesystem*, se espera que toda esa información se lea de disco en memoria, y la operación continúe exclusivamente en memoria. Cuando el *filesystem* se desmonte (o si ocurre una llamada explícita a `fflush`), la información debe persistirse nuevamente en disco. De esta forma, a través de múltiples ejecuciones, los datos persistirán.

Persistencia en disco

- Es *requisito* que el sistema de archivos se persista en disco
  - En un único archivo, de extensión `.fisopfs`
  - Al lanzar el *filesystem*, se debe especificar un nombre de archivo, si no se hace, se elige uno por defecto
  - Del archivo especificado se lee todo el *filesystem*, y se inicializan las estructuras acordemente (esto ocurre en la función `init`)
  - Si ocurre un `flush` o cuando el sistema de archivos se desmonta (esto ocurre en la función `destroy`), la data debe persistirse en el archivo nuevamente

Modularización

Es *recomendable* tener la lógica del *filesystem* en otro archivo (por ejemplo `fs.c`) y que `fisopfs.c` llame a estas primitivas. Dichas funciones **no** deberían recibir ningún tipo de dato que sea de *FUSE* ya que ésto romperían la *abstracción*.

Entonces, si uno quisiera tener una primitiva para leer las *entradas* de un directorio, se podría hacer:

```
char entry_name[MAX_ENTRY_NAME];

res = fs_read_dir(&fs, path, entry_name);
```

Y luego el puntero a función que recibe FUSE (`fuse_fill_dir_t filler`) se llamaría por cada una de éstas entradas. Algo como:

```
while (res > 0) {
        filler(buffer, entry_name, NULL, 0);
        res = fs_read_dir(&fs, path, entry_name);
}
```

Otra posible implementación, podría ser con *memoria dinámica* devolviendo una lista de las entradas. Por ejemplo:

```
char **entries = fs_list_dir_entries(&fs, path);

int entry_idx = 0;

while (entries[entry_idx] != NULL) {
        filler(buffer, entries[entry_idx], NULL, 0);
        entry_idx++;
}
```

Para poder compilar fácilmente los nuevos módulos, se pueden agregar al `Makefile` de la siguiente forma:

```
# ...

build: $(FS_NAME)

# por cada módulo se agrega un nuevo item
# ejemplo:
#   si además tenemos un archivo llamado file.c
#   la siguiente linea quedaría
# $(FS_NAME): fs.o file.o
$(FS_NAME): fs.o

format: .clang-files .clang-format

# ...
```

Pruebas y salidas de ejemplo

Como parte de la implementación de `fisopfs` también será necesario **incorporar pruebas** de caja negra sobre lo implementado. Las mismas deben consistir de una serie de secuencias de comandos pensadas para generar un escenario de prueba, junto con la salida esperada del mismo. Un ejemplo, es lo presentado en la sección "Software necesario".

Cada funcionalidad implementada debe incluir una prueba asociada. Las salidas de las pruebas deben incluirse como una sección en el informe.

Es altamente recomendable pensar y escribir las pruebas incluso antes de arrancar con la implementación, para tener una guía del comportamiento del sistema.

Esqueleto y compilación

AVISO: El esqueleto del TP3 se encuentra disponible en **fisop/fisopfs**.

IMPORTANTE: leer el archivo `README.md` que se encuentra en la raíz del proyecto. Contiene información sobre cómo realizar la compilación de los archivos, y cómo ejecutar el formateo de código.

Para compilar nuestro *filesystem*, se puede:

```
make
```

Desafíos

Las tareas listadas aquí no son obligatorias, pero suman para el régimen de final alternativo.

Implementación de más operaciones para `fisopfs`

Más allá de los requisitos obligatorios, los grupos podrán optar por implementar *al menos* dos de las siguientes funcionalidades adicionales.

- Soporte para enlaces simbólicos
  - Debe implementarse la operación `symlink`
  - Deben incluirse pruebas utilizando `ln -s`
- Soporte para hard links
  - Debe implementarse la operación `link`
  - Deben incluirse pruebas utilizado `ln`
  - Notar que ahora el borrado *real* de un archivo solo debe ocurrir si no quedan más *hard links* asociados al mismo.
- Soporte para múltiples directorios anidados
  - Más de dos niveles de directorios
  - Se debe implementar una cota máxima a los niveles de directorios y a la longitud del *path*

- Agregar validaciones de permisos
  - Comprobar si el usuario que accede tiene permisos para leer el archivo/directorio
  - Implementar las operaciones `chown` y `chmod` para modificar permisos y ownership de un archivo/directorio

En cualquier caso, las operaciones elegidas deben implementarse incluyendo pruebas de la misma forma que para el resto de las funcionalidades.

Bibliografía útil

A continuación se presentan algunos enlaces y bibliografía útiles como referencia.

- OSTEP, capítulo 39: *Interlude: Files and Directories* (PDF)
- OSTEP, capítulo 40: *File System Implementation* (PDF)
- The Linux Programming Interface, capítulo 14: *File systems*