Laboratorio 4: BigBrother FS

FaMAFyC - Sistemas operativos 2023

Fecha de entrega 16/11/2023 15:59hs

Sistemas de archivos

El *sistema de archivos* es una de las tantas abstracciones (exitosas) que presenta el sistema operativo para manejar recursos. Nacidas originalmente como una interfaz para los medios de almacenamiento permanente, evolucionaron rápidamente hasta convertirse en una parte fundamental de los sistemas operativos. En file systems modernos es posible *montar* varios dispositivos con sistemas de archivos independientes, brindando una interfaz abstracta, uniforme y cómoda de usar, todo dentro de la misma jerarquía de directorios provista por el file system nativo.

En este laboratorio vamos a trabajar con un filesystem de tipo FAT pero que solo tiene implementada una parte de su interfaz. Por ejemplo, en principio, no podemos eliminar directorios porque la operación rmdir no está desarrollada. En el proyecto vamos a completar parte de la interfaz para ampliar la funcionalidad del filesystem lo que nos va a ayudar a entender la flexibilidad que nos provee el filesystem de unix. Por otro lado, vamos a añadir funcionalidades que nos permitan vigilar la actividad dentro del sistema de archivos de manera transparente al usuario, con el objetivo de poner en perspectiva la importancia de los sistemas operativos de código abierto.

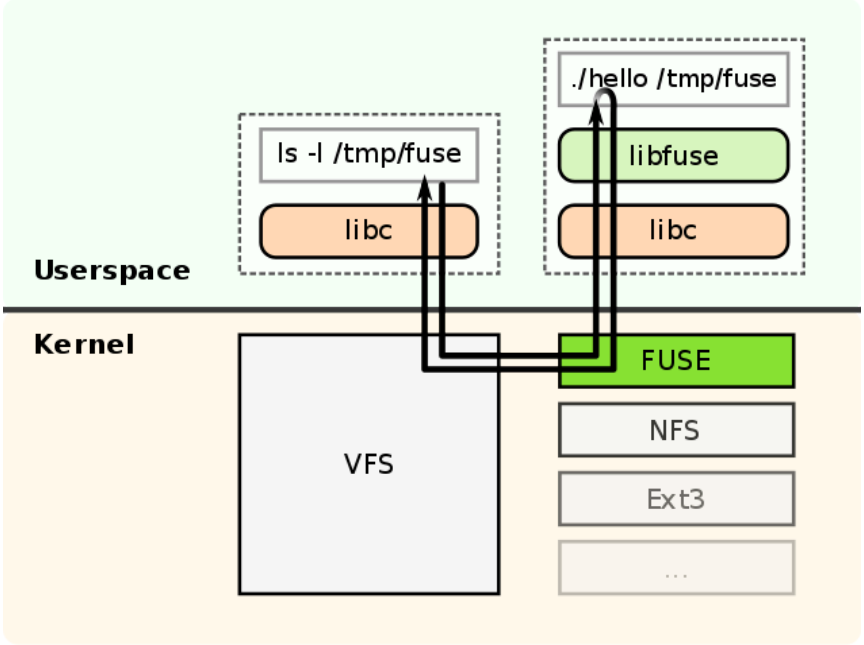
FAT-FUSE

Debido al diseño del VFS de UNIX, resulta relativamente sencillo agregar un nuevo FS al kernel: basta con implementar el módulo correspondiente siguiendo los lineamientos y formato de cualquiera de ellos y cargarlo. Sin embargo, el código que desarrollamos corre dentro del espacio kernel y cualquier falla involucra posiblemente un reinicio del mismo. Esto sin contar con la escasez de herramientas de debugging y el hecho de tener que involucrarse con las estructuras internas del kernel, por lo que todo el proceso de desarrollo se vuelve dificultoso.

FUSE o Filesystem in USErspace, es un módulo más del VFS que permite implementar filesystems a nivel de userspace. Las ventajas son varias:

- Interfaz sencilla: aísla de ciertas complejidades inherentes al desarrollo dentro del kernel.
- Estabilidad: como el FS está en userspace, cualquier error se subsana desmontando el sistema de archivos.
- Posibilidad de debuggear: tenemos todas las herramientas userland para atacar los bugs.
- Variedad de language bindings: se puede hacer un FUSE con Python, Perl, C, Haskell, sh, etc.

FUSE exporta la interfaz de un filesystem a userspace mediante una serie de funciones que definimos en la estructura `struct fuse operations`. Estas funciones son las que definen de qué forma se organizará el sistema de archivos, cómo serán las funciones de lectura y escritura, etc. FUSE hace de interfaz entre nuestra implementación, en modo usuario, y las llamadas a sistema que realiza el usuario. De esta manera, aseguramos que el código que escribamos sólo accede al volumen sobre el que tenemos permisos, ya que el único código que se ejecuta en espacio kernel es el de FUSE. Por ejemplo, si un programa de usuario quiere abrir un archivo, ejecuta la llamada a sistema `open`. El sistema operativo, al ver que el volumen está montado al VFS con FUSE, le transfiere la llamada a sistema y FUSE busca en la estructura `fuse_operations` la función correspondiente que nosotros hemos implementado para efectivamente realizar la apertura del archivo.



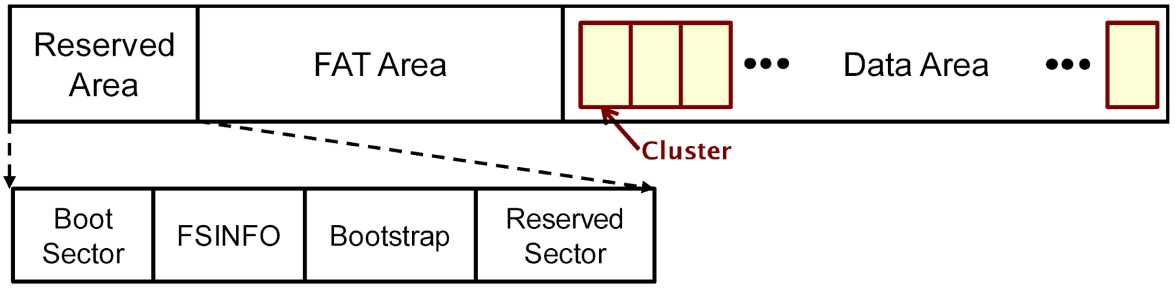
Con esta arquitectura, FUSE es lo suficientemente flexible como para permitir cualquier tipo de sistema de archivos, de acuerdo a cómo están descritas en las operaciones de `fuse_operations`. En este laboratorio, hemos implementado dichas operaciones para que sean compatibles con el sistema de archivos FAT, pero se podría

utilizar cualquier otro protocolo, como se muestra en este ejemplo. Incluso puede haber operaciones que no estén definidas, como por ejemplo en un sistema de archivos de solo lectura, sin operaciones de escritura.

Sistema de archivos FAT

El sistema de archivos FAT está resumido en la correspondiente página de Wikipedia que les recomendamos que lean. Aquí sólo les daremos un paseo por los conceptos más importantes. Los sistemas FAT dividen al dispositivo en las siguientes áreas principales:

- Sectores reservados, donde se encuentra la información que define el tipo de sistema FAT y sus propiedades, como el número de tablas FAT, el tamaño de cada cluster y el número total de clusters. Para este laboratorio podemos abstraernos de todo esto.
- La tabla FAT (File Allocation Table), que contiene información sobre qué clusters de datos están siendo utilizados y cuáles pertenecen a cada archivo o directorio. Usualmente hay dos copias de la tabla FAT para redundancia de datos.
- El directorio root, que está separado del resto de los directorios sólo en FAT12 y FAT16. En el protocolo FAT32, el directorio raíz está almacenado con el resto de los datos.
- El área de datos, que se divide en los clusters descritos en la FAT. Aquí se encuentran efectivamente los contenidos de los archivos y las tablas de los directorios.



Clusters

El área de datos se divide en clusters de tamaño idéntico, que son pequeños bloques de disco continuos. El tamaño varía dependiendo del tipo de FAT, nosotros usaremos 512 bytes por cluster.

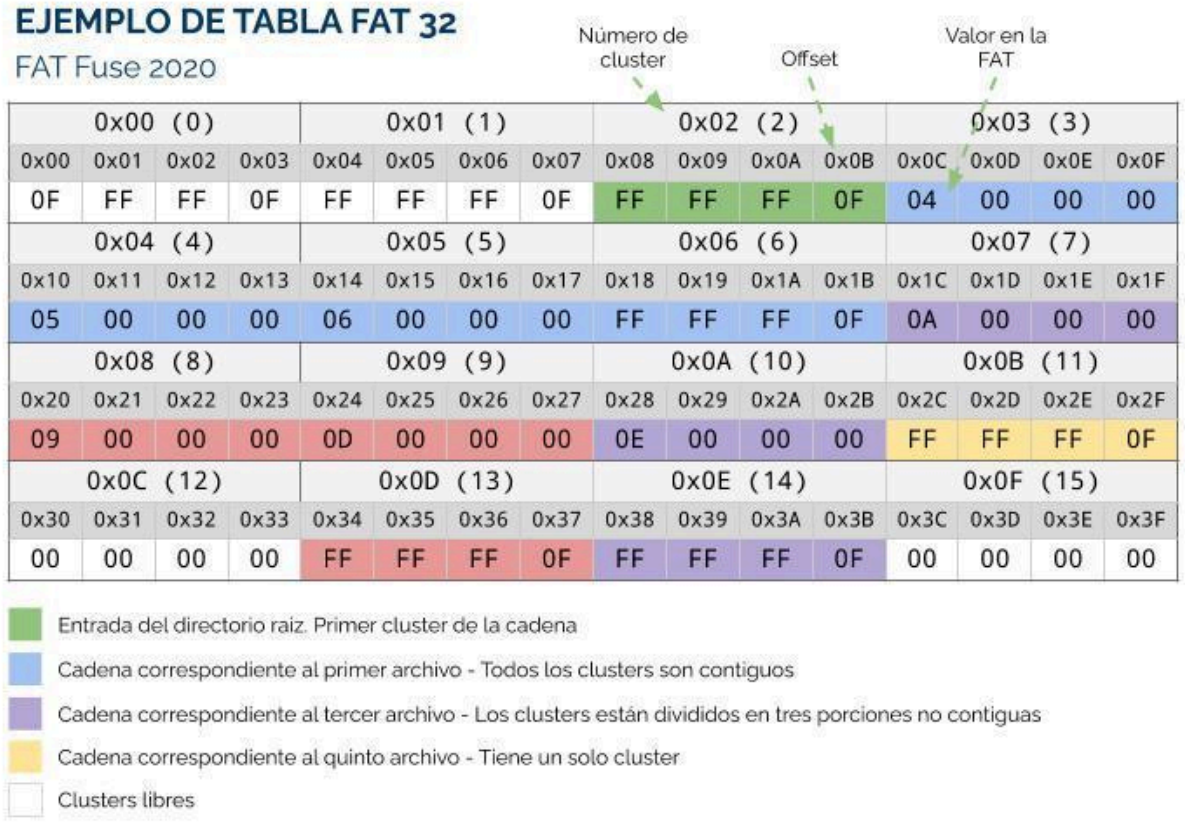
Cada archivo puede ocupar uno o más de estos clusters, dependiendo de su tamaño. Por lo tanto, cada archivo es representado como una lista ordenada de los

clusters que contienen su información (*linked list*), almacenada en la FAT. Sin embargo, estos clusters no necesariamente se encuentran en espacios contiguos en el disco, y cada archivo reserva nuevos clusters a medida que va creciendo. (¿Se acuerdan de desfragmentar Windows? Es por esto, ya que un mismo archivo puede estar desperdigado en distintos sectores del disco, ralentizando el proceso de lectura. La desfragmentación reubica los clusters de un archivo en espacios continuos).

Si el archivo es un directorio, entonces su cluster contiene una lista de entradas de directorio (`dentrys`) donde se especifica el nombre de los archivos (incluyendo subdirectorios) que están contenidos en él, con sus correspondientes metadatos. Cada entrada de directorio es de tamaño fijo y están almacenadas en espacios contiguos de memoria. Por ende, podemos saber que ya hemos leído todas las entradas del directorio cuando encontramos el primer archivo con el nombre vacío.

Tabla FAT

La FAT es una tabla donde cada cluster está representado por 12, 16 o 32 bits de información contiguos (FAT12, FAT16 y FAT32 respectivamente). Esto también define el número máximo de clusters del sistema.

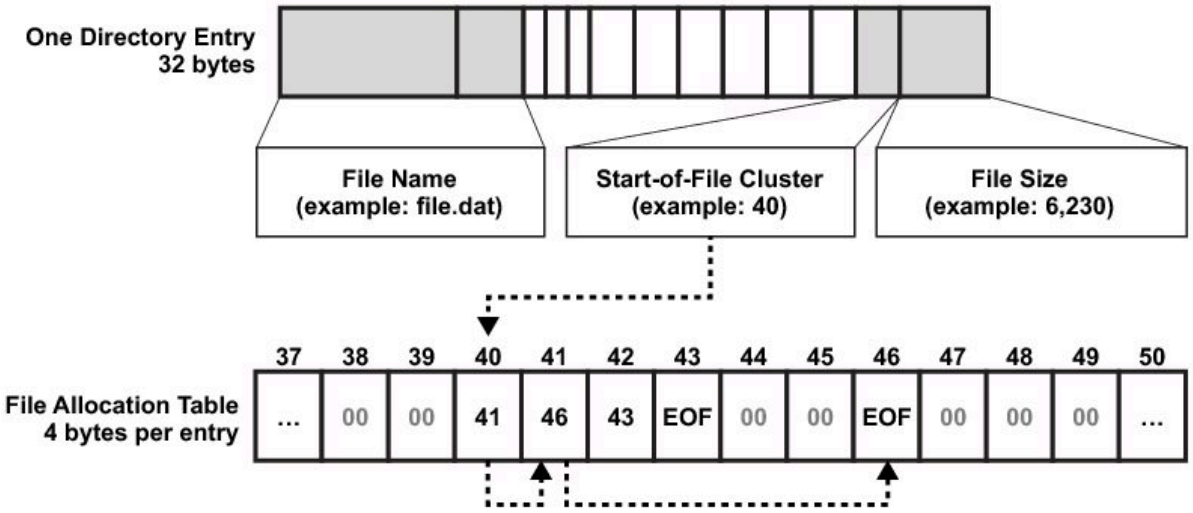


0x00 (0)				0x01 (1)				0x02 (2)				0x03 (3)			
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
0F	FF	FF	0F	FF	FF	FF	0F	FF	FF	FF	0F	04	00	00	00
0x04 (4)				0x05 (5)				0x06 (6)				0x07 (7)			
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
05	00	00	00	06	00	00	00	FF	FF	FF	0F	0A	00	00	00
0x08 (8)				0x09 (9)				0x0A (10)				0x0B (11)			
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27	0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
09	00	00	00	0D	00	00	00	0E	00	00	00	FF	FF	FF	0F
0x0C (12)				0x0D (13)				0x0E (14)				0x0F (15)			
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F
00	00	00	00	FF	FF	FF	0F	FF	FF	FF	0F	00	00	00	00

Los valores posibles para una entrada de la FAT son:

- `0x?0000`: indica que el cluster está libre
- `0x?0001`: indica que el cluster está reservado para el sistema
- de `0x?0002` a `0x?FFEF`: indica el siguiente cluster en la cadena para un archivo dado.
- `0x?FFF0` a `0x?FFF6` están reservados.
- `0x?FFF7`: indica sector dañado en algunos sistemas.
- `0x?FFF8` - `0x?FFFF`: indica el final de la cadena de clusters.

Dentro de la información que encontramos en una entrada de directorio que corresponde a un determinado archivo, encontramos, además de su nombre y su tamaño, el número de su clúster de datos inicial. Esto nos permite, junto con la FAT table, conseguir todos los clusters que ocupa un archivo



Entrega

Requisitos necesarios para aprobar:

- Utilizar el repositorio de Bitbucket, con commits pequeños y nombres significativos. Todos los integrantes del grupo deben colaborar.
- Utilizar `clang-format` para mantener un estilo de código consistente.
- **Pasar los tests provistos por la cátedra.**
- Deben entregar una versión básica con la implementación pedida en la consigna. Si quieren hacer puntos estrella, deben hacerlo en una branch aparte.

Consigna

En este laboratorio, vamos a modificar un sistema de archivos FAT en espacio de usuario para agregar nuestras propias funcionalidades. La implementación original que

tomamos como base está en este repositorio, aunque hemos introducido muchos cambios.

Parte 1: Vigilancia

Vamos a crear un archivo secreto que inicialmente se llamará `/fs.log` en donde se guardará un registro de todas las operaciones de lectura y escritura de archivos realizadas por el usuario. Deberán:

1. **Crear el archivo para guardar los logs:** tiene que ser creado luego de montar el volumen, y tienen que ser cuidadosos de no crearlo varias veces. Al desmontar y volver a montar el volumen, el archivo tiene que ser leído correctamente. Pueden usar `mknod`.

2. **Escribir registros:** tienen que escribir en el archivo de logs un registro de las operaciones que realiza el usuario. Sólo deben registrar llamadas a `read` o `write`. Les proveemos del código que arma el string para escribir en la función `fat_fuse_log_activity`, pero deben completarla con la lógica que realiza la escritura.
3. **Ocultar el archivo en nuestro FS:** el archivo tiene que ser invisible a comandos como `ls`, pero se debería poder leer/escribir con comandos como `cat` o redirecciones. Para esto consideren a qué `fuse_operation` llamará `ls` para hacer allí las modificaciones correspondientes.

NOTA: toda esta parte se resuelve en el archivo fat_fuse_ops.c

Parte 2: Ocultando el archivo en otros FS

Para que la vigilancia sea verdaderamente transparente, debería funcionar para otras implementaciones de FAT (como la nativa de Linux). Sin embargo, debemos ser cuidadosos de no alterar el formato FAT de ninguna manera. La imagen debe poder ser montada con otras herramientas, pero tenemos que lograr que ignoren el archivo `fs.log`, pero que no lo sobreescriban tampoco (en la mayoría de los casos, nunca lo podremos ocultar al 100%). Para esto, les proponemos que el archivo `fs.log` tenga las siguientes características:

- Que el primer byte de su dentry sea `0xe5`. Esto marca el archivo como "pendiente para ser eliminado", de forma que el FS ignora la entrada, pero no la reutiliza hasta que haya sido propiamente eliminada por otras herramientas.
- Que el campo `attribute` de la `dentry` sea `System`, `Device` o `Reserved`, lo cual indica que la entrada no debe ser modificada por las herramientas del FS. De esta manera, logramos que la entrada nunca sea efectivamente eliminada y marcada como libre.

Para implementar esta parte, trabajarán con el nivel de abstracción más bajo del sistema: el TAD `fat_file`. Este tipo es el que guarda la información del archivo, y tiene funciones asociadas que la leen y escriben directamente en la imagen (por ejemplo, `write_dir_entry`). Deberán:

- Modificar la entrada de directorio (`dentry`) del archivo secreto al guardarla en la imagen.
- Asegurarse de que al desmontar y volver a montar la imagen no se ignore esta entrada, que ahora está marcada para ser borrada, y que se lea correctamente el nombre del archivo.

Parte 3: Unlink y rmdir

La siguiente tarea será agregar al filesystem la funcionalidad para borrar archivos y directorios. Tanto para `unlink` como para `rmdir` hay tres tareas a tener en cuenta:

1. Marcar como libres los clusters correspondientes (incluido el primero)
2. Actualizar la entrada de directorio correspondiente (y acordarse de guardar este cambio a disco!)
3. Actualizar el árbol de directorios en memoria.

Para la interfaz de alto nivel pueden ayudarse viendo otras `fuse_operations` con comportamiento similar. Sin embargo, deben hacer una llamada a una función de más bajo nivel que se encargue de marcar como libres los clusters correspondientes e

indicar que la entradas de directorio del elemento están “listas para eliminar”. Esto último se puede lograr escribiendo el carácter `0xE5` en el primer byte de la `dentry`, que corresponde al primer carácter del nombre base del archivo.

Para la tarea de liberar clusters, pueden copiar y simplificar la función `fat_file_truncate`, que corta el tamaño de un archivo y libera los clusters no utilizados. Las funciones que leen y escriben los clusters en la tabla ya están implementadas en `fat_table.c`, ¡úsenlas!

Tocar al menos `fat_fuse_ops.c` y `fat_file.c`

Parte 4: Escribiendo nuevos clusters

La implementación base presentada permite operaciones de escritura sólo en los clusters ya reservados para el archivo.

Por ejemplo, supongamos el volumen tienen clusters de 512 bytes, y que un archivo `FILE.TXT` ocupa 800 bytes. Luego, `FILE.TXT` tiene reservados 2 clusters de datos. Si queremos escribir 500 bytes desde un offset de 700, entonces necesitamos agregar un cluster más, ya que 500 + 700 > 512 * 2. La función `fat_file_pwrite`, tal y como se las entregamos, sólo escribirá los bytes que entren en los clusters disponibles, es decir 512 * 2 - 700.

La tarea es modificar el filesystem para que agregue nuevos clusters al archivo, y realice la escritura completa. La función `fat_file_pwrite` es bastante compleja. Les recomendamos que se tomen un tiempo para entender el código, así cómo el código de `fat_table.c`.

Algunas recomendaciones

- Utilicen generosamente las funciones de logging, como `fat_table_print`, `fat_file_print_dentry` o `fat_tree_print_preorder` para ayudarse a debuggear.
- Concéntrense sólo en las partes relevantes del problema, y no en todos los detalles de bajo nivel.
- Usen como modelo funciones similares que ya estén implementadas.
- El sistema no es perfecto y probablemente tenga muchos errores. Si encuentran algo que parece sospechoso, avisen a la cátedra para que lo revisemos.

Puntos estrella

1. [Fácil pero lleva tiempo] Agregar una opción al montar el filesystem que permita mostrar/ocultar el archivo de logs. En este caso, también se debería ocultar el archivo ante cualquier tipo de operación si el flag no está presente.
2. Escribir tests para asegurar que implementaron correctamente las partes 1 y 2.
3. Agregar registro de palabras censuradas

   Deben modificar el sistema original para que, al leer o escribir un archivo, registre también si alguna de las siguientes palabras se encontraba en el contenido leído/escrito.

```
{“Oldspeak”, “English”, “revolution”, “Emmanuel”, “Goldstein”}
```

   Por ejemplo, si montamos nuestro filesystem y leemos el archivo `1984.txt`, en el archivo de logs debe guardarse:

```
25-10-2021 17:43  milagro    /1984.txt   read [Oldspeak, English, revolution, Emmanuel, Goldstein]
```

   a. Las palabras deben estar contenidas en un arreglo, y pueden agregar todas las que quieran. Pueden definir el arreglo de palabras y otras constantes en el `big_brother.h`.
   b. Ignorar el caso en que la palabra aparezca más de una vez, o cuando haya una diferencia entre mayúsculas y minúsculas.
4. Agregar una caché en memoria con el número de los clusters de datos leído de un archivo, que se llena a medida que se leen. De esta forma, evitamos recorrer la tabla FAT por cada operación de lectura o escritura. Cuando el número de file descriptors abiertos para el archivo es 0, la caché se destruye. Les recomendamos implementarlo agregando una `GList` a la estructura `fat_file_s`.
5. [Difícil pero probablemente gratificante] Agregar soporte para nombres de archivo de más de 8 caracteres. Hay que implementar la lectura de nuevos tipos de clusters, no sólo datos y directorios.
6. Modificar el TAD `fat_table` (entre otras cosas) para que utilice correctamente la segunda copia de la tabla FAT.
7. [Masomenos fácil] Agregar un nuevo cluster al directorio cuando este se llena de entradas.

Instrucciones para compilar y correr

Para compilar el proyecto tienen que instalar las siguientes dependencias:

```
$ sudo apt-get install libfuse-dev
$ pip3 install fs
$ pip3 install unitest
```

Además tienen que tener instalado `python3`, que suele venir por defecto en linux.

Para compilar el código tienen que correr

```
$ make
```

Para obtener una imagen para montar tienen que correr

```
$ make image
```

Una vez tienen la imagen pueden montar con fat-fuse usando alguna de estas opciones:

1. `$ make mount`
2. `$ ./fat-fuse test.img mnt`
3. `$ ./fat-fuse -f test.img mnt (corre en foreground, útil para debuggear y con Ctrl+C se desmonta)`

Para desmontar pueden usar alguna de estas opciones:

1. `$ make unmount`
2. `$ umount mnt`
3. `$ fusermount -u mnt`

Para correr los tests hagan:

```
$ make testfs
```

Si no tienen el SO en inglés prueben con correr antes:

```
$ export LC_ALL=C
```

Los archivos con imágenes que están en `/resources` no los deberían tocar. Si los tocan háganles `git checkout` para restaurarlos a la versión original

Revisiones

2023 Tadeo Cocucci
2022 Milagro Teruel, Tadeo Cocucci
2019, 2020 y 2021 Milagro Teruel, y aportes de código de Christian Moreno (genio!)
2018 Cristian Cardellino, Milagro Teruel
Tomamos mucho de:

- 2012, Nicolás Wolovick
- Original 2009-2010, Rafael Carrascosa, Nicolás Wolovick

Errores que vamos encontrando

- PONGAN COMENTARIOS SI ENCUENTRAN MÁS! O si corrigen, eso está genial.
- Al tratar de escribir más de 4096 bytes en una única llamada, hay un segmentation fault. Ya sabemos cómo se resuelve, pero no se preocupen por eso en este lab.
- El manejo de errores no está 100% completo. Por ejemplo, una función puede fallar y devolver `NULL`, pero la función que la llama no contempla ese caso.
- En la función `fill_time`, en `fat_util.c`, la función `gmtime_r` trabaja con el horario utc (gmt 0), por lo tanto, eso genera que al modificar o un crear un archivo, el campo last_modified_time esté 3hs adelantado a la hora de la pc (suponiendo que la tenemos en gmt -3), esto provoca que al hacer `ls -l`, aparezca el año en vez de la hora en la cual se editó por última vez. [Créditos: Valentino Beorda]
- En ningún lugar se hacen chequeos para ver que los nombres de archivo sean de menos de 8 caracteres y sus extensiones de menos de 3, eso causa varios

errores, por ejemplo, a veces se sobreescriben punteros del `fat_file` y después cuando se trata de acceder se produce un segmentation fault.
- Puede haber memory leaks, sobre todo los los strings utilizados para los nombres de los archivos.
- Puede haber problemas con archivos creados con espacios.

- agregar +1 en la guarda del `while`, de la función `fat_table_print()`, para que imprima archivos de 1 solo cluster
- Si no se hace `ls -a` cada directorio en el path, entonces el archivo no se carga en el árbol de directorios.
- Los nombres con “init” son confusos y no reflejan que la función está cargando a partir de la imagen en lugar de crear recursos nuevos. Cambiar por “load” o similar?
- Crédito: Guillermo de Ipola. Al crear directorios, la basura que está en el disco puede re-interpretarse como dentries. Una solución posible es llenar el cluster con 0s al crear el directorio.

Agregar `fflush()` antes de desmontar el archivo para que exija escribir el disco, planteado en consulta con el profe Matias, agregarlo en `fat_volume_umont` en `munmap` (o cerca) ya que extrañamente tarda en leer los datos cuando se desmonta rápido y se vuelve a montar (en mi caso tardaba en leer que se había “borrado un archivo”)
- Cambiar el nombre de la función `fat_file_cmp_path` para que devuelva un booleano que tenga sentido de acuerdo al nombre.