Proyecto 2: libfs - Una Biblioteca para un Sistema de Archivo en Espacio de Usuario

Objetivos

- Entender cómo funciona un sistema de archivos, específicamente la estructura y administración del almacenamiento.
- Entender cómo lidiar con algunas situaciones que pueden presentar en algunos sistemas de archivos.

Resumen

En este proyecto, los estudiantes desarrollarán una biblioteca en el espacio de usuario llamada `libfs`, la cual implementará una pequeña porción de un sistema de archivos. El sistema de archivos desarrollado debe estar construido dentro de la biblioteca, la cual podrá ser enlazada desde otras aplicaciones para obtener acceso a archivos. La biblioteca que desarrollarán, a su vez, se enlaza con una biblioteca que simula e implementa un disco físico, `libdisk`, la cual será provista por el GDSO.

Especificaciones de la biblioteca `libfs`:

Se iniciará este proyecto describiendo el API del sistema de archivos que proveerá `libfs`. Existen tres partes en el API: dos llamadas genéricas al sistema de archivos, un conjunto de operaciones sobre archivos, y las aplicaciones que se enlazan con `libfs` con la finalidad de acceder y probar el sistema de archivos implementado.

La bilioteca `libfs` será probada y evaluada tomando en cuenta su funcionalidad y el manejo correcto de errores. Cuando un error ocurra, su biblioteca debe colocar en la variable global `os_errno` el valor definido en la API.

Definición de la API

API genérico del sistema de archivos

- `int fs_boot(char *path);`

`fs_boot` debe ser llamado exactamente una vez antes que cualquier otra función de `libfs`. Esta toma un argumento único, el nombre de archivo de la "imagen de disco" donde estará almacenado el sistema de archivos. De no existir, el archivo debe ser creado. Si la invocación es satisfactoria la función devuelve 0, en caso contrario devuelve -1 y el valor de `os_errno` será `E_GENERAL`.

- `int fs_sync(void);`

`fs_sync` se asegura de que el contenido del sistema de archivos se almacene de manera persistente en el disco. Más detalles de cómo se logra esto utilizando `libdisk` se encuentra disponible en las próximas secciones. Si la invocación es satisfactoria la función devuelve 0, en caso contrario devuelve -1 y el valor de `os_errno` será `E_GENERAL`.

API para la manipulación de archivos

Para el acceso a los archivos asumimos que la longitud máxima de un nombre de archivo es de 16 bytes (15 caracteres terminados con `\0`).

- `int file_create(char *file);`

`file_create` crea un archivo nuevo con el nombre indicado en `file`. Si el archivo existe, se retorna -1 y se asigna el valor `E_CREATE` en `os_errno`. Note que el archivo no será abierto después de la invocación a la función de creación, es decir, `file_create` simplemente crea un archivo en el disco de tamaño 0. De tener éxito, la función retorna 0; en caso contrario la función devuelve -1 y `os_errno` debe valer `E_CREATE`.

- `int file_open(char *file);`

`file_open` se encarga de abrir un archivo `file` y retornar el descriptor de archivo (`int` mayor o igual a 0), el cual puede ser usado para leer o escribir datos en dicho archivo. Si el archivo no existe, la función retorna -1 y el valor de `os_errno` debe ser `E_NO_SUCH_FILE`. Si ya se ha alcanzado el número de archivos abiertos, la función debe devolver -1 y `os_errno` debe valer `E_TOO_MANY_OPEN_FILES`.

- `int file_read(int fd, void *buffer, int size);`

`file_read` debe leer `size` bytes del archivo referenciado por el descriptor de archivo `fd`. Los datos leídos deben ser almacenados en el área de memoria apuntada por `buffer`. Todas las lecturas deben iniciar en la posición actual del puntero de archivo, y el puntero de archivo debe ser actualizado después de realizar cada lectura a una nueva posición. Si el archivo no se encuentra abierto, la función retorna -1, y el valor de `os_errno` será `E_BAD_FD`. Si el archivo se encuentra abierto, el número total de bytes que se han leído debe ser retornado, siendo este valor menor o igual a `size` (el valor de retorno podría ser menor a `size`, por ejemplo, si se alcanza el final del archivo). Si el puntero del archivo se encuentra al final del archivo, la función retornará 0, incluso tras llamadas sucesivas a `file_read`.

- `int file_write(int fd, void *buffer, int size);`

`file_write` debe escribir `size` bytes desde `buffer` en el archivo referenciado por `fd`. Todas las escrituras deben iniciar en la posición actual del puntero de archivo, y el puntero de archivo debe ser actualizado después de realizar la escritura (posición actual del puntero de archivo + `size`). Note que las escrituras es la única manera de hacer crecer el tamaño del archivo. Si el archivo no se encuentra abierto, la función retorna -1 y `os_errno` debe valer `E_BAD_FD`. Una vez que se haya realizado la escritura, todos los datos deben ser escritos en disco y el valor de `size` debe retornarse. Si la escritura no puede ser completada (debido a falta de espacio), la función debe retornar -1 y `os_errno` valdrá `E_NO_SPACE`. Finalmente, si el archivo excede el tamaño máximo permitido para un archivo, la función debe devolver -1 y `os_errno` valdrá `E_FILE_TOO_BIG`.

- `int file_seek(int fd, int offset);`

`file_seek` debe actualizar la posición actual del puntero de archivo. Esta posición está dada como un desplazamiento `offset` desde el inicio del archivo. Si `offset` es más grande que el tamaño del archivo o su valor es negativo, la función debe devolver -1 y el valor de `os_errno` será `E_SEEK_OUT_OF_BOUNDS`. Si el archivo no se encuentra abierto, la función devolverá -1 y `os_errno` valdrá `E_BAD_FD`. Una vez que la función tenga éxito, esta debe devolver la nueva posición del puntero de archivo.

- `int file_unlink(char *file);`

Esta función debe borrar el archivo referenciado por `file`; este `unlink` incluye la eliminación del nombre del archivo correspondiente, y la liberación de cualquier bloque e inodo que el archivo este utilizando. Si el archivo no existe, la función retorna -1 y establece el valor de `os_errno` a `E_NO_SUCH_FILE`. Si el archivo se encuentra abierto, la función retorna -1 y `os_errno` valdrá `E_FILE_IN_USE` (y no se eliminará el archivo). Si la invocación es exitosa, la función retornará 0.

Consideraciones

Cuando se lee o escribe un archivo, usted debe implementar la noción del puntero actual de archivo. La idea es simple: luego de abrir el archivo, el puntero de archivo se coloca al inicio del archivo (byte 0). Si el usuario lee n bytes del archivo, el puntero de archivo debe ser actualizado a n. Una nueva lectura de m bytes debe devolver el desplazamiento correspondiente desde la posición n, esto es $n+m$. De esta manera, llamadas sucesivas de lectura o escritura podrán leer o escribir todo el archivo. Por supuesto, `file_seek` existe para actualizar la posición del puntero de archivo de forma explícita.

Detalles de implementación

Abstracción de disco La primera pregunta que usted debe hacerse es: ¿dónde voy a almacenar todos los datos del sistema de archivos? Un sistema de archivos real debe almacenarse en disco, pero dado que en este proyecto se implementará en espacio de usuario, nosotros utilizaremos un disco simulado. En los archivo `libdisk.c` y `libdisk.h` usted encontrará la implementación o simulación de disco con la cual debe interactuar en este proyecto.

El disco que se les provee consta de un número de sectores (`NUM_SECTORS`), cada uno de tamaño específico (`SECTOR_SIZE`), ambos valores definidos como constantes en el archivo `libdisk.h`. Así, será necesario que usted use estos valores en la estructura de su sistema de archivos. El modelo del disco es bastante simple; en general, su implementación de sistema de archivos permitirá realizar lecturas y escrituras, lo cual se traducirá en lecturas o escrituras de sectores del disco. En realidad, las lecturas y escrituras acceden a un arreglo en memoria para realizar la manipulación de los datos; mientras que otros aspectos del API de disco le permitirán salvar el contenido de su sistema de archivos en un archivo regular en Linux, para luego poder restaurarlo.

Descripción de API de disco

- `int disk_init(void);`

`disk_init` debe ser invocado solo una vez por el SO antes de que otra operación de disco ocurra.

- `int disk_load(char *file);`

`disk_load` es llamada para cargar el contenido del sistema de archivos file en memoria. Esta rutina (y `disk_init`) probablemente será ejecutada por su biblioteca en tiempo de arranque, por ejemplo, durante `fs_boot`.

- `int disk_save(char *file);`

`disk_save` salva la vista actual del disco en memoria en una archivo apuntado por file. Esta rutina debe ser usada para salvar el contexto del disco en un archivo real, para que luego pueda recuperarse la información del disco. Esta rutina debería ser invocada por `fs_sync`.

- `int disk_write(int sector, char *buffer);`

`disk_write` escribe los datos de `buffer` en el sector especificado por `sector`. Se asume que `buffer` tiene el tamaño exacto de un sector.

- `int disk_read(int sector, char *buffer);`

`disk_read` lee el contenido de un `sector` y lo coloca en el buffer especificado por `buffer`. Como en `disk_write` se asume que el `buffer` tiene el tamaño exacto de un sector.

Todas las operaciones retornan 0 en caso de éxito y -1 en caso de falla. Si existe una falla, la variable `disk_errno` tendrá el valor apropiado de la falla.

Estructura de datos on-disk Una parte importante y esencial en el sistemas de archivos es la comprensión de las estructuras de datos; esto debido a que existen diferentes maneras de diseñar e implementar un disco y su sistema de archivos. A continuación, se presenta una aproximación sencilla a las estructuras requeridas, lo cual será de vital importancia para que inicie la programación.

En primer lugar, en necesario almacenar alguna información genérica en el propio disco sobre el sistema de archivos, generalmente en un bloque llamado *superblock*. Este debe estar ubicado en una posición bien conocida del disco, la cual habitualmente es el primer bloque. En este proyecto, usted no necesita almacenar mucho allí. De hecho, usted solo debe almacenar una cosa en el *superblock*, un *magic number*. Elija cualquier número, y cuando usted inicialice el nuevo sistema de archivos, escriba el *magic number* en el *superblock*. Con base a esta explicación, asegúrese de leer el *magic number*

almacenado en el *superblock*, y adicionalmente verifique que su valor sea el mismo. Si el valor del *magic number* ha cambiado, o no puede accederse, usted debe asumir que el sistema de archivos se ha corrompido, y por lo tanto no puede utilizarse.

Para lidiar con directorios (en este caso, solo hay uno) y archivos, usted necesita dos tipos de bloques: inode blocks y data blocks. Primero, examinemos los inodos. Para cada inodo, usted necesita lidiar con al menos dos cosas para cada archivo. La primera, es que usted debe lidiar con el tamaño del archivo. La segunda, es que usted necesita saber qué tipo de archivo maneja (un archivo normal o un directorio). Tercero, usted debe saber que bloques se han asignado a un archivo. Para este proyecto, usted puede asumir que el tamaño máximo de un archivo es de 30 bloques. De esta forma, cada inodo debe contener 1 entero (size), 1 entero (type), y 30 apuntadores (a bloques). Usted también debe notar que cada inodo es más pequeño que un sector de disco, por lo tanto, usted debe colocar múltiples inodos en el mismo sector de disco para salvar espacio.

Nota: no es necesaria una implementación completa de directorios. Solo existirá un inodo de tipo directorio, para el directorio raíz.

Lo segundo, son los bloques de datos. Asuma que cada bloque de datos es del mismo tamaño que un sector de disco. De este modo es directo inferir que parte del disco será dedicado al almacenamiento de bloques de datos. Por supuesto, que usted también tendrá que seguir la pista de que inodos han sido asignados, y que bloque de datos están siendo usados y por quien. Para implementar esto, usted probablemente necesite utilizar un mapa de bits para seguir la pista de ambas asignaciones, por ejemplo, el primer bloque después del superblock podría ser el mapa de bits de los inodos, y el segundo bloque el mapa de bits de los bloques de datos.

Los mapas de bit de inodos y de bloque de datos son arreglos de bits que indican si la estructura está ocupada o desocupada.

La estructura del sistema de archivo es el siguiente: Súper-bloque, Mapa de bit de Inodos, Mapa de bit de Bloque de Datos, Bloques de inodos y Bloques de Datos.

Tabla de archivos abiertos Cuando un proceso abre un archivo, primero se lleva a cabo la ubicación de dicho archivo en disco. Al finalizar la búsqueda, usted necesitará de al menos una parte de la información obtenida con la finalidad de realizar las lecturas y escrituras de manera eficiente (sin la necesidad de realizar las mismas búsquedas una y otra vez). Esta información debe mantenerse en la tabla de archivos abiertos. Cuando un proceso abre un archivo, se asignará la primera entrada de esta tabla, así el primer archivo abierto obtendrá el primer slot en esta estructura, y se debe devolver el descriptor de archivo 0. El segundo archivo abierto (si el primero continua abierto) debe devolver el descriptor 1, y así sucesivamente. Cada entrada en la tabla debe contener la información necesaria para poder seguir realizando las lecturas y escrituras de manera eficiente, piense en esto antes de diseñar e implementar esta tabla. El tamaño de esta tabla es fijo y tiene un tamaño máximo de 20 entradas.

Persistencia en disco La abstracción de disco le provee un mecanismo para salvaguardar el estado actual del disco en memoria en un archivo al invocar `disk_save`. Es por esto, que usted debe invocar a la función `disk_save` para mantener la persistencia de su sistema de archivos. En un SO real, los datos son almacenados en disco luego de cortos instantes de tiempo, esto con la intención de prevenir perdida de datos. Sin embargo, en este proyecto, usted solo deberá invocar `disk_save` cuando la función `fs_sync` sea utilizada por la aplicación que usa la implementación de la biblioteca `libfs`. Es necesario llamar a `disk_read` y a `disk_write` por cada `file_read`, `file_write`, y otras operaciones inherentes al sistema de archivos que requieran interacción con el disco.

Arranque del sistema Cuando su SO inicie, usted debe pasar el nombre del archivo que contendrá al disco virtual. Esto es el nombre del archivo que contendrá toda la información del disco simulado. Si el archivo existe y usted desea cargarlo (usando `disk_load`), entonces usted debe realizar acciones pertinentes para garantizar que el archivo simula un disco válido. Por ejemplo, el tamaño del archivo debe ser equivalente a NUM_SECTORS por SECTOR_SIZE, y el superblock debe contener la información correcta. Si luego del análisis correspondiente se detecta alguna discrepancia, usted debe reportar el error y abortar.

Adicionalmente, considere la siguiente situación: el archivo que simula el disco no existe, significa que un nuevo disco debe ser creado y es necesario inicializar su superblock, además de crear el directorio raíz (vacío) en el sistema de archivos. En este caso en particular, usted debe usar la función `disk_init` seguida de algunas operaciones `disk_write` para poder inicializar el disco, y luego de esto usted debe hacer persistente los cambios invocando a `disk_save`.

Materiales provistos La estructura de directorios para este proyecto es la siguiente:

```
proyecto_fs/
|-- bin
|-- Makefile
|-- make.libdisk
|-- make.libfs
|-- make.main
|-- obj
|-- README
|-- shobj
|-- src
    |-- libdisk.c
    |-- libdisk.h
    |-- libfs.c
    |-- libfs.h
    |-- main.c
```

Se han implementado makefiles separados para cada una de las partes más importantes de este código: la implementación de la simulación de disco, `libdisk`, el esqueleto de `libfs`, que deben implementar, y un `main`, para hacer las pruebas de la API del sistema de archivos. Estos makefiles han sido integrados en `Makefile`. Llamando `make` sin ningún parámetro ejecuta el `makefile` de cada una de estas partes, compilando `main` dentro de `bin`, y las bibliotecas compartidas en `shobj`.

Para este proyecto se requiere que usted implemente cada una de las funciones en `libfs.c`, usando la API de disco descrita en este enunciado. Se recomienda modificar `main.c` para probar `libfs` durante todas las fases de su desarrollo.

Directrices importantes

- Tamaño máximo de un archivo: 30 bloques.
- No se permiten colisiones de nombre de archivo.
- Los caracteres válidos para nombrar un archivo incluyen: caracteres alfanuméricos, puntos, guiones y guiones bajos.
- Su implementación debe soportar un máximo de 1000 archivos, que debe estar definida en un macro MAX_FILES.
- No modificar código en la interfaz de la API de disco `libdisk.h` ni modificar su implementación, `libdisk.c`.
- No modificar código en la interfaz de la API del sistema de archivos, `libfs.h`.
- La ejecución de las pruebas debería hacerse de la siguiente manera: `./bin/main [archivo de disco]`.

Evaluación

- Funcionalidad. El 100 % de la calificación del proyecto se apoyará en la implementación de todas las funcionalidades descritas en este enunciado, así como la capacidad del estudiante de explicar el diseño e implementación de todas las funcionalidades el día pautado para la revisión del proyecto.
- Las copias serán penalizadas con la nota mínima.

- El proyecto podrá ser realizado tanto individualmente como en parejas.
- La fecha de entrega y revisión será anunciada por el GDSO vía email y sitio web de la materia.