PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2333 — Sistemas Operativos y Redes — 2/2020

Proyecto 1

Miercoles 5 de Mayo, 2021

Fecha de Entrega: Viernes 28 de Mayo, 2021

Objetivos

- Conocer la estructura de un sistema de archivos y sus componentes.
- Implementar una API para manejar el contenido de un disco virtual con particiones a partir de su sistema de archivos.

OldSchool FileSystem

Luego de intentar corregir la Tarea 1 de Sistemas Operativos, los ayudantes se dieron cuenta que los múltiples *fork-bombs* provocados por el código de los alumnos terminaron sobreescribiendo los *drivers* de sus computadoras que leían el sistema de archivos. Esto les dió la gran oportunidad de diseñar un novedoso sistema de archivos, el cual debido a problemas de organización del tiempo no fueron capaces de programar. Por esto, les han pedido que implementen el sistema de archivos, mientras ellos se encargan de ~~jugar~~ corregir.

Introducción

Los sistemas de archivos nos permiten organizar nuestros datos mediante la abstracción de archivo y almacenar estos de manera ordenada en dispositivos de almacenamiento secundario que se comportan como un dispositivo de bloques. En esta tarea tendrán la posibilidad de experimentar con una implementación de un sistema de archivos simplificado sobre un disco virtual, el cual puede contener desde 1 a 128 particiones. Este disco virtual será simulado por un archivo en el sistema de archivos real. Deberán leer y modificar el contenido de este disco mediante una API desarrollada por ustedes.

Estructura de sistema de archivos `osfs`

El sistema de archivos a implementar será denominado `osfs`. Este sistema almacena archivos en bloques mediante asignación indexada.

El disco virtual es un archivo en el sistema de archivos real. Este disco está organizado en conjuntos de Bytes denominados **particiones**, las cuales están definidas por una **master boot table**. Vada partición está dividida en **bloques**.

Las características generales del disco son las siguientes:

- Tamaño del disco: 4 GB + 1 KB. De ellos, 4 GB están divididos en bloques.
- Tamaño de bloque: 2 KB. El disco contiene un total de $2^{21} = 2097152$ bloques, identificados desde el 0 hasta el 2097151.

- Cada bloque posee un número secuencial, que se puede almacenar en 3 Bytes (24 Bits) para lo cual se puede guardar en un `unsigned int`. **Este valor corresponde a su identificador absoluto.** Además, cada bloque tiene un identificador relativo a su partición, que va de 0 hasta la cantidad de bloques de la partición.
- El primer KB del disco contiene la **master boot table**, mientras que los 4 GB restantes están separados en bloques.

Cada bloque en el disco pertenece a uno de cuatro tipos de bloque: directorio, *bitmap*, índice, y datos. La única excepción a esto es la **master boot table**.

Master Boot Table (MBT). Esta tabla siempre será el primer KB del disco y tiene toda la información de las particiones de este. Está separada en entradas las cuales siguen la siguiente estructura:

- Tamaño de entrada: 8 Bytes.
- Cantidad de entradas: 128.
- Primer bit de cada entrada es el **bit de validez**. Si es 1 indica que esta entrada es una partición válida, si es 0 la entrada no es válida.
- Siguientes 7 bits representan el número identificador único de la partición, que puede ser un número del 0 al 127.
- Los siguientes 3 Bytes contienen el identificador absoluto del primer bloque de la partición, que siempre debe ser su bloque directorio.
- Los últimos 4 Bytes contienen la cantidad de bloques de la partición. La mínima cantidad de bloques es 16384 y la cantidad máxima es 131072.

Toda partición es inicializada con un bloque de directorio seguido por la cantidad mínima de bloques bitmap necesarios para cubrir su cantidad de bloques, que pueden ser como mínimo 1 y como máximo 8. Cabe recalcar que estos bloques se consideran dentro del tamaño de la partición.

Bloque de directorio. Este bloque está compuesto por una secuencia de entradas de directorio, donde cada entrada de directorio ocupa 32 Bytes, por lo tanto, puede haber como máximo 64 entradas en un bloque de directorio. Una entrada de directorio contiene:

- 1 Byte. Indica si la entrada es inválida (`0x00`) o un archivo (`0x01`).
- 3 Byte. Identificador relativo del bloque índice del archivo. Que el identificador sea relativo a la partición significa que este puede ser desde 0 hasta la cantidad total de bloques de la partición.
- 28 Bytes. Nombre de archivo, expresado usando caracteres de letras y números ASCII (*8-bit*), incluyendo la extensión del tipo de archivo (`.png`, `.txt`, etc.) si corresponde. En caso de que el nombre del archivo ocupe menos de 28 Bytes, el resto de los Bytes deberán ser iguales a `0x00`, cabe recalcar que la lectura y escritura se debe realizar de izquierda y derecha.

El **primer bloque** de una partición **siempre** corresponderá al directorio raíz de su sistema de archivos, es decir, siempre es un bloque de directorio.

Bloque de bitmap. Los bloques de *bitmap* corresponden siempre a los bloques después de un bloque de directorio. El contenido de los bloques es el *bitmap* de la partición. Cada Bit del *bitmap* indica si el bloque correspondiente en la partición está libre (0) o no (1). Por ejemplo, si el primer Bit del primer bloque de *bitmap* de una partición tiene el valor 1, quiere decir que el primer bloque de esa partición está utilizado.

- El *bitmap* contiene 1 Bit por cada bloque de la partición, sin importar su tipo. Estos bloques de disco, además de los bloques directorio raíz deben considerarse siempre como ocupados.

- El *bitmap* debe reflejar el estado de la partición y se debe mantener actualizado respecto al estado de los bloques.

Bloque índice. Un bloque índice es un bloque que contiene la metadata de un archivo y la información necesaria para acceder al contenido de este. El primer bloque de un archivo siempre es un bloque índice. Está compuesto por:

- 5 Bytes para el tamaño del archivo.
- 2043 Bytes, reservados para 681 punteros, estos punteros también son relativos a la partición. Cada uno apunta a un bloque de datos.

Bloque de datos. El contenido de un archivo se escribe únicamente en bloques de datos. Un bloque de datos utiliza todo su espacio para almacenar el contenido (datos) de un archivo. Estos bloques contienen directamente la información de los archivos. Una vez que un bloque ha sido asignado a un archivo, se asigna de manera **completa**. Es decir, si el archivo requiere menos espacio que el tamaño del bloque, el espacio no utilizado sigue siendo parte del bloque (aunque no contenga datos del archivo) y **no puede ser parcialmente asignado** a otro archivo.

La lectura y escritura de Bytes en los bloques de datos **deben** ser realizadas en orden **big endian** para mantener consistencia entre todos los sistemas de archivos implementados.

API de `osfs`

Para poder manipular los archivos del sistema (tanto en escritura como en lectura), deberá implementar una biblioteca que contenga las funciones necesarias para operar sobre el disco virtual. La implementación de la biblioteca debe escribirse en un archivo de nombre `os_API.c` y su interfaz (declaración de prototipos) debe encontrarse en un archivo de nombre `os_API.h`. Para probar su implementación debe escribir un archivo con una función `main` (por ejemplo, `main.c`) que incluya el *header* `os_API.h` y que utilice las funciones de la biblioteca para operar sobre un disco virtual que debe ser recibido por la línea de comandos. Dentro de `os_API.c` se debe definir un `struct` que almacene la información que considere necesaria para operar con el archivo. Ese `struct` debe ser nombrado `osFile` mediante una instrucción `typedef`. Esta estructura representará un *archivo abierto*.

La biblioteca debe implementar las siguientes funciones.

Funciones generales

- `void os_mount(char* diskname, int partition)`. Función para montar el disco en una partición en especifico. Establece como variable global la ruta local donde se encuentra el archivo `.bin` correspondiente al disco y también guarda como variable global la partición a montar. Cabe recalcar que toda la API debe funcionar tomando la partición especificada como punto de referencia. La función debe poder ser llamada múltiples veces si uno desea cambiar el disco o bien cambiar la partición en la cual se está trabajando.
- `void os_bitmap(unsigned num)`. Función para imprimir el *bitmap*. Cada vez que se llama esta función, imprime en `stderr` el estado actual del bloque de *bitmap* correspondiente a `num` en hexadecimal. Si se ingresa $\texttt{num} = 0$, se debe imprimir **el bitmap completo**. Se debe imprimir además una línea con la cantidad de bloques ocupados, y una segunda línea con la cantidad de bloques libres.
- `int os_exists(char* filename)`. Función para ver si un archivo existe. Retorna `1` si el archivo existe y `0` en caso contrario.
- `void os_ls()`. Función para listar los elementos de la partición actual. Imprime en pantalla los nombres de todos los archivos.

Funciones Master Boot Table (MBT)

- `os_mbt()`. Función que muestra en pantalla las particiones válidas de la MBT.

- `os_create_partition(int id, int size)`. Función que crea una nueva partición con el `id` y tamaño especificado. Se debe crear la partición en el primer espacio contiguo mayor o igual a `size` disponible que se encuentre y se debe corroborar que su `id` no este ocupado.
- `os_delete_partition(int id)`. Función que elimina la partición de la MBT. No es necesario llenar de ceros la partición, basta con cambiar el bit de validez en la fila correspondiente de la MBT.
- `os_reset_mbt()`. Función que elimina todas las particiones de la MBT.

Funciones de manejo de archivos

- `osFile* os_open(char* filename, char mode)`. Función para abrir un archivo. Si `mode` es `'r'`, busca el archivo con nombre `filename` y retorna un `osFile*` que lo representa. Si `mode` es `'w'`, se verifica que el archivo no exista y se retorna un nuevo `osFile*` que lo representa.
- `int os_read(osFile* file_desc, void* buffer, int nbytes)`. Función para leer archivos. Lee **los siguientes** `nbytes` desde el archivo descrito por `file_desc` y los guarda en la dirección apuntada por `buffer`. Debe retornar la cantidad de Bytes efectivamente leídos desde el archivo. Esto es importante si `nbytes` es mayor a la cantidad de Bytes restantes en el archivo. La lectura de `read` se efectúa desde la posición del archivo inmediatamente posterior a la última posición leída por un llamado a `read`.
- `int os_write(osFile* file_desc, void* buffer, int nbytes)`. Función para escribir archivos. Escribe en el archivo descrito por `file_desc` los `nbytes` que se encuentren en la dirección indicada por `buffer`. Retorna la cantidad de Bytes escritos en el archivo. Si se produjo un error porque no pudo seguir escribiendo, ya sea porque el disco se llenó o porque el archivo no puede crecer más, este número puede ser menor a `nbytes` (incluso `0`). Cabe recalcar que los archivos parten con tamaño 0 y crecen a medida que se escribe en estos.
- `int os_close(osFile* file_desc)`. Función para cerrar archivos. Cierra el archivo indicado por `file_desc`. Debe garantizar que cuando esta función retorna, el archivo se encuentra actualizado en disco.
- `int os_rm(char* filename)`. Función para borrar archivos. Elimina el archivo con el nombre `filename`. Los bloques que estaban siendo usados por el archivo deben quedar libres. No es necesario rellenar de 0 los bloques que quedan liberados.

Nota: Debe respetar los nombres y prototipos de las funciones descritas. Las funciones de la API poseen el prefijo `os` para diferenciarse de las funciones de POSIX `read`, `write`, etc.

Ejecución

Para probar su biblioteca, debe usar un programa `main.c` que reciba un disco virtual (ej: `simdisk.bin`). El programa `main.c` deberá usar las funciones de la biblioteca `os_API.c` para ejecutar algunas instrucciones que demuestren el correcto funcionamiento de éstas. Una vez que el programa termine, todos los cambios efectuados sobre el disco virtual deben verse reflejados en el archivo recibido.

La ejecución del programa principal debe ser:

```
./osfs simdisk.bin
```

Por otra parte, un ejemplo de una secuencia de instrucciones que puede encontrarse en `main.c` es el siguiente:

```
os_mount("simdisk.bin", 1); // Se monta el disco.
osFile* file_desc = os_open("test.txt", 'w');
```

```
// Suponga que abrió y leyó un archivo desde su computador,
// almacenando su contenido en un arreglo f, de 300 Bytes.
os_write(file_desc, f, 300); // Escribe el archivo en el disco.
os_close(file_desc);  // Cierra el archivo. Ahora debería estar actualizado en el disco.
```

Al terminar de ejecutar todas las instrucciones, el disco virtual `simdisk.bin` debe reflejar todos los cambios aplicados. Para su implementación, puede ejecutar todas las instrucciones dentro de las estructuras definidas en su programa y luego escribir el resultado final en el disco, o bien aplicar cada comando de forma directa en el disco de forma inmediata. Lo importante es que el estado final del disco virtual sea consistente con la secuencia de instrucciones ejecutada.

Para probar las funciones de su API, se hará entrega de dos discos:

- `simdiskformat.bin`: Disco virtual formateado. Posee el bloque de directorio base y todas sus entradas de directorio no válidas (*i.e.* vacías). Se podrá descargar del servidor a través de la siguiente ruta:

  `/home/iic2333/P1/simdiskformat.bin`

- `simdiskfilled.bin`: Disco virtual con archivos escritos en él. Se podrá descargar del servidor a través de la siguiente ruta:

  `/home/iic2333/P1/simdiskfilled.bin`

Bonus

En este proyecto habrá un bonus de 10 décimas que consiste en implementar **manejo de errores estilo `errno`**. La librería `errno` funciona definiendo una variable global en la cual las funciones guardan un código de error en caso de que algo haya pasado.

Para obtener el bonus el proyecto deberá cumplir con las siguientes condiciones:

- Definición de la variable global `OS_ERROR`.
- Definición de un enum[1] que contenga todos los códigos de error. Estos deben ser lo más genéricos[2] posible.
- Definición de una función `void os_strerror` que reciba un código de error e imprima en consola un mensaje que lo explique de forma breve.
- Cada función de la API debe guardar un código de error relevante en `OS_ERROR` en caso de error.
- Escribir un breve informe que explique sus códigos de error, en que casos aplican y cuál es su mensaje asociado.

Se deben cumplir, al menos parcialmente, con todos estos puntos. La cantidad de décimas de bonus que se obtenga depende de:

- Cuanta cobertura de errores se logró.
- Que tan genéricos son los códigos de error.
- Que tan útiles son sus mensajes de errores.
- Proyectos con nota mayor o igual a $4{,}0$ pueden obtener hasta 5 décimas de bonus.
- Proyectos a nota mayor o igual a $5{,}0$ pueden obtener hasta 10 décimas de bonus.

[1] Pueden leer de enums en el siguiente enlace.
[2] Por ejemplo, en vez de tener códigos de error que validen los *input* de cada función, se puede tener uno solo que englobe ese caso.

Corrección "presencial"

A diferencia de las tareas, este proyecto será corregido de **forma "presencial"**. Se hará uso de la plataforma Google Meets para llevarla a cabo. Esto se hará de la siguiente forma:

1. Como grupo, deberán elaborar uno o más *scripts* `main.c` que hagan uso de **todas las funciones de la API que hayan implementado de forma correcta**. Si implementan una función en su librería pero no evidencian su funcionamiento en la corrección, **no será evaluada**. Es importante manejar bien los tiempos para que alcancen a mostrar todo lo que implementaron.
2. No es necesario que los *scripts* `main.c` sean subidos al servidor en la fecha de entrega, pero sí que los compartan al momento de llevar a cabo la corrección.
3. Para que el proceso sea transparente, se descargarán desde el servidor los *scripts* de su API y, en conjunto con el `main.c` elaborado, le mostrarán a los ayudantes el funcionamiento de su programa.
4. Pueden usar los archivos que deseen y de la extensión que deseen para evidenciar el funcionamiento correcto de su API. Como recomendación, los archivos `gif` y de audio son muy útiles para mostrar las limitantes del tamaño de los archivos.
5. Puede (y se recomienda) hacer uso de más de un programa `main.c`, de forma que estos evidencien distintas funcionalidades de su API.

Observaciones

- **La primera función a utilizar siempre será la que monta el disco.**
- Los bloques de datos de un archivo no están necesariamente almacenados de manera contigua en el disco, pero si deben estar en la misma partición. Para acceder a los bloques de un archivo debe utilizar la estructura del sistema de archivos.
- Debe liberar los bloques asignados a archivos que han sido eliminados. Al momento de liberar bloques de un archivo, no es necesario mover los bloques ocupados para *defragmentar* el disco, ni limpiar el contenido de los bloques liberados.
- Si se escribe un archivo y ya no queda espacio disponible en el disco virtual, debe terminar la escritura. **No** debe eliminar el archivo que estaba siendo escrito.
- Cualquier detalle **no especificado** en este enunciado puede ser abarcado mediante **supuestos**, los que deben ser indicados en el `README` de su entrega.

Formalidades

Deberá incluir un `README` que indique quiénes son los autores del proyecto (**con sus respectivos números de alumno**), cuáles fueron las principales decisiones de diseño para construir el programa, qué supuestos adicionales ocuparon, y cualquier información que considere necesaria para facilitar la corrección. Se sugiere utilizar formato **Markdown**.

La tarea **debe** ser programada en **C**. No se aceptarán desarrollos en otros lenguajes de programación.

La entrega del código de su API será a traves del servidor del curso, en la fecha estipulada. La entrega del proyecto deberá ser en una carpeta llamada `P1`, en la carpeta de cualquiera de los integrantes del curso.

El no respeto de las formalidades o un código extremadamente desordenado podría originar descuentos adicionales, los que son detallados en la sección correspondiente. Se recomienda modularizar, utilizar funciones y ocupar nombres de variables explicativos. En el caso de no entregar en la carpeta especificada, el proyecto **no se corregirá**.

Evaluación

- **1.00 pts.** Estructura de sistema de archivos[3].
  - **0.20 pts.** Representación de la master boot table.
  - **0.20 pts.** Representación de bloques directorio.
  - **0.20 pts.** Representación de bloque índice.
  - **0.20 pts.** Representación de bloque de datos.
  - **0.20 pts.** Representación de bloque de *bitmap*.
- **4.00 pts.** Funciones de biblioteca.
  - **0.30 pts.** `os_mount`.
  - **0.30 pts.** `os_bitmap`.
  - **0.30 pts.** `os_exists`.
  - **0.30 pts.** `os_ls`.
  - **0.30 pts.** `os_mbt`.
  - **0.30 pts.** `os_create_partition`.
  - **0.20 pts.** `os_delete_partition`.
  - **0.20 pts.** `os_reset_mbt`.
  - **0.20 pts.** `os_open`.
  - **0.50 pts.** `os_read`.
  - **0.50 pts.** `os_write`.
  - **0.30 pts.** `os_close`.
  - **0.30 pts.** `os_rm`.
- **1.00 pts.** Manejo de memoria perfecto y sin errores (**Valgrind**).

Descuentos

- Descuento de 0.5 puntos por subir **archivos binarios** (programas compilados).
- Descuento de 1 punto si la entrega **no tiene un** `Makefile`**, no compila o no funciona (*segmentation fault*)**.
- Descuento de 3 puntos si se sube alguno de los archivos `simdisk.bin` al servidor[4].

[3] Con "representación" no solo se espera que tengan una estructura que los represente o que lo hayan considerado en su código, sino que funcione **correctamente**.

[4] De ser posible, el descuento sería de Gúgol puntos. No puede subir los discos en **ningún momento**, no sólo para la recolección del proyecto, ya que esto puede atentar contra la estabilidad del servidor. Debido a la naturaleza de la corrección, no es necesario que prueben su API en el servidor.