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

IIC2333 — Sistemas Operativos y Redes — 2/2020

Proyecto 1

Jueves 1 de Octubre, 14:00hrs

Fecha de Entrega: Domingo 18 de Octubre, 14:00hrs
Composición: grupos de n personas, donde $3 \leq n \leq 4$
Fecha de ayudantía:Viernes 2 de Octubre, 08:30hrs

Objetivos

- Conocer la estructura de un sistema de archivos y sus componentes.
- Implementar una API para manejar el contenido de un disco virtual 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. 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 la asignación indexada con un nivel de indirección y permite la existencia de varios directorios.

El disco virtual es un archivo en el sistema de archivos real. Este disco está organizado en conjuntos de Bytes denominados **bloques**, de acuerdo a las siguientes características:

- Tamaño del disco: 2 GB.
- Tamaño de bloque: 2 KB. El disco contiene un total de $2^{20} = 1048576$ bloques, identificados desde el 0 hasa el 1048575.
- Cada bloque posee un número secuencial, que se puede almacenar en un `unsigned int` de 4 Bytes (32 Bits). **Este valor corresponde a su puntero.**

Cada bloque en el disco pertenece a uno de cinco tipos de bloque: directorio, *bitmap*, índice, direccionamiento indirecto y datos.

Su disco deberá además soportar la adición de links, en específico **hard links**.

Bloque de directorio. Está compuesto por una secuencia de entradas de directorio, donde cada entrada de directorio ocupa 32 Bytes. Una entrada de directorio contiene:

- 2 Bits. Indica si la entrada es inválida (`0x0`), un archivo (`0x1`) u otro directorio (`0x2`).
- 22 Bits. Número de bloque donde se encuentra el bloque índice del archivo o el bloque directorio de la carpeta (*i.e.* su puntero). Como hay 1048576 bloques posibles, este rango puede ir desde `0x000000` hasta `0x0FFFFF`, o sea, desde 0 hasta 1048575.
- 29 Bytes. Nombre de archivo o carpeta, 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 29 Bytes, el resto de los Bytes deberán ser iguales a `0x00`.

El **primer bloque** del disco **siempre** corresponderá al directorio raíz de su sistema de archivos.

Bloque de bitmap. Los bloques de *bitmap* corresponden siempre a los bloques 1 al 64. Estos bloques son iguales en estructura y son los únicos de este tipo. El contenido de los bloques es el *bitmap* del disco. Cada Bit del *bitmap* indica si el bloque correspondiente en el disco está libre (0) o no (1). Por ejemplo, si el primer Bit del primer bloque de *bitmap* del disco tiene el valor 1, quiere decir que el primer bloque del disco está utilizado.

- El *bitmap* contiene 1 Bit por cada bloque del disco, 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 del disco 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:

- 1 Byte al inicio del bloque para la cantidad total de referencias (*hardlinks*) del archivo.
- 7 Bytes para el tamaño del archivo.
- 2036 Bytes, reservados para 509 punteros. Cada uno apunta a un bloque de direccionamiento indirecto simple.
- 4 Bytes, almacena un puntero a otro bloque índice.

Para el caso de un bloque índice que no sea el primero de su archivo, se usará espacio de metadata (cantidad de *hardlinks* y tamaño del archivo) para almacenar 2 punteros adicionales. En resumen, los bloques índice que no son el primer bloque del archivo contienen:

- 2044 Bytes, reservados para 511 punteros. Cada uno apunta a un bloque de direccionamiento indirecto simple.
- 4 Bytes, almacena un puntero a otro bloque índice.

Bloque de direccionamiento indirecto simple. Un bloque de direccionamiento indirecto simple utiliza todo su espacio para almacenar punteros a bloques 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)`. Función para montar el disco. Establece como variable global la ruta local donde se encuentra el archivo `.bin` correspondiente al disco.
- `void os_bitmap(unsigned num, bool hex)`. 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` (`num` $\in \{1, ..., 64\}$), ya sea en binario (si `hex` es `false`) o en hexadecimal (si `hex` es `true`). Si se ingresa `num` $= 0$, se debe imprimir **el bitmap completo**, imprimiendo además una línea con la cantidad de bloques ocupados, y una segunda línea con la cantidad de bloques libres.
- `int os_exists(char* path)`. Función para ver si un archivo existe. Retorna `1` si el archivo existe y `0` en caso contrario.
- `void os_ls(char* path)`. Función para listar los elementos de una carpeta. Imprime en pantalla los nombres de todos los archivos contenidos en la carpeta indicada por `filename`.

Funciones de manejo de archivos

- `osFile* os_open(char* path, char mode)`. Función para abrir un archivo. Si `mode` es `'r'`, busca el archivo en la ruta `path` y retorna un `osFile*` que lo representa. Si `mode` es `'w'`, se verifica que el archivo no exista en la ruta especificada 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`).
- `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* path)`. Función para borrar archivos. Elimina el archivo referenciado por la ruta `path` del directorio correspondiente. Los bloques que estaban siendo usados por el archivo deben quedar libres si y sólo si no existe ninguna otra referencia al archivo.

- `int os_hardlink(char* orig, char* dest)`. Función que se encarga de crear un **hardlink** del archivo referenciado por `orig` en una nueva ruta `dest`, aumentando la cantidad de referencias al archivo original.
- `int os_mkdir(char* path)`. Función que se encarga de crear un nuevo directorio con el nombre especificado.
- `int os_rmdir(char* path, bool recursive)`. Función que se encarga de eliminar un nuevo directorio con el nombre especificado, siempre y cuando este se encuentre vacío. Si el bool `recursive` es true, entonces elimina los contenidos de la carpeta recursivamente y luego la carpeta.
- `int os_unload(char* orig, char* dest)`. Función que se encarga de copiar un archivo o carpeta referenciado por `orig` hacia un nuevo archivo o directorio de ruta `dest` en su computador.
- `int os_load(char* orig)`. Función que se encarga de copiar un archivo o los contenidos de una carpeta, referenciado por `orig` al disco. En caso de que un archivo sea demasiado pesado para el disco, se debe escribir todo lo posible hasta acabar el espacio disponible. En caso de que el sea una carpeta, se deben copiar los archivos que estén dentro de esta carpeta, ignorando cualquier carpeta adicional que tenga.

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`) de 2 GB. 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"); // 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`

Estos discos se harán disponibles el día Lunes 5 de Octubre.

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**.
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. 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 y dar aviso de que ésta no fue realizada en su totalidad mediante un mensaje de error en `stderr`[1]. **No** debe eliminar el archivo que estaba siendo escrito.
- En el sistema **no existirán** dos archivos con el mismo path absoluto. Sin embargo, pueden existir archivos con el mismo nombre en carpetas distintas.
- Cualquier detalle **no especificado** en este enunciado puede ser abarcado mediante **supuestos**, los que deben ser indicados en el `README` de su entrega.

[1] Para más información con respecto al manejo de errores en C, ver el siguiente enlace.

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

- **0.80 pts.** Estructura de sistema de archivos[2].
  - **0.10 pts.** Representación de bloques directorio.
  - **0.20 pts.** Representación de bloque índice.
  - **0.20 pts.** Representación de bloques de direccionamiento indirecto.
  - **0.10 pts.** Representación de bloque de datos.
  - **0.20 pts.** Representación de bloque de *bitmap*.
- **4.2 pts.** Funciones de biblioteca.
  - **0.20 pts.** `os_mount`.
  - **0.20 pts.** `os_bitmap`.
  - **0.20 pts.** `os_exists`.
  - **0.20 pts.** `os_ls`.
  - **0.20 pts.** `os_open`.
  - **0.30 pts.** `os_read`.
  - **0.30 pts.** `os_write`.
  - **0.20 pts.** `os_close`.
  - **0.20 pts.** `os_hardlink`.
  - **0.20 pts.** `os_mkdir`.
  - **0.30 pts.** `os_rmdir`.
  - **0.30 pts.** `os_rm`.
  - **0.70 pts.** `os_unload`.
  - **0.70 pts.** `os_load`.
- **1.00 pts.** Manejo de memoria perfecto y sin errores (**Valgrind**).

[2] 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**.

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[3].

Política de atraso

Se puede hacer entrega de la tarea con un máximo de 4 días de atraso. La fórmula a seguir es la siguiente:

$$N_{P_1} = 1{,}0 + \sum_i p_i + b - d$$

$$N_{P_1}^{\text{Atraso}} = \text{mín}(N_{P_1}, 7{,}0 - 0{,}75 \cdot d + b - d)$$

Siendo N_{P_1} la nota obtenida, d el descuento total obtenido, p_i el puntaje obtenido del ítem i, b el puntaje asignado a través del bonus (ver siguiente sección), d la cantidad de días de atraso y $\text{mín}(x, y)$ la función que retorna el valor mas pequeño entre x e y.

Bonus (+1.0 pts): documentación y manejo de errores

Se aplicará este bonus a la nota final si elabora una documentación en formato `PDF` que contenga lo siguiente para **cada** función implementada:

- Descripción general de lo que realiza la función y sus argumentos. Si bien pueden usar de guía las descripciones y el formato utilizado en este enunciado, se espera que **profundicen** en su implementación propia para dejar claro el funcionamiento interno de su API.
- **Manejo de errores**. Como habrá notado, las descripciones de este enunciado son bastante amplias para muchas de las funciones solicitadas. Por ejemplo, ¿qué pasa si se trata de leer un archivo que no existe en el disco? ¿Qué pasa si se trata de escribir un archivo en una ruta que ya existe? Esto fue con un propósito: darles la posibilidad de **estandarizar** los retornos y mensajes impresos de las funciones en casos de error. Es decisión de ustedes, por ejemplo, ver si se retorna `NULL` o un código numérico según el error incurrido, además del contenido del mensaje impreso. Es importante mencionar que esto será considerado **solo si implementan el manejo descrito en el código, siendo evidenciado en la corrección presencial**.

Cabe destacar que el bonus será mayor en función del porcentaje de la API implementado: mientras más funciones de esta implementen y estandaricen según lo detallado anteriormente, mayor será el puntaje otorgado. Esperamos que esto sirva de motivación para que traten de realizar el proyecto en su totalidad.

Preguntas

Cualquier duda preguntar a través del **foro**.

[3] 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.