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

IIC2333 — Sistemas Operativos y Redes — 2/2019

Proyecto 1

Viernes 13-Septiembre-2019

Fecha de Entrega: Lunes 07-Octubre-2019 a las 14:00
Composición: grupos de n personas, donde $3 \leq n \leq 4$
Fecha de ayudantía: Viernes 13-Septiembre-2019 a las 8:00

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.

Cruz FileSystem

Luego de haber terminado el *Scheduler*, los ayudantes de Sistemas Operativos ya pueden comenzar a trabajar en las correcciones y ayudantías. Sin embargo, olvidaron una parte muy importante de su trabajo: poner notas. Debido a la gran cantidad de alumnos este semestre y las múltiples notas de tareas, actividades y proyectos, lamentablemente no pueden almacenar todas las notas en la RAM, por lo que necesitarán comprar un disco duro. Lamentablemente, debido a que **Germy** quería bajar costos, los ayudantes terminaron comprando un disco duro defectuoso, el que terminó borrando el sistema de archivos del servidor. Por este motivo, **Richi** te pide a ti y a tu grupo que implementen el sistema de archivos **ext2**, ojalá antes que **Cruz** los descubra y los despida.

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 `crfs`

El sistema de archivos a implementar será denominado `crfs`. Este sistema almacena archivos en bloques mediante la asignación indexada de dos niveles y permite la existencia de un directorio raíz y un conjunto de subdirectorios.

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

- Tamaño del disco: 1 GB.
- Tamaño de bloque: 1 KB. El disco contiene un total de $2^{20} = 1048576$ bloques.
- Cada bloque posee un número secuencial, almacenado en un `unsigned int` de 4 Byte (32 bit). **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.

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

- 1 Byte. Indica si la entrada es inválida (`0x01`), válida y correspondiente a un directorio (`0x02`), válida y correspondiente a un archivo (`0x04`), el mismo directorio "." (`0x08`), el directorio padre ".." (`0x10 = 16`) o el directorio de continuación (`0x20 = 32`), siendo este último detallado más adelante. Cualquier otro valor se toma como inválido.
- 27 Byte. Largo del nombre de archivo o subdirectorio, 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 27 Byte, el resto de los Byte deberán ser iguales a `0x00`.
- 4 Byte. Número de bloque donde se encuentra el bloque índice del archivo (*i.e.* puntero al archivo), o bien otro bloque directorio. Como hay 1048576 bloques posibles, este rango puede ir desde `0x00000000` hasta `0x000FFFFF`, o sea, desde 0 hasta 1048575.

Cada subdirectorio está representado por un bloque que contiene la estructura de un bloque directorio. El directorio raíz siempre estará ubicado en el primer bloque del disco, correspondiente al puntero `0x00000000`.

Por otra parte, **todo directorio** debe contener **una** entrada "." y **una** entrada "..". Estas son importantes dado que influirán en el resultado de algunas de las funciones de la API a implementar.

Por último, al final del bloque **se reservan** 32 Byte donde se pondrá una entrada a otro bloque de directorio, el que servirá como continuación en caso de que se quieran incluir más entradas de directorio correspondientes a archivos y/o subdirectorios. Al ser esta, en teoría, una entrada al mismo directorio, el primer Byte de identificación debe ser (`0x32`). Es importante notar que esto implicará que el directorio ocupará dos bloques para poder contener más entradas.

Bloque de bitmap. Un bloque de *bitmap* corresponde siempre a uno de los 128 bloques siguientes al bloque de directorio del disco, es decir, desde el bloque `0x00000001` al `0x00000080`. 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. De este modo, los primeros 129 bloques de disco, correspondientes al bloque directorio raíz más los 128 bloques *bitmap*, deben considerarse 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 el primer bloque de un archivo y contiene la información necesaria para acceder al contenido del archivo. Está compuesto por:

- 4 Bytes al inicio del bloque para el tamaño del archivo.
- 1008 Bytes, reservados para 252 punteros. Cada uno apunta a un bloque de datos.
- 4 Bytes, almacena un puntero de direccionamiento indirecto simple de ser necesario.
- 4 Bytes, almacena un puntero de direccionamiento indirecto doble de ser necesario.
- 4 Bytes, almacena un puntero de direccionamiento indirecto triple de ser necesario.

Cuando se escribe un archivo, se escribe en los bloques apuntados por los punteros directos. Una vez que estos se han llenado, se empieza a usar el bloque de direccionamiento indirecto.

Bloque de direccionamiento indirecto simple. Un bloque de direccionamiento indirecto simple utiliza todo su espacio para almacenar punteros a bloques de datos.

Bloque de direccionamiento indirecto doble. Un bloque de direccionamiento indirecto doble utiliza todo su espacio para almacenar punteros a bloques de direccionamiento indirecto simple.

Bloque de direccionamiento indirecto triple. Un bloque de direccionamiento indirecto triple utiliza todo su espacio para almacenar punteros a bloques de direccionamiento indirecto doble.

Bloque 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 sea parte del archivo) y **no puede ser subasignado** a otro archivo.

Es importante destacar que la lectura y escritura de Byte en los bloques de datos **deben** ser realizadas en orden ***big endian*** para mantener consistencia entre todos los sistemas de archivos implementados.

API de `crfs`

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 `cr_API.c` y su interfaz (declaración de prototipos) debe encontrarse en un archivo de nombre `cr_API.h`. Para probar su implementación debe escribir un archivo con una función `main` (por ejemplo, `main.c`) que incluya el *header* `cr_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 `cr_API.c` se debe definir un `struct` que almacene la información que considere necesaria para operar con el archivo. Ese `struct` debe ser nombrado `crFILE` mediante una instrucción `typedef`. Esta estructura representará un *archivo abierto*. Las funciones de la `API` usarán únicamente **rutas absolutas** de la siguiente manera:

Carpeta: `/folder_1/.../folder_n`
Archivo: `/folder_1/.../folder_n/filename`

La biblioteca debe implementar las siguientes funciones.

Funciones generales

- `void cr_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 cr_bitmap(unsigned block, 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 `block` (`bitmap_block` $\in \{1, ..., 129\}$), ya sea en binario (si `hex` es `false`) o en hexadecimal (si `hex` es `true`). Si se ingresa `block` $= 0$, 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 cr_exists(char* path)`. Función para ver si un archivo o carpeta existe en la ruta especificada por `path`. Retorna `1` si el archivo o carpeta existe y `0` en caso contrario.
- `void cr_ls(char* path)`. Función para listar los elementos de un directorio del disco. Imprime en pantalla los nombres de todos los archivos y directorios contenidos en el directorio indicado por `path`.
- `int cr_mkdir(char *foldername)` Función para crear directorios. Crea el directorio vacío referido por `foldername`.

Funciones de manejo de archivos

- `crFILE* cr_open(char* path, char mode)`. Función para abrir un archivo. Si `mode` es `'r'`, busca el archivo en la ruta `path` y retorna un `crFILE*` que lo representa. Si `mode` es `'w'`, se verifica que el archivo no exista en la ruta especificada y se retorna un nuevo `crFILE*` que lo representa.
- `int cr_read(crFILE* 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 Byte efectivamente leídos desde el archivo. Esto es importante si `nbytes` es mayor a la cantidad de Byte 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 cr_write(crFILE* 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 Byte 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 cr_close(crFILE* 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 cr_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.
- `int cr_unload(char* orig, char* dest)`. Función que se encarga de copiar un archivo o un árbol de directorios (es decir, un directorio y **todos** sus contenidos) del disco, referenciado por `orig`, hacia un nuevo archivo o directorio de ruta `dest` en su computador.
- `int cr_load(char* orig)`. Función que se encarga de copiar un archivo o árbol de directorios, 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.

Nota: Debe respetar los nombres y prototipos de las funciones descritas. Las funciones de la API poseen el prefijo `cr` 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 1 GB. El programa `main.c` deberá usar las funciones de la biblioteca `cr_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:

```
./crfs simdisk.bin
```

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

```
cr_mount("simdisk.bin"); // Se monta el disco.
file_desc = cr_open("test.txt", 'w');
// Suponga que abrió y leyó un archivo desde su computador,
// almacenando su contenido en un arreglo f, de 300 byte.
cr_write(file_desc, f, 300); // Escribe el archivo en el disco.
cr_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`

Corrección presencial

A diferencia de las tareas, este proyecto será corregido de **forma presencial**. Se hará uso del módulo de ayudantía siguiente a la fecha de entrega 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.

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.
- Dada la estructura de los bloques índice, sus archivos tendrán un tamaño máximo. Si está escribiendo un archivo y éste supera ese tamaño máximo, **no** debe eliminar el archivo, sino que debe dejar almacenado el máximo de datos posible y retornar el valor apropiado desde `cr_write`.

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

- En el sistema **no existirán** dos archivos o subdirectorios con el mismo nombre dentro de un mismo directorio.
- Cada bloque directorio puede tener, **como máximo**, un puntero a un bloque directorio de continuación. Este último, por ende, usará su espacio **exclusivamente** para entradas de archivos o subdirectorio. Notar que esto implica que también excluirá las entradas "." y "..".
- 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.

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 el repositorio de equipo creado, el proyecto **no se corregirá**.

Evaluación

- **1.10 pts.** Estructura de sistema de archivos[2].
  - **0.20 pts.** Representación de bloque directorio.
  - **0.10 pts.** Representación de bloque índice.
  - **0.20 pts.** Representación de bloques de direccionamiento indirecto simple.
  - **0.20 pts.** Representación de bloques de direccionamiento indirecto doble.
  - **0.20 pts.** Representación de bloques de direccionamiento indirecto triple.
  - **0.10 pts.** Representación de bloque de datos.
  - **0.10 pts.** Representación de bloque de *bitmap*.
- **4.6 pts.** Funciones de biblioteca.
  - **0.20 pts.** `cr_mount`.
  - **0.20 pts.** `cr_bitmap`.
  - **0.20 pts.** `cr_exists`.
  - **0.20 pts.** `cr_ls`.
  - **0.20 pts.** `cr_mkdir`.
  - **0.40 pts.** `cr_open`.
  - **0.40 pts.** `cr_read`.
  - **0.40 pts.** `cr_write`.
  - **0.40 pts.** `cr_close`.
  - **0.40 pts.** `cr_rm`.
  - **0.70 pts.** `cr_unload`.

[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**.

    - **0.70 pts.** `cr_load`.

- **0.50 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 2 puntos si se sube alguno de los archivos `simdisk.bin` al repositorio[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:

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

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

Siendo N_{P_1} la nota obtenida, 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 será de Gúgol puntos.