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

IIC2333 — Sistemas Operativos y Redes — 1/2018

Tarea 3

Jueves 10-Mayo-2018

Fecha de Entrega: Jueves 24-Mayo-2018 a las 23:59
Composición: grupos de n personas, donde $n \leq 2$

El objetivo de esta tarea es experimentar con la construcción y mantenimiento de un sistema de archivos simulado sobre un disco virtual.

Introducción

Los sistemas de archivos nos permiten organizar nuestros datos mediante la abstracción de archivo y almacenar estos archivos de manera ordenada en dispositivos de almacenamiento secundario. 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.

Estructura de sistema de archivos `czfs`

El sistema de archivos a implementar será denominado `czfs`. Este sistema almacena archivos en bloques mediante asignación indexada de dos niveles y permite la existencia de un único directorio (sin 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: 64 MB.
- Tamaño de bloque: 1 KB. El disco contiene un total de $2^{16} = 65536$ bloques.
- Cada bloque posee un número secuencial, almacenado en un `unsigned int` de 4 Byte (32 bit).

El disco puede almacenar cinco tipos de bloque: directorio, *bitmap*, índice, direccionamiento indirecto y datos.

Bloque de directorio. Un bloque de directorio corresponde siempre al bloque 0 del disco y es el único de este tipo. Está compuesto por una secuencia de entradas de directorio, donde cada entrada de directorio ocupa 16 Byte. Una entrada de directorio contiene:

- 1 Byte. Indica si la entrada es válida (`0x01`) o no (`0x00`).
- 11 Byte. Nombre de archivo, expresado usando caracteres ASCII (8-bit), donde el último Byte es `0x00`.
- 4 Byte. Número de bloque donde se encuentra el bloque índice del archivo (*i.e.* puntero al archivo). Como hay 65536 bloques posibles, este rango puede ir desde `0x00000000` hasta `0x0000FFFF`, o sea, desde 0 hasta 65535. Los 2 Byte más significativos no se usan.

Bloque de bitmap. Un bloque de *bitmap* corresponde siempre a cada uno de los 8 bloques siguientes al bloque de directorio del disco, es decir, de los bloques 1 al 8. Estos 8 bloques son iguales en estructura y únicos de este tipo. Cada bit del bloque indica si el bloque correspondiente en el disco está libre (0) o no (1).

Por ejemplo, si el primer bit del bloque 1 del disco tiene el valor 1, quiere decir que el primer bloque del disco está utilizado.

Tenga en cuenta:

- Se consideran todos los bloques del disco, sin importar su tipo, incluidos el de directorio, los de bitmap, los de índice y, por supuesto, los de datos. De este modo, los primeros 9 bloques, correspondientes al de directorio más los 8 de *bitmap*, deben considerarse como ocupados. Esto se traduce en que el valor de los primeros 4 Byte del bloque 1 para un disco vacío debe ser `0xFF800000`, y el resto del bloque contener 0's, al igual que los siguientes 7 bloques.
- Debe reflejar el estado del disco y se debe mantener actualizado.

Bloque índice. Un bloque índice es el primer bloque del archivo y contiene la información necesaria para acceder al contenido del archivo. Está compuesto por:

- 12 Byte al inicio del bloque para *metadata*:
  - 4 Byte. Tamaño del archivo.
  - 4 Byte. *Timestamp* de creación.
  - 4 Byte. *Timestamp* de última modificación.
- 1008 Byte. Espacio para 252 punteros. Cada uno apunta a un bloque de datos.
- 4 Byte al final del bloque, almacenan un puntero a un bloque de direccionamiento indirecto.

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 (apuntado por el último puntero).

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

Bloque de datos. Un bloque de datos utiliza todo su espacio para almacenar el contenido (datos) de un archivo. El formato de este bloque depende exclusivamente de lo que se quiere guardar en el archivo. Una vez que un bloque ha sido asignado a un archivo, se asigna de manera **completa**. Si el archivo requiere menos espacio que el tamaño del bloque, el espacio no utilizado por el archivo sigue siendo parte del bloque (si bien no es parte del archivo) y no puede ser *subasignado* a otro archivo.

API de `czfs`

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 `cz_API.c` y su interfaz (declaración de prototipos) debe encontrarse en un archivo de nombre `cz_API.h`. Para probar su implementación debe escribir un archivo con una función `main` (por ejemplo, `main.c`) que incluya el *header* `cz_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 `cz_API.c` se debe definir un `struct` que almacene la información que considere necesaria para operar con el archivo. Ese `struct` debe ser nombrado `czFILE` mediante una instrucción `typedef`. Esta estructura representará un *archivo abierto*. La biblioteca debe implementar las siguientes funciones:

- `czFILE* cz\_open(char* filename, char mode)`. Si `mode` es `'r'`, busca `filename` en el directorio y retorna un `czFILE*` que lo representa, en caso de existir el archivo; si el archivo no existe, se retorna `NULL`. Si `mode` es `'w'`, se verifica que el archivo no exista en el directorio y se retorna un nuevo `czFILE*` que lo representa. Si el archivo ya existía, se retorna `NULL`. Cualquier otro tipo de error también debe retornar `NULL`.

- `int cz_exists(char* filename)`. Retorna `1` si el archivo existe en el sistema de archivos y `0` en caso contrario.
- `int cz_read(czFILE* file_desc, void* buffer, int nbytes)`. 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`. Debe retornar `-1` si se produce un error (por ejemplo, si el archivo estaba abierto en modo `'w'`).
- `int cz_write(czFILE* file_desc, void* buffer, int nbytes)`. 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`). Si se produce otro tipo de error, debe retornar `-1`.
- `int cz_close(czFILE* file_desc)`. Cierra el archivo indicado por `file_desc`. Debe garantizar que cuando esta función retorna, el archivo se encuentra actualizado en disco. Retorna `0` si no hubo errores, o algo distinto de `0` en caso contrario.
- `int cz_mv(char* orig, char *dest)}`. Cambia el nombre del archivo `orig` a `dest`. Este comando se usa para mover archivos de carpeta, no obstante, en este contexto solo funciona para cambiar el nombre dado que no existen subdirectorios. Se debe cuidar que `dest` no sea un archivo ya existente en el directorio principal. Si no hay errores, retorna `0`; en caso contrario, retorna algo distinto a `0`.
- `int cz_cp(char* orig, char* dest)`. Copia el archivo de nombre `orig` en otro llamado `dest`. Se debe cuidar que los nombres sean distintos y que `dest` no sea un archivo ya existente en el directorio principal. Si no hay errores, retorna `0`; en caso contrario, retorna algo distinto a `0`.
- `int cz_rm(char* filename)`. Elimina el archivo de nombre `filename` del bloque de directorio. Si el archivo no existe, no se realizan cambios. Los bloques que estaban siendo usados por el archivo deben quedar libres.
- `void cz_ls()`. Escribe en pantalla los nombres de todos los archivos contenidos en el directorio principal.

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

Ejecución

Para probar su biblioteca, debe entrar un programa `main.c` que reciba un disco virtual (ej: `simdisk.bin`) de 64 MB. El programa `main.c` deberá montar (*i.e.* abrir) el disco virtual y usar las funciones de la biblioteca `cz_API.c` para ejecutar algunas instrucciones que demuestren el correcto funcionamiento de estas. 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:

```
./czfs simdisk.bin
```

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

```
file_desc = cz_open("test.txt", 'w');
// Suponga que abrió y leyó un archivo desde su computador,
```

```
// almacenando su contenido en un arreglo f, de 300 byte.
cz_write(file_desc, &f, 300);
cz_cp("test.txt", "copy.txt");
cz_close(file_desc);
```

Al terminar de ejecutar todas las instrucciones, debe escribir el nuevo `simdisk.bin` con los cambios aplicados. Para ello, puede ejecutarlas todas 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 este para ir guardando los cambios realizados de forma inmediata. Lo importante es que el estado final del disco virtual sea consistente con la secuencia de instrucciones ejecutada.

Dentro de la ejecución, es importante que tenga las siguientes consideraciones:

- Los bloques de datos de un archivo no están almacenados, necesariamente, 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.
- No es posible, mediante la biblioteca, borrar el bloque de directorio.
- Si se escribe un archivo y ya no queda espacio disponible en el disco virtual, debe terminar la escritura y dar aviso de que esta no fue realizada en su totalidad mediante un mensajes 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 este supera ese tamaño máximo, **no** debe eliminar el archivo, sino que debe dejar almacenado el máximo de datos posible de este y retornar el valor apropiado desde `cz_write`.
- Los errores que puedan generar las funciones de la biblioteca (por ejemplo, la lectura de un archivo inexistente) debe ser reportada al usuario a través de mensajes impresos en `stderr`. Debe considerar todos los casos explicitados en cada función de la sección "API de `czfs`".
- Para escribir en `stderr` puede usar `fprintf(stderr,...)`.

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/T3/simdiskformat.bin`

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

  `/home/iic2333/T3/simdiskfilled.bin`

Observaciones

- Existirá una única carpeta equivalente a la raíz del sistema de archivos, por lo que no habrá otro bloque que represente un directorio dentro del disco virtual.
- En el sistema **no existirán** dos archivos con el mismo nombre.

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

README y Formalidades

Deberá incluir un archivo `README` que indique quiénes son los autores de la tarea (**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 necesarias para facilitar la corrección de su tarea. Se sugiere utilizar formato ***markdown***.

A cada alumno se le asignó un nombre de usuario y una contraseña para el servidor del curso (`iic2333.ing.puc.cl`). Para entregar su tarea usted deberá crear una carpeta llamada `T3` en su carpeta personal y subir su tarea a esta. En su carpeta `T3` **solo debe incluir código fuente** necesario para compilar su tarea, además del README y un `Makefile`. **NO debe incluir archivos binarios (será penalizado)**. Se revisará el contenido de dicha carpeta el día Jueves 24-Mayo-2018 a las 23:59.

Además, debe considerar los siguientes puntos:

- Puede ser realizada en forma individual, o en grupos de dos personas. En cualquier caso, recuerde indicar en el `README` los autores de la tarea con sus respectivos números de alumno.
- En caso de ser realizada por dos personas, **solo uno de los integrantes** debe subir su desarrollo a la carpeta correspondiente en el servidor.
- 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. Se recomienda modularizar, utilizar funciones y ocupar nombres de variables explicativos. En el caso de no entregar en la carpeta especificada la tarea **no se corregirá**.

Evaluación

- $0{,}25$ pts. Formalidades. Esto incluye cumplir las normas de la sección de formalidades
- $1{,}60$ pts. Estructura de sistema de archivos
  - $0{,}4$ pts. Manejo de bloque directorio
  - $0{,}4$ pts. Manejo de bloque índice
  - $0{,}4$ pts. Manejo de bloque de direccionamiento indirecto
  - $0{,}4$ pts. Manejo de bloque de datos
- $3{,}40$ pts. Funciones de biblioteca.
  - $0{,}4$ pts. `cz_open`
  - $0{,}4$ pts. `cz_close`
  - $0{,}6$ pts. `cz_read`
  - $0{,}6$ pts. `cz_write`
  - $0{,}4$ pts. `cz_rm`
  - $0{,}4$ pts. `cz_cp`
  - $0{,}2$ pts. `cz_mv`
  - $0{,}2$ pts. `cz_exists`
  - $0{,}2$ pts. `cz_ls`
- $0{,}75$ pts. Programa de prueba `main.c`

Bonus (+0.5 pts): manejo de memoria perfecto

Se aplicará este bonus si *valgrind* reporta en su código 0 *leaks* y 0 errores de memoria, considerando que los programas funcionen correctamente. El bonus a su nota se aplica solo si la nota correspondiente es $\geq 3{,}95$.

Preguntas

Cualquier duda preguntar a través del foro.