

AYUDANTÍA P1

Germán Leandro Contreras Sagredo

Ricardo Esteban Schilling Broussaingaray

IIC2333 [2019-1] - Sistemas Operativos y Redes

INTRODUCCIÓN

Los objetivos de esta ayudantía son:

1. Comprender la estructura de nuestro sistema de archivos.
2. Aclarar el funcionamiento esperado de la API `crfs`.
3. Resolver las dudas que surjan durante la explicación de lo pedido.

SISTEMA DE ARCHIVOS

- El disco es de tamaño 128MB con bloques de 2KB cada uno.
- Este tiene 65536 bloques, ordenados de manera secuencial.
- Un puntero a un bloque no es más que un `int` de 4 bytes, correspondiente al número de bloque.
- Cuando un bloque es asignado a una función específica, este se asigna **completamente**.
- El primer bloque **siempre** será nuestro directorio **root**.

El bloque de directorio corresponde a un directorio en nuestro sistema de archivos.

Cada entrada del directorio, es decir, cada archivo o subdirectorio dentro de este está representado como una secuencia de 32 bytes.

- Un byte para indicar si la entrada es inválida (0x01), válida y correspondiente a directorio (0x02) o válida y correspondiente a archivo (0x04). Cualquier otro valor del primer byte también representará una entrada inválida, no obstante, no se debería dar.
- 27 bytes que representan el nombre de la entrada en **ASCII**, cuando un byte no esté siendo ocupado, este debe ser seteado a **0x00**.
- 4 bytes que representan el puntero al **bloque índice** del archivo.

- Corresponden a los siguientes cuatro bloques del disco.
- Su función es indicar los bloques que están ocupados y los que están libres.
- Habrá un bit igual a 1 si el bloque correspondiente está ocupado y 0 en caso contrario.

Ejemplo

El **byte** número 123 (contando desde 0) del primer bloque de directorio es **0xA3** (10100011_2), por lo que inmediatamente sabemos que los bloques 984, 986, 990 y 991 están ocupados.

- Los primeros 5 bloques del disco **siempre** estarán ocupados.
- Los bloques de bitmap **siempre** deben reflejar el estado actual del disco.

- Es el primer bloque de un archivo.
- Posee 4 bytes para el tamaño del archivo.
- Posee 4 bytes para contar la cantidad de referencias al archivo (hardlinks)
- También tiene espacio para 500 punteros distintos, cada uno de 4 bytes, estos apuntan **exclusivamente** a bloques de datos.
- Al final de estos bloques, se reservan 40 bytes para tener hasta diez punteros a bloques de direccionamiento indirecto, de ser necesario.
- El orden de los bloques en el bloque índice dicta el orden de los bloques de datos del archivo.

- Similar al bloque índice, solo que **no** posee bytes para metadata o puntero final, por lo que este bloque tiene espacio para guardar 512 punteros a bloques de datos.
- Al existir hasta once punteros a bloques indirectos, el **tamaño máximo** de un archivo es de $(500 + 512 * 10) * 2 = 11240\text{KB}$

- Utiliza la totalidad de su espacio para guardar los archivos.
- No pueden ser subasignados, es decir, cuando uno asigna un bloque de datos a un archivo, este se asigna en su totalidad y no pueden haber dos o más archivos compartiendo el mismo bloque.

CRFS API

- Cristian Ruz File System.
- La API debe estar implementada en un archivo `cr_API.c` con la interfaz llamada `cr_API.h`.
- Debe funcionar a partir de un programa `main.c` que utilice las funciones de su librería.
- Son libres de subir sus propios archivos para agregarlos al disco.
- Dentro de su API, deben definir un `struct` llamado `crFILE`, el que representa un archivo abierto. Este es similar al `struct FILE` de la librería `stdio.h`. Son libres en cuanto a los datos que posee esta estructura.
- La API utiliza únicamente rutas `absolutas`.

- `void cr_mount(char* diskname)` Esta función se encarga de **montar** el disco, dejando como **variable global** la ruta al archivo binario correspondiente. **Siempre** es la primera función que corre en su archivo **main**.
- `void cr_bitmap()` Imprime el bitmap del disco previamente montado, la cantidad de bloques ocupados y la cantidad de bloques libres.

- `int cr_exists(char* path)` Retorna 1 si `path` existe en el disco y 0 si no.
- `void cr_ls(char* path)` Similar al comando `ls` de Unix, imprime los contenidos de `path`.
- `int cr_mkdir(char *foldername)` Crea un directorio vacío en la ruta indicada. Debe preocuparse que la ruta sea válida.

- `crFILE* cr_open(char* path, char mode)` Abre un archivo y retorna un puntero a la instancia de `crFILE` que lo representa. El modo puede ser `'r'` para leer archivos existentes o `'w'` para escribir nuevos archivos.
- `int cr_read(crFILE* file_desc, void* buffer, int nbytes)` Lee los **siguientes** `nbytes` del archivo descrito por `file_desc` y lo guarda en un `buffer`. La función debe retornar la cantidad de bytes leídos.
- `int cr_write(crFILE* file_desc, void* buffer, int nbytes)` Escribe los `nbytes` que se encuentren en el `buffer` al archivo descrito por `file_desc`.

- `int cr_close(crFILE* file_desc)` Función que cierra un archivo abierto previamente. Cuando un archivo es cerrado, este debe estar actualizado en el disco.
- `int cr_rm(char* path)` Función que elimina una referencia a un archivo. Es muy importante que se liberen **todos** los bloques ocupados por el archivo si es que no existen mas referencias a este.

- `int cr_hardlink(char* orig, char* dest)` Crea un **hardlink** del archivo de **orig** hacia un nuevo archivo **dest**, además de aumentar la cantidad de referencias del archivo original.
- `int cr_load(char* orig)` Toma una carpeta o archivo del computador y la carga dentro del disco, con los mismos nombres de archivos y estructura.
- `int cr_unload(char* orig, char *dest)` Contrario a la función **load**, esta función recibe la ruta de un archivo o carpeta del disco y lo/la guarda en la ruta **dest** del computador.

FIN
