Nombre: GRUPO001

Christian Ruiz Filesystem SysCalls

Todas las funciones que retornan un int, retornan cero al fallar y uno al ejecutarse correctamente.
Todas las funciones cierran y abren el disco y liberan la memoria de las variables creadas con malloc.

Syscall	Description
int move_index(char * path, crFILE* p);	Esta función se utiliza para viajar a través del disco. Es fundamental para el desarrollo de la API. Separa el path entregado según el carácter "/" y luego revisa comenzando en el root, que se encuentre cada sub string separado por "/" en el bloque, y luego viaja hasta donde apunta la entrada correspondiente y repite la comparación hasta que ya no quedan más strings de la separación por "/". Igual que iterar sobre un split de python. En cada paso guarda el número de bloque y número de entrada en el directorio al que apunta en la struct crFILE p.
char* dirfinder(char* path);	Función que toma un argumento path y lo separa en el sub-path que lleva hasta el directorio o archivo apuntado por path.
char* basefinder(char* path);	Función que toma un argumento path y entrega el nombre del directorio o archivo apuntado por path.
blockIndex* find_empty_block();	Función que retorna una estructura blockIndex y escribe en el bitmap donde encuentra el primer bit que represente a un bloque desocupado y lo ocupa con un 1. Retorna una struct con la info necesaria para encontrar el bloque apuntado.
void change_bitmap(blockIndex* block);	Función auxiliar que escribe un byte en el bitmap del disco obtenido de un blockIndex. (Posiblemente proveniente de una llamada a find_empty_block() según se diseñe).
void cr_mount(char* diskname);	Función que escribe en una variable global el path al disco binario.
void cr_bitmap();	Se abre el archivo y se comienza a leer desde el primer bloque de bitmap (2048 bytes dentro del disco) y para cada byte del bitmap (2048 * 4 en total) se revisan sus bits, contando cada cero y cada uno. Se imprimen en pantalla todos los bits y los contadores finales.
int cr_exists(char* path);	Esta función llama a move_index(path, crFILE), una función auxiliar que accede al path y guarda en la struct crFILE el numero de bloque donde se encuentra el archivo o directorio apuntado por path. Además de retornar un int uno si el archivo/directorio

	fue encontrado en el disco y un cero en caso contrario. cr_exists retorna este mismo int.
void cr_ls(char* path);	Función que lee cada entrada del bloque apuntado por path y las imprime en pantalla, clasificándolas como FILE o DIR.
int cr_mkdir(char *foldername) ;	Separamos el nombre del directorio del el path hacia él. En caso de que ya exista un directorio con el nombre entregado en ../ , se retorna un cero. En caso contrario se utiliza find_empty_block para encontrar un bloque libre en el bitmap. Luego se crea en un buffer de tamaño 32 bytes un byte de directorio (un 4) luego 27 bytes correspondientes al nombre base obtenido por basefinder() y luego se viaja al bloque donde esta el archivo para obtener su numero de bloque y el puntero hacia este. Este puntero se escribe en los ultimos 4 bytes del buffer. Finalmente el fibber se escribe en la entrada correspondiente a obtenida de dirfinder().
crFILE* cr_open(char* path, char mode);	Se checkea la existencia de un archivo apuntado por path utilizando move_index. En caso de que ya exista se retorna una estructura crFILE con su atributo exists seteado en 1 y apuntando al bloque cero. En caso contrario si el modo es “r” se retorna una struct crFILE con la información del archivo obtenido por move_index. En caso de que mode ser “w, se escribe un archivo escribiendo con un buffer de 32 bytes 64 veces por bloque hasta completar los 2048 bytes y cambiando el bitmap con find_empty_block().
int cr_read(crFILE* file_desc, void* buffer, int nbytes);	Se abre el archivo binario disc_path, para luego encontrar el bloque de índice de archivo de puntero de file_desc. Esto ya está guardado en la estructura de el crFILE como un atributo llamado bloque. El bloque de índice consiste de cuatro bytes que representan el tamaño del archivo, cuatro bytes que representan el número de hard-links, 2000 bytes para los 500 punteros a bloques de datos y 40 bytes para 10 punteros indirectas. Usamos el bloque índice para extraer el tamaño del archivo y buscar a los punteros a los bloques de datos donde se encuentra el contenido del archivo. Luego utilizamos el tamaño del archivo y la variable nbytes, que es una de las variables del input para decidir cuántos bytes vamos a leer. Si los nbytes, que corresponden a la cantidad de bytes que el usuario nos pidió a leer) son más grandes que el tamaño del archivo, solo podremos leer el número de bytes del tamaño del archivo. Así tambíen sabemos cuántos bloques vamos a leer. Para todos los bloques que queremos leer, usamos el

	puntero al bloque y leemos los datos. Al final devolvemos el número de bytes leídos, ya sea de tamaño del archivo o nbytes como se explica.
int cr_write(crFILE* file_desc, void* buffer, int nbytes);	Igual a el cr_read se abre el archivo binario disk_path. El bloque de índice del archivo nos da información sobre el tamaño del archivo y la ubicación de los bloques de datos que contienen el contenido del archivo. Se encuentra el número de bytes para leer, y se lee el nbytes o el tamãno del archivo. Los datos se escriben desde el buffer (void * buffer) hasta el archivo file_desc (crFile * file_desc) en lotes de 2048 bytes. Se devuelve el número de bytes escrito en el archivo. Si el número deseado de bytes para escribir es mayor que 1000 kB (500 bloques de datos x 2048 bytes en un bloque) o el disco está lleno, cr_FILE devolverá 0.
int cr_close(crFILE* file_desc);	Se libera la memoria tomada.
int cr_rm(char* path);	1. Invalidar la entrada del bloque de directorio donde se encuentra el archivo: Para esto buscamos el bloque de directorio en donde se encuentra el puntero del archivo a eliminar y cambiamos el primer byte de la entrada a 0x01. 2. Disminuir en uno el contador de hardlinks del archivo: Para esto buscamos el bloque índice del archivo, nos saltamos los primeros 44 bytes y luego leemos los siguientes 4 bytes. Disminuimos en 1 el valor encontrado y escribimos en el disco el nuevo valor. 3. Si no quedan más hardlinks (contador = 0) invalidar bloques de espacio en el bitmap: Si el valor actualizado es igual a cero, proseguimos a invalidar los bloques. a. Invalidar el bloque índice del archivo: cambiamos a 0 el bit del bitmap que representa el bloque. b. Invalidar los bloques de datos: Leemos en un buffer los 2000 byte de punteros, luego vamos revisando al bloque que corresponde cada puntero válido (de los 500 punteros) e invalidamos el bloque en el bitmap. c. Invalidar los bloques de direccionamiento indirecto : Por cada puntero que hay en el direccionamiento indirecto hay que ir al bloque, leer todos los punteros y poner inválido en el bitmap el bloque de datos, luego poner

Nombre: GRUPO001

	inválido el bloque de direccionamiento en el bitmap.
int cr_hardlink(char* orig, char* dest);	Primero se checkea con move_index() que no exista otro hardlink en con el nombre dest en el directorio donde se quiere escribir el hardlink. Si ya existe, se retorna cero. En caso contrario se checkea que el directorio donde escribir el hardlink sea root, de ser root se corrige dirfinder con un string vacío en vez de un punto ".". Luego se apunta al directorio contenedor del hardlink nuevo para ver si quedan entradas disponibles en el bloque. Al encontrar una entrada se escribe una entrada de archivo con un buffer de 32 bytes y los ultimos 4 bytes como punteros al archivo original. En caso contrario se retorna un cero. Para conseguir el byte puntero al archivo original se utiliza move_index, si existe el archivo se obtiene el puntero, en caso contrario se retorna cero y se imprime en pantalla el error. Finalmente se escribe se libera la memoria tomada y se cierran los archivos.
int cr_unload(char* orig, char* dest);	Primero el archivo binario "disk_path" está abierto. Si la base de la input orig es un archivo, el archivo se lee del disco utilizando el tamaño del archivo y los punteros al bloque de datos. Luego, el contenido del archivo se escribe a el input destino (dest). De lo contrario, iteramos a través de los bloques en el directorio. Si los bloques contienen un archivo, el archivo se escribe en el destino de la misma manera como ya hemos explicado. Si el bloqueo es un otro directorio, se vuelve a llamar a unload, pero orig y dest tiene el nombre del directorio como base.
int cr_load(char* orig)	Esta función toma un path a un archivo en la computador. El archivo se abre y se lee a un buffer. Luego, se crea un crFILE usando cr_open(). Por último, el contenido del archivo de la computadora se escribe a el nuevo crF