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

IIC2333 — Sistemas Operativos y Redes — 2/2023

Proyecto 0

Viernes 15 de Septiembre, 2023

Fecha de Entrega: Viernes 29 de Septiembre 2023
Preguntas: foro oficial.

Objetivos

- Conocer la estructura de un sistema de archivos y sus componentes.
- Implementar una API para manejar el contenido de un disco virtual con particiones a partir de su sistema de archivos.

OldSchool FileSystem

Luego de intentar corregir la Tarea 0 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. Para este proyecto se recomienda ver las cápsulas de sistemas de archivos y asignación indexada y multinivel.

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 asignación indexada.

El disco virtual es un archivo en el sistema de archivos real, está dividido en **bloques** y sus características generales son las siguientes:

- Tamaño del disco: 2 GB divididos en bloques.
- Tamaño de bloque: 1 KB. El disco contiene un total de $2^{21} = 2097152$ bloques, identificados desde el 0 hasta el 2097151. Los identificadores o números de bloques serán almacenados dentro del disco virtual en porciones de 4 Bytes, esto quiere decir que un identificador se puede guardar en un `unsigned int`.

Tipos de Bloques

Cada bloque en el disco pertenece a uno de cinco tipos de bloque: *bitmap*, directorio, índice, direccionamiento indirecto simple y datos. A continuación, se decribe cada uno de estos:

Bloque de bitmap. Los bloques de *bitmap* corresponden siempre a los primeros 256 bloques 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* de tiene el valor 1, quiere decir que el primer bloque está siendo utilizado.

- El *bitmap* contiene 1 bit por cada bloque de la partición, sin importar su tipo.
- El *bitmap* debe reflejar el estado de la partición y se debe mantener actualizado respecto al estado de los bloques.
- El bloque directorio root y los bloques de bitmap siempre deben considerarse siempre como ocupados, es decir, los primeros 257 bits deben ser 1 en cualquier estado del disco.

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

- 1 Byte. Indica si la entrada es inválida (`0x00`), válida y correspondiente a un directorio (`0x01`), o válida y correspondiente a un archivo (`0x02`).
- 27 Byte. 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. El último byte del nombre no necesariamente es igual 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.

El **bloque 256 siempre** corresponderá al directorio raíz de su sistema de archivos, es decir, siempre es un bloque de directorio. Además, cada subdirectorio está representado por un bloque que contiene la estructura de un bloque directorio.

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:

- 4 Bytes para el tamaño del archivo.
- 1020 Bytes, reservados para 255 identificadores de bloques direccionamiento indirecto simple.

Bloque de direccionamiento indirecto simple. Un bloque de direccionamiento indirecto simple utiliza todo su espacio para almacenar indentificadores a bloques de datos, es decir, cuenta con 256 identificadores.

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**y su información será y debe ser escrita desde el primer byte. 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.

Es importante destacar que la lectura y escritura de datos **deben** ser realizadas en orden **little endian**. Consideren que el *endianess* solo afecta a los tipos de datos que tengan un tamaño mayor a un Byte, por lo que para este proyecto deben tenerlo en cuenta para los valores numéricos (direcciones, tamaños, etc). Además, si utilizan los tipos de datos correctos no deberían *swapear* los Bytes, ya que en general los computadores trabajan con ***little endian***.

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. Las funciones de la `API` usarán únicamente **rutas absolutas** de la siguiente manera:

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

osFile

Dentro de `os_API.h` 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* y como mínimo debe presentar los siguientes campos:

- Nombre del archivo.
- El tamaño del archivo.
- Identificador del bloque índice del archivo.

Funciones

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, de tal manera que esta ruta pueda ser usada por otras funciones.
- `void os_tree()`. Función que lista el árbol de directorios y archivos completo del disco, a partir del directorio raíz. Debe imprimir también para cada directorio su número de bloque, y para cada archivo, su número de bloque índice y su tamaño. Se espera que la indentación indique el nivel dentro del árbol. A continuación, se presenta un ejemplo para que se guíen en la implementación de la función.

```
mi_directorio_raiz - n_bloque
    carpeta1 - n_bloque
        archivo1.txt - n_bloque - size_archivo
        archivo2.txt - n_bloque - size_archivo
        subcarpeta1 - n_bloque
            archivo3.txt - n_bloque - size_archivo
    carpeta2 - n_bloque
        archivo4.txt - n_bloque - size_archivo
```

En donde, `n_bloque` corresponde al identificador del bloque y el `size_archivo` corresponde al tamaño del archivo en Byte.

- `void os_bitmap(~~unsigned~~ int num)`. Función para imprimir el *bitmap*. Cada vez que se llama esta función, imprime en binario el consola el estado del bloque bitmap `num`. Si se ingresa $\texttt{num} = -1$, se debe imprimir **el bitmap completo**. Se debe imprimir además una línea con la cantidad de bloques ocupados, y una segunda línea con la cantidad de bloques libres de todo el bitmap.
- `void os_ls(char* path)`. Función para listar los elementos de directorio indicado por `path`. Imprime en pantalla los nombres de todos las entradas válidas del bloque directorio.
- `int os_exists(char* path)`. Función para ver si un archivo o directorio existe. Retorna `1` si existe y `0` en caso contrario. Nótese que se deben comprobar la existencia de todos los directorios previos al archivo o directorio buscado.
- `int os_mkdir(char *foldername, char* path)`. Función para crear directorios. Crea el directorio vacío referido por `foldername` dentro del directorio indicado por `path`. El bloque directorio asignado será siempre el primer bloque vacío del disco virtual. **La función retorna 0 si el directorio fue creado, y 1 en caso contrario.**
- `void os_unmount()`. Función que permite desmontar el disco virtual del sistema. Si durante la ejecución se asignó memoria en el heap, esta función se encargará de liberarla adecuadamente para evitar fugas de memoria.

Funciones de Manejo de Archivos

- `osFile* os_open(char* path, char mode)`. Función para abrir un archivo. Si `mode` es `'r'`, busca el archivo inidicado por `path` y retorna un `osFile*` que lo representa. Si `mode` es `'w'`, se deben seguir lo siguientes pasos:

  1. Se verifica que el archivo no exista.
  2. Se le asigna el primer bloque vacío del disco como bloque índice.
  3. Crea una entrada válida en el bloque directorio correspondiente.
  4. Se retorna un nuevo `osFile*` que lo representa.

  En cualquier otro caso, retorna un puntero `NULL`.
- `void os_read(osFILE* file_desc, char* dest)`. Función para descargar un archivo del disco. Lee un archivo desde descrito por `file_desc` y crea una copia del archivo en la ruta indicada por `dest` dentro de su computador.
- `int os_write(osFile* file_desc, char* path)`. Función para escribir un archivo en el disco. Toma el archivo ubicado en la ruta `src` de su computador, y lo escribe dentro del disco montado. Toda escritura debe comenzar desde el primer bloque vacío en el disco virtual, y es importante que cualquier cambio producido debe verse reflejado tanto en el disco montado, como en `file_desc`. Finalmente, esta función retorna la cantidad de Bytes escritos.
- `void 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.
- `void os_rm(char* path)`. Función para borrar archivos. Elimina el archivo referenciado por la ruta `path` del directorio correspondiente. Nótese que eliminar un archivo debe producir cambios en el bitmap y en la entrada correspondiente en su directorio.
- `void os_rmdir(char* path)`. Función para borrar un directorio. Elimina el directorio referenciado por la ruta `path` y todo su contenido. Al igual que la función anterior, eliminar un directorio debe producir cambios en el bitmap y en la entrada correspondiente en su directorio.

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.

Bonus

En este proyecto habrá un bonus de 10 décimas que consiste en implementar la función `void cleanup()`. Esta función sirve para limpiar o liberar bloques "muertos" o basura en un sistema de archivos. Para ello, tienen que buscar los bloques que en el bitmap aparecen como *utilizados* (o sea que el bit correspondiente al bloque es 1 en el bitmap), pero que, al mismo tiempo, no estén siendo referenciado por ningún otro bloque, y marcarlos como bloques libres.

Se deben cumplir, al menos parcialmente, con todos estos puntos. La cantidad de décimas de bonus que se obtenga depende de:

- Proyectos con nota mayor o igual a $4{,}0$ pueden obtener hasta 5 décimas de bonus.
- Proyectos a nota mayor o igual a $5{,}0$ pueden obtener hasta 10 décimas de bonus.

Ejecución

Para probar su biblioteca, debe usar un programa `main.c` que reciba un disco virtual (ej: `simdisk.bin`). 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(argv[1]); // Se monta el disco.
osFile* file_desc = os_open("test.txt", 'w');
os_write(file_desc, f); // Escribe el archivo en el disco.
os_close(file_desc);  // Cierra el archivo. Ahora debería estar actualizado en el disco.
os_unmount();  // Libera memoria
```

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.zip`: Archivo comprimido de 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/P0/simdiskformat.zip`

- `simdiskfilled.zip`: Archivo comprimido de disco virtual con archivos escritos en él. Se podrá descargar del servidor a través de la siguiente ruta:

  `/home/iic2333/P0/simdiskfilled.zip`

Supuestos

- La primera función a utilizar siempre será la que monta el disco.
- La última función a utilizar siempre será la que monta el disco.
- Para efectos de la corrección, los directorios o archivos indicados por un `path` entregados a una función siempre existirán, a excepción de la función `os_exists`.
- El nombre de los archivos o directorios es único.

Observaciones

- La lectura y escritura de archivos debe realizarse desde la primera entrada disponible en el bloque de índice y la primera entrada disponible en el bloque de indirección simple.
- 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.
- Siempre que se requiera escribir sobre un nuevo bloque, de cualquier tipo, se tomará el primer bloque vacío. En caso de tener que escribir varios tipos de bloques nuevos, debe seguir la jerarquía *directorio* $>$ *índice* $>$ *direccionamiento* $>$ *datos*.
- 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. Basta con marcarlos como *libres* en el bitmap.
- Si se está escribiendo un archivo y ya no queda espacio disponible en el disco, debe terminar la escritura. **No** debe eliminar el archivo que estaba siendo escrito.
- Cualquier detalle **no especificado** en este enunciado puede ser abarcado mediante **supuestos**, los que deben ser indicados en el `README` de su entrega.

Formalidades

A cada alumno se le asignó un nombre de usuario y una contraseña para el servidor del curso[1]. Para entregar su tarea usted deberá crear una carpeta llamada `P0` en el directorio principal de su carpeta personal y subir su tarea a esa carpeta. En esta carpeta **solo debe incluir el código fuente** necesario para compilar su tarea y un `Makefile`. Se revisará el contenido de dicha carpeta el día Viernes 29 de Septiembre 2023.
Solo uno de los integrantes de cada grupo debe entregar en su carpeta. Los grupos los elegirán ustedes y en caso de que alguien no tenga compañero, puede crear una discusión en el foro del curso buscando uno.
Antes de acabado el plazo de entrega de la tarea, se enviará un form donde podrán registrar los grupos junto con el nombre de usuario de la persona que realizará la entrega en el servidor.

- **NO debe subir su tarea a un repositorio público**. En caso contrario, tendrá como nota máxima 4.0.
- **NO debe incluir archivos binarios**. En caso contrario, tendrá un descuento de 0.3 puntos en su nota final.
- Si inscribe de forma incorrecta su grupo, tendrá un descuento de 0.3 puntos.
- Su tarea deberá compilar utilizando el comando `make` en la carpeta `P0`, y generar un ejecutable llamado `osfs` en esta misma. Si su programa **no tiene** un `Makefile` o el nombre de la carpeta no es el correcto, **no se corregirá** y tendrá un descuento de 0.5 puntos en la recorrección.

[1] `iic2333.ing.puc.cl`

- Si usan el *plugin* de SSH para VSCode o uno similar, que instale software en el servidor, tendrán un descuento de 0.5 puntos.
- Si su programa **no compila** o **no funciona** (*segmentation fault*), obtendrán la nota mínima, pudiendo recorregir modificando líneas de código con un descuento de una décima por cada cuatro líneas modificada, con un máximo de 20 líneas a modificar.
- Descuento de 3 puntos si se sube alguno de los archivos `simdisk.bin` al servidor[2].

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.

Evaluación

- **5.00 pts.** Funciones de biblioteca.
  - **0.20 pts.** `os_mount`.
  - **0.40 pts.** `os_bitmap`.
  - **0.50 pts.** `os_exists`.
  - **0.30 pts.** `os_ls`.
  - **0.40 pts.** `os_tree`.
  - **0.40 pts.** `os_mkdir`.
  - **0.30 pts.** `os_open`.
  - **0.70 pts.** `os_read`.
  - **0.80 pts.** `os_write`.
  - **0.20 pts.** `os_close`.
  - **0.50 pts.** `os_rm`.
  - **0.30 pts.** `os_rmdir`.
- **1.00 pts.** Manejo de memoria perfecto y sin errores (**Valgrind**). Se obtiene este puntaje si `valgrind` reporta en su código **0** ***leaks*** y **0 errores** de memoria en **todo caso de uso**. Si el código entregado no muestra una intención clara de realizar el proyecto, este puntaje no será asignado.
- **1.00 pts.** Bonus `os_cleanup`. No hay puntaje parcial.

Política de Integridad Académica y Código de Honor de la UC

Los alumnos de la Escuela de Ingeniería de la Pontificia Universidad Católica de Chile deben mantener un comportamiento acorde al Código de Honor de la Universidad:

"Como miembro de la comunidad de la Pontificia Universidad Católica de Chile me comprometo a respetar los principios y normativas que la rigen. Asimismo, prometo actuar con rectitud y honestidad en las relaciones con los demás integrantes de la comunidad y en la realización de todo trabajo, particularmente en aquellas actividades vinculadas a la docencia, el aprendizaje y la creación, difusión y transferencia del conocimiento. Además, velaré por la integridad de las personas y cuidaré los bienes de la Universidad."

[2] 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.