Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2333 – Sistemas Operativos y Redes (I/2022)

Proyecto 1

Fecha de entrega: Miércoles 4 de Mayo a las 21:00
Composición: grupos de n personas, con $n = 5$

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.

SuperSpeedDrive FileSystem

Luego de años de esperar que el servidor del curso se encendiera, el cuerpo de ayudantes del curso, ha decidido mejorar su antiguo disco duro, por un moderno disco en estado sólido. Pero antes de esto, es necesario diseñar un sistema de archivos que sea compatible con esta nueva tecnología, por lo que les han pedido que implementen un nuevo sistema de archivos para poder tener más tiempo de ~~jugar~~ corregir actividades.

Introducción

Uno de los roles que cumple un sistema operativo, es el de acceder y almacenar nuestros datos de forma ordenada, y a través de la abstracción de archivos y directorios. Estos archivos se guardan en dispositivos de almacenamiento secundario, que en el caso de un disco en estado sólido, se organizan en páginas como su menor unidad de lectura y escritura. En esta tarea deberán experimentar con la implementación de un sistema de archivos real, que además toma en consideración algunas de las limitaciones físicas de un disco en estado sólido, como el límite de ciclos Program/Erase (P/E) de cada página, este valor indica cuántas veces una página puede ser escrita (program) o borrada (erase) antes de pasar al estado *rotten*, una vez en este estado la página no puede ser leída ni programada nuevamente. Deberán poder modificar y leer los contenidos de este disco, a través de una API desarrollada por ustedes.

Estructura del sistema de archivos `ssdfs`

El disco está compuesto por celdas de dos niveles (MLC), agrupadas en páginas, que a su vez se agrupan en bloques, a su vez agrupados en planos. La unidad más pequeña que se puede leer o escribir es una página. Sus características son las siguientes:

- Cada celda almacena 2 ~~bits~~ Byte.
- Cada página tiene 2048 celdas, o 4 KiB en total.
- Cada bloque tiene 256 páginas o 1 MiB de capacidad.

- Hay 2 planos de 1024 bloques cada uno de 1 GiB en total, que juntos hacen un disco de 2 GiB.
- Cada bloque está identificado por un puntero, que corresponde a un entero de 2 Bytes, enumerados secuencialmente a partir del bloque 0 del primer plano. Si bien cada puntero es un entero, debido a la estructura del disco, este se puede descomponer de la siguiente manera: los últimos 5 bits siempre serán 0s, el bit ~~11~~ 10 indica el plano y los primeros 10 bit (desde el bit 0 al bit 9) indican el bloque dentro de dicho plano.

Al momento de leer o escribir solo se puede hacer **de a páginas**, es decir solo se puede tener en memoria múltiplos de páginas. Además no se puede leer fracciones de páginas, o leer páginas desalineadamente.

Existen cinco tipos de bloque: *bitmap*, *lifemap*, directorio, índice y datos.

Bloque de *bitmap*: se utiliza sólo la página 0, del primer bloque del primer plano. Corresponde a un bitmap completo del disco, donde un 1 indica que el bloque correspondiente está ocupado, y 0 indica que esta está desocupado. Los bloques se encuentran ordenados de manera secuencial dentro del *bitmap*, por lo que el bit 0 corresponde al primer bloque del disco, y el bit 2047 corresponde al último bloque del disco.

Bloques de *lifemap*: corresponden a los bloques 1 y 2 del primer plano. Almacenan en orden secuencial por cada una de las páginas, un `int` de 4 Bytes, que indica la cantidad de operaciones *program-erase* a la que se ha sometido cada página. Una vez que este llega a su límite, este se debe dejar con valor -1 y sus contenidos deben ser reubicados a otra página antes de que esto ocurra. El valor de ciclos P/E para las páginas de *bitmap*, *lifemap* y directorio base debe ser siempre 0.

Bloque de directorio: Está compuesto por una secuencia de entradas de directorio, donde cada entrada ocupa 32 Bytes y contiene:

- 1 Byte: los primeros 2 bits indican si la entrada es inválida (`0b00`), otro directorio (`0b01`) o un archivo (`0b11`). Los demás bits se mantienen en 0.
- 4 Bytes: el puntero de la página donde se encuentra índice del archivo o el directorio de la carpeta.
- 27 Bytes: el nombre del archivo o carpeta, expresado con caracteres ASCII, incluyendo la extensión del archivo ((`.png, .txt`, etc). Si el nombre ocupa menos de 27 bytes, se deberá rellenar con bytes iguales a 0 hasta ocupar 27 bytes.

Al crearse el disco duro, el primer bloque disponible luego del *lifemap*, será el bloque de directorio base y este corresponde al bloque 3 del primer plano. A partir de este bloque comienza el sistema de archivos.

Bloque de índice: contiene la metadata de un archivo y la información necesaria para acceder a este. El primer bloque de un archivo siempre será el bloque de índice y contiene:

- 8 Bytes para el tamaño del archivo.
- 4048 Bytes para un máximo de 1012 punteros que apuntan a un bloque de datos.

Bloque de datos: un archivo guarda su contenido únicamente en las páginas de bloques de datos. Este bloque no puede compartir datos de dos archivos diferentes y cada bloque utilizado por un archivo se debe marcar como ocupado en el *bitmap*.

Para la lectura y escritura de Bytes en los bloques de datos debe ser realizada en orden **little endian** para mantener consistencia entre los sistemas de archivos que tendrán que implementar.

Por último, sistema de archivos almacena archivos en páginas mediante asignación indexada y permite la existencia de múltiples directorios anidados. Partiendo del directorio base, este puede contener punteros a

índices de archivos o a otros directorios, aunque no pueden haber dos o más punteros a directorios con el mismo nombre. Por ejemplo, podría haber un archivo en `~\folder1\folder2\file.txt`, lo que significa que para acceder a `file.txt`, existe un puntero desde `~` hacia `folder1`, de `folder1` a `folder2` y finalmente `folder2` tiene un puntero al índice de `file.txt`

API de `ssdfs`

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. 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*.

Las funciones que debe implementar la biblioteca son las siguientes:
Funciones generales

- `void os_mount(char* diskname, unsigned life)`: monta el disco virtual, para esto establece como variable global el archivo `.bin` correspondiente al disco. Además, define como límite de ciclos P/E al valor de `life`. La función debe poder ser llamada múltiples veces si se desea abrir diferentes discos a lo largo de la ejecución de `main.c`.

- `void os_bitmap(unsigned num)`: imprime el valor del *bitmap* para el bloque `num`. Si `num=0` se debe imprimir todo el *bitmap*. Además se debe imprimir en una segunda línea el número de bloques ocupados y libres

- `void os_lifemap(int lower, int upper)`: imprime el estado P/E de las páginas desde `lower` y `upper-1`. Si ambos valores son `-1`, se debe imprimir el *lifemap* completo. Además se debe imprimir en una segunda línea la cantidad de bloques *rotten* y la cantidad de bloques saludables.

- `int os_trim(unsigned limit)`: esta función debe recorrer todo el disco, y para cada bloque que contenga páginas cuyo valor P/E se encuentra a `limit` ciclos de pasar a estado *rotten*, reubicarla a un bloque que no contenga páginas en esta condición. Esta operación no debe corromper archivos ni directorios, por lo que mover un bloque implica actualizar todos los punteros que sea necesario para no perder su referencia. En caso de que no hayan suficientes bloques disponibles para realizar cualquiera de estas operaciones, se debe indicar la cantidad de estos, y además indicar que archivos o directorios se podrían ver afectados por pérdida de información en `limit` ciclos. Esta función retorna el número bloques que fueron reubicados exitosamente.

- `void os_tree()`: función para imprimir el árbol de directorios y archivos del sistema, a partir del directorio base.

Funciones de manerjo de archivos

- `int os_exists(char* filename)`: permite revisar si un archivo existe o no. Retorna 1 en caso de que exista, 0 en caso contrario.

- `osFile* os_open(char* filename, char mode)`: esta función abre un archivo. Si `mode='r'`, se busca el archivo `filename` y se retorna el `osFile*` que lo representa. Si `mode='w'`, se verifica que el archivo no exista, y se retorna un nuevo `osFile*` que lo representa.

- `int os_read(osFile* file_desc, void* buffer, int nbytes)`: esta función sirve 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 Bytes efectivamente leídos desde el archivo. Esto es importante si `nbytes` es mayor a la cantidad de Bytes restantes en el archivo o en el caso que el archivo contenga páginas *rotten*. La lectura de `read` se efectúa desde la posición del archivo inmediatamente posterior a la última posición leída por un llamado a `read`.

- `int os_write(osFile* file_desc, void* buffer, int nbytes)`: esta función permite escribir un archivo. Escribe en el archivo descrito por `file_desc` los `nbytes` que se encuentren en la dirección indicada por `buffer`. Retorna la cantidad de Bytes escritos en el archivo. Si se produjo un error porque no pudo seguir escribiendo, ya sea porque el disco se llenó, ya sea porque existen demasiadas páginas *rotten* o porque el archivo no puede crecer más, este número puede ser menor a `nbytes` (incluso `0`). Esta función aumenta en 1 el contador P/E en el *lifemap* asociado a cada página que se escriba.

- `int os_close(osFile* file_desc)`: esta función permite cerrar un archivo. Cierra el archivo indicado por `file_desc`. Debe garantizar que cuando esta función retorna, el archivo se encuentra actualizado en disco.

- `int os_rm(char* filename)`: esta función elimina el archivo indicado por `filename`. El bloque de índice del archivo debe ser borrado (todos sus bits puestos en 0), lo que aumenta en 1 el contador P/E asociado a dichas páginas. También se deber actualizar la página del bloque de directorio que contenía el puntero a dicho índice, lo que también incrementa su contador P/E en 1.

- `int os_mkdir(char* path)`: esta función crea un directorio con el nombre indicado. Esto incrementa en 1 el contador P/E de las páginas que sea necesario actualizar para crear las referencias a este directorio.
- `int os_rmdir(char* path)`: esta función elimina un directorio vacío con el nombre indicado. Esto incrementa en 1 el contador P/E de las páginas que sea necesario actualizar para borrar las referencias a este directorio.
- `int os_rmrfdir(char* path)`: esta función elimina un directorio con el nombre indicado, todos sus archivos y subdirectorios correspondientes. Esto incrementa en 1 el contador P/E de las páginas que sea necesario actualizar para borrar las referencias a este directorio.
- `int os_unload(char* orig, char* dest)`: esta función se encarga de copiar un archivo o carpeta referenciado por `orig` hacia un nuevo archivo o directorio de ruta `dest` en su computador.
- `int os_load(char* orig)`: esta función se encarga de copiar un archivo o los contenidos de una carpeta, 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. En caso de que el origen sea una carpeta, se deben copiar los archivos que estén dentro de esta carpeta, ignorando cualquier carpeta adicional que tenga. Esta función debe actualizar el *lifemap* según corresponda.

Ejecución

Para probar su biblioteca, debe usar un programa `main.c` que reciba un disco virtual (ej: `simdisk.bin`) de 2 GiB. 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. Por lo tanto, la ejecución del programa principal debe ser:

```
./ssdfs simdisk.bin
```

Luego, una secuencia de instrucciones que puede encontrarse en `main.c` podría ser:

```
os_mount("simdisk.bin", 5000); // Se monta 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 *bitmap*, *lifemap* en 0 para todas las páginas, 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`

Observaciones

- El funcionamiento de su implementación no debe depender de funciones que sean de la API. [1]
- La primera función a utilizar en `main.c` será siempre la que monta el disco.
- Los bloques de directorio no necesariamente están ubicados de forma continua, lo mismo para los bloques de datos.
- Al eliminar un archivo, basta con borrar sus referencias y marcar los bloques utilizadas por este como disponibles en el bitmap.
- Si se desea escribir y no queda espacio, ya sea porque hay páginas *rotten* por haber llegado a su límite P/E o porque no quedan suficientes páginas disponibles, el archivo se escribirá hasta ocupar toda la capacidad disponible.
- Una página que ha llegado a su límite P/E no puede ser leída ni escrita. Si un archivo contiene páginas *rotten*, este ya no podrá ser leído correctamente.
- En el sistema **no existirán** dos archivos con el mismo path absoluto. Sin embargo, pueden existir archivos con el mismo nombre en carpetas distintas.

[1] Pueden hacer funciones auxiliares en su main, pero no pueden agregar, ni modificar funciones de la API (por ejemplo, una función unmount para liberar memoria)

Entrega

La entrega del código de su API será a traves del servidor del curso, en la fecha estipulada. La entrega del proyecto deberá ser en una carpeta llamada `P1`, en la carpeta de cualquiera de los integrantes del curso.

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.

La entrega del código de su API será a través del servidor del curso, en la fecha estipulada. La entrega del proyecto deberá ser en una carpeta llamada `P1`, en la carpeta base de cualquiera de los integrantes del curso.

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 la carpeta especificada, el proyecto **no se corregirá**.

Evaluación

- **1.00 pts.** Estructura de sistema de archivos[2].
  - **0.20 pts.** Representación de bloques directorio.
  - **0.20 pts.** Representación de bloque *bitmap*.
  - **0.20 pts.** Representación de bloques de *lifemap*.
  - **0.20 pts.** Representación de bloques de índice.
  - **0.20 pts.** Representación de bloque de datos.
- **6.0 pts.** Funciones de biblioteca.
  - **0.20 pts.** `os_mount`.
  - **0.20 pts.** `os_bitmap`.
  - **0.20 pts.** `os_lifemap`.
  - **1.50 pts.** `os_trim`
  - **0.20 pts.** `os_exists`.
  - **0.20 pts.** `os_tree`.
  - **0.20 pts.** `os_open`.
  - **0.30 pts.** `os_read`.
  - **0.40 pts.** `os_write`.
  - **0.20 pts.** `os_close`.
  - **0.20 pts.** `os_mkdir`.
  - **0.30 pts.** `os_rmdir`
  - **0.20 pts.** `os_rmrfdir`
  - **0.30 pts.** `os_rm`.
  - **0.70 pts.** `os_unload`.
  - **0.70 pts.** `os_load`.
- **1.00 pts.** Manejo de memoria perfecto y sin errores (**Valgrind**).

[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**.

Descuentos

- Descuento de 0.5 puntos por subir **archivos binarios** (programas compilados).
- Descuento de 0.5 puntos por leer o escribir desalineados respecto a las páginas y por no leer en múltiplos de una página.
- Descuento de 1 punto si la entrega **no tiene un `Makefile`, no compila o no funciona (*segmentation fault*)**.
- Descuento de 3 puntos si se sube alguno de los archivos `simdisk.bin` al servidor.

Corrección

A diferencia de las tareas, este proyecto será corregido junto a ustedes. Se hará uso de la plataforma Google Meets 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.
5. Puede (y se recomienda) hacer uso de más de un programa `main.c`, de forma que estos evidencien distintas funcionalidades de su API.

Política de atraso

Se puede hacer entrega de la tarea con un máximo de 4 días hábiles de atraso. La fórmula a seguir es la siguiente:

$$N_{T_1}^{\text{Atraso}} = \min(N_{T_1}, 8.0 - 0.75 \cdot d)$$

Siendo d la cantidad de días de atraso. Notar que esto equivale a un descuento *soft*, es decir, cambia la nota máxima alcanzable y no se realiza un descuento directo sobre la nota obtenida. El uso de días de atraso no implica días extras para alguna tarea futura, por lo que deben usarse bajo su propio riesgo.

Para poder hacer entrega atrasada se debe responder el formulario de atraso.

Preguntas

Cualquier duda preguntar a través del **foro**.