**Sistemas Operativos**

**Trabajo práctico nº 2**

Primer cuatrimestre - 2013

Profesores:

* Verónica Tramanelli
* Hugo Eduardo Etchegoyen

Integrantes:

* Germán Rodrigo Romarión (51296)
* Gabriel Andrés Zanzotti (52067)

**Índice**

**1. Introducción 3**

**2. Selección del driver 3**

**3. Implementación del driver y**

**estructuras seleccionadas 4**

**4. Reconstrucción de archivos**

**en memoria 2**

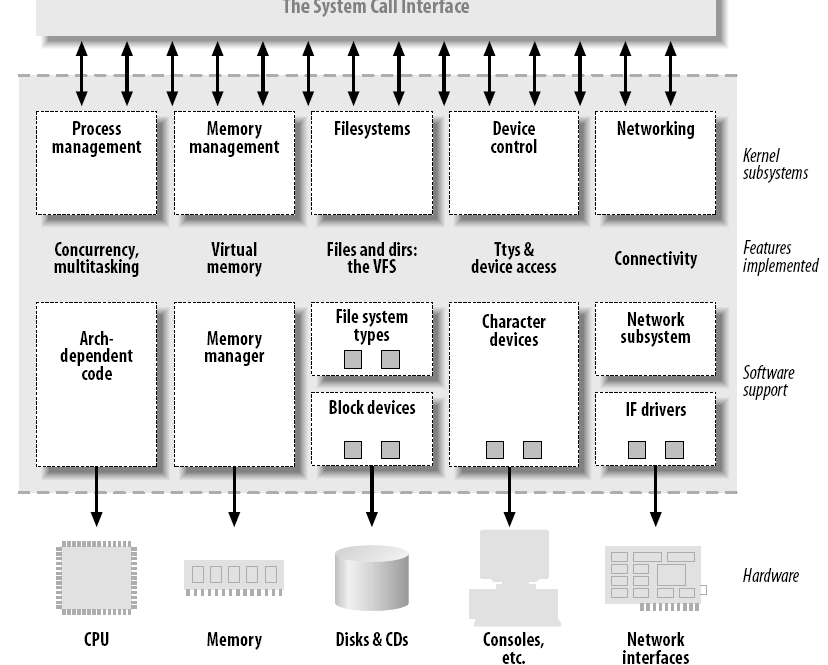
**1. Introducción**

En esta oportunidad la cátedra de Sistemas Operativos nos solicitó desarrollar un driver a elección, junto su implementación. Dicho driver debía ejecutarse sobre el kernel de un pequeño sistema operativo multitarea.

El presente informe se encargará de detallar a fondo nuestra elección, y cada una de las estructuras utilizadas y el diseño ideado para funcionar en armonía con el sistema, junto con la debida justificación para cada caso.

**2. Selección del driver**

La cátedra nos ha dado una serie de opciones para elegir entre varios drivers:



Las opciones fueron las siguientes:

* Driver de Disco y Sistemas de Archivos.
* Driver de Video y Sistema de consolas.
* Driver e interfaz de red.

Nosotros hemos decidido seleccionar el driver de disco por varias razones: creímos que sería el más sencillo de utilizar, porque pensamos que sería el más intuitivo a la hora de implementarlo. Además del driver de disco, es decir, el programa que se encarga de comunicarse directamente con el disco y de escribir y leer los datos pertinentes, hemos tenido que desarrollar todo el sistema de archivos con el cual se manejará el sistema multitarea provisto por la cátedra.

En la siguiente sección de este informe se tratarán las diversas estructuras utilizadas para la implementación del sistema de archivos y el diseño.

**3. Implementación del driver y estructuras seleccionadas**

**Sistema de archivos:** El sistema de archivos cuenta con dos tipos distintos de estructuras. Una de ellas es utilizada para mantener un árbol de dependencias en memoria, y la otra es utilizada para ser escrita en disco.

La estructura utilizada en memoria, de tipo *file\_sys\_t* tiene el siguiente aspecto:

***typedef* struct file\_sys** {

**char** file\_name[FILE\_NAME\_SIZE];

**bool** is\_file;

**struct file\_sys** \*father\_directory;

**int** sector;

**struct file\_sys** \*dependencies[DEPENDENCIES\_SIZE];

**struct file\_sys** \*next;

**int** next\_sector;

**int** dependencies\_amount;

**struct file** \*content;

}***file\_sys\_t***;

Similar a lo que plantea Linux, en nuestro diseño todo es un archivo: es decir, tanto los directorios como los archivos en si poseen casi las mismas características. La diferencia entre un archivo y un directorio radica en que un directorio posee dependencias, es decir, puede contener a otros directorios y archivos dentro suyo. Por otro lado, un archivo no puede poseer dependencias; en cambio, un archivo tiene contenido.

Para poder diferenciar a un archivo de un directorio, en la estructura mostrada anteriormente existe un valor booleano *is\_file* que determina si es o no un archivo.

En un principio se nos había ocurrido limitar la estructura de cada archivo y directorio para que no pudiera ocupar más de 1 sector de disco (512 bytes): este diseño resultó muy acotado y obsoleto, por lo que inmediatamente tuvimos que encontrar una solución. La mejor solución que se nos ocurrió fue colocar un puntero a otro archivo en caso de que contuviera más dependencias de las que soportamos, o en caso de que un archivo contuviera más contenido del admitido. En el primer caso, el puntero apuntaría a otra estructura del mismo tipo. En el segundo, *content* apuntaría a una estructura de tipo *file\_t*, que contendrá el resto de la información de dicho archivo.

Otros campos que componen a esta estructura son:

* ***File\_name:*** Indica el nombre del directorio/archivo.
* ***Sector:*** Indica el sector que ocupa en disco dicho directorio/archivo.
* ***Dependencies\_amount:*** Indica la cantidad de sub-archivos/directorios posee un directorio (solo usado por directorios).

Cada vez que guardamos en disco las estructuras existentes en memoria, usamos una estructura de datos distinta. Esto es porque en disco los punteros no tienen sentido, por lo que es necesario crear la siguiente estructura:

**typedef struct serialized\_file** {

**char** name[FILE\_NAME\_SIZE];

**int** sector;

**bool** is\_file;

**char** content[FILE\_SIZE];

**int** dependencies[DEPENDENCIES\_SIZE];

**int** dependencies\_amount;

**int** next\_sector;

}**serialized\_file\_t;**

Esta estructura es muy similar a la anterior, pero en lugar de poseer un puntero a una estructura siguiente en caso de no soportar la cantidad de información a contener en ella, contiene un valor *next\_sector*, que indica en qué sector se encontrará el resto de la información que no está aquí.

**4. Reconstrucción de archivos en memoria**

La idea de tener un sistema de archivos es poder trabajar en una computadora con un sistema que garantice la total preservación de los archivos y carpetas.

Para poder levantar a memoria todos los archivos que se escribieron en disco la última vez, lo que hacemos es leer el tamaño de la tabla de sectores de disco, leemos dicha tabla con ese tamaño, y en la primera posición de la tabla decidimos guardar al directorio *root*, que será el directorio principal de todos.

Para poder saber de donde leeremos la tabla de sectores, y si es que el sistema ya había escrito en disco antes, contamos con un sistema de reconocimiento bastante sencillo: la primera vez que se prende el sistema, se verifica que en el sector 0 y 1 del disco hayan escritos dos números particulares. De ser así, la el sistema sabrá de donde leer el resto de las dependencias, que se guardan siempre en lugares fijos en disco.

Una vez que levantamos de disco la carpeta raíz, comenzamos a reconstruir el árbol de dependencias en memoria.

**5. Conclusiones**

El sistema de archivos parece funcionar de manera infalible, por lo que hemos cumplido con la consigna al pie de la letra. Sin embargo, hay varios aspectos de optimización que deberíamos solucionar: al principio en la ejecución del sistema, levantamos de disco a memoria absolutamente todo lo que haya escrito, por lo que si tengo una memoria RAM de 512 MB y un disco duro de 1TB, el sistema fallará.

Hemos optimizado el tiempo de lecto-escritura: en lugar de poner un *sleep()* para esperar a que el disco lea y escriba, tenemos una función que pregunta por uno de los flags del disco para saber si está listo o no para leer o escribir.