BrunOS®

Un sistema operativo .. diferente!

*Karpovsky - Martinez - Mesa*

Presentado el

[25/06/2012]

File System 3

Introducción 3

Diagramas 3

Estructuras del File System 5

Comandos implementados 6

Interfaz gráfica 6

Consideraciones a tener en cuenta 7

Conclusiones 8

Trabajo base 8

Assembler 8

Conceptos aprendidos 8

Manejo de disco 9

# File System

## Introducción

El requerimiento principal del tercer trabajo práctico era extender el sistema multitasker y agregarle un file system. Esto implicó la programación de un driver de disco para poder persistir la información, como así también el planeamiento de una estructura de datos adecuada para poder crear un sistema de archivos similar al utilizado en los sistemas operativos convencionales.

Para el diseño del mismo se optó por tomar inspiración de los sistemas **UNIX** y basar el desarrollo en la forma que tiene UNIX de tratar a los archivos. A continuación se desarrollarán las estructuras utilizadas para el sistema de archivos como así también los comandos que éste soporta.

Cabe destacarse que el presente informe es un anexo al informe entregado en el trabajo práctico número dos: todas las alcaraciones sobre la implementación del sistema multitasker pueden ser consultadas en el informe anterior ya que las mismas no han variado. La única diferencia entre la entrega anterior y la entrega actual es que en la estructura que representa a un proceso **Task\_t** se ha incluído un campo *inode* con el fin de saber cuál es el directorio en el cual está ejecutándose esa tarea.

## Diagramas

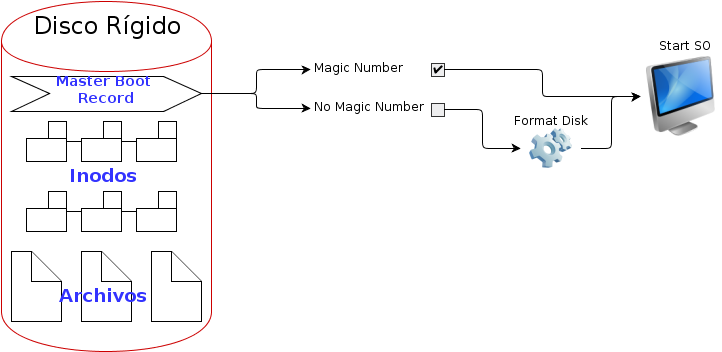
Es importante saber si, al momento de bootear el sistema operativo, el disco está o no formateado. Para lograrlo, se utiliza la siguiente técnica: el primer sector del disco contiene el **Master Boot Record**. El él se guarda un *magic number* que nos indica si el disco tiene o no formato. Por ende el proceso de booteo es el siguiente:

1. Hacer una lectura (*read)* del primer sector del disco y guardar en una variable entera el primer dato que éste contenga.
2. Validar el dato obtenido con el *magic number* (número que sólo nuestro sistema operativo conoce). Las probabilidades de que justo en el primer sector de un disco no formateado esté alojado por casualidad el *magic number* son ínfimas.
   1. Caso 1: *Magic number* correcto => Se inicia el sistema operativo.
   2. Caso 2: *Magic number*  incorrecto => Se formatea el disco y se lo deja listo para que el sistema operativo pueda ser iniciado y pueda persistir información en él.

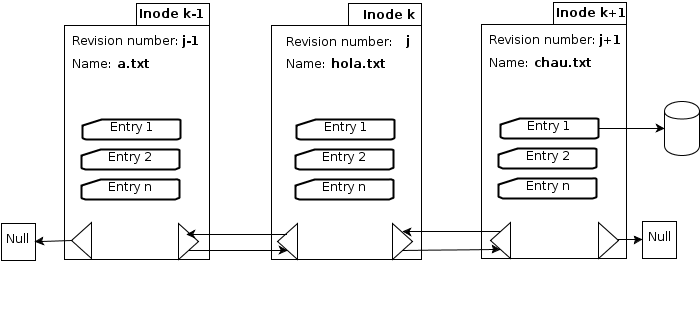
Asimismo el disco está subdividido de la siguiente forma:

* **Master boot record:** Como se explicó, el MBR nos sirve para validar si el disco está o no formateado y está alojado al comienzo del disco.
* **Superblock:** Luego del MBR, los primeros sectores del disco alojan un superbloque de inodos.
* **Archivos:** Por último, todo el resto del disco está destinado a los archivos persé.

El siguiente diagrama esquematiza la subdivisión del disco y el proceso de booteo:



Asimismo, los inodos de nuestro sistema operativos pueden ser esquematizados como sigue:



## Estructuras del File System

Las estructuras más relevantes que componen el file system son las estructuras de **inodo**, **fileentry** y **superbloque.**

**Inodos:**

**struct** inode\_t{

**int** inode\_number;

**int** count; // Number of references

type\_t type;

ushort sectors[MAX\_SECTORS];

**char** name[MAX\_FILENAME];

inode\_t \* next; // SVN

inode\_t \* prev; // SVN

inode\_t \* parent;

size\_t size;

**int** rev\_no;

**int** status;

};

**File entry:**

**struct** fileentry\_t{

type\_t type;

**char** name[MAX\_FILENAME];

**int** inode\_number;

uint position;

state\_t state;

};

**Superbloque:**

**struct** superblock\_t{

**int** check\_number;

inode\_t inodes[MAX\_INODES];

};

En líneas generales el superbloque mantiene una lista de todos los inodos con los que opera el sistema de archivos. Luego cada inodo tiene un número distinto que lo identifica unívocamente entre todos los demás como así también un arreglo de sectores a los que "apunta". En dichos sectores está alojada la información persé. Por último un *fileentry* conserva un estado (presente/ausente/borrado/etc), una posición dentro de la tabla de archivos referenciados por un inodo y un tipo (directorio/archivo/link).

## Comandos implementados

Existen dos tipos de comandos: los que son transparentes al versionado y los que no lo son (\*)

* **ls:** Lista todos los archivos y directorios que se encuentren en el directorio actual (pwd).
* **lsRemoved** *(\*)***:** Lista todos los archivos y directorios que se encuentren en el directorio actual (pwd) aunque estos no estén "presentes" (hayan sido removidos de manera soft).
* **cd** *dirname***:** Cambia el pwd por *dirname*.
* **touch** *filename*: Crea en el pwd un archivo vacío con nombre "filename".
* **mkdir** *dirname*: Crea en el pwd un directorio vacío con nombre "dirname" (sólo contendrá referencia a él mismo "." y a su padre "..").
* **rm** *filename:* Borra, en caso de existir, el archivo *filename* de manera soft.
* **rmHard** *filename (\*):* Borra, en caso de existir, el archivo *filename* de manera hard (no recuperable).
* **rmRec** *dirname:* Borra, en caso de existir, el directorio con nombre *dirname* de forma recursiva (borra a él y todo su contenido).
* **vh** *filename (\*)*: Muestra la version history de un archivo (las revisiones por las que fué pasando).
* **revert** *filename versionN (\*):* Revierte, en caso de existir, el archivo *filename* a la versión *versionN*.
* **mv** *filename anotherFileName*: Cambia el nombre del archivo *filename* por *"anotherFileName"*.
* **mv** *filename dirname*: Mueve el archivo *filename* al directorio con nombre *dirname.*
* **cp** *filename dirname:* Copia, en caso de existir, el archivo *filename* al directorio *dirname*.

## Interfaz gráfica

Creemos que es sumamente importante proveer una buena interfaz gráfica para los usuarios de nuestro SO. Es por ello que realizamos varias acciones de modo que sea más intuitivo la operación sobre el mismo:

* **Colores en ls:** Para que sea simple diferenciar un archivo de un directorio, los mismos se visualizarán con colores distintos al ejecutar el comando *ls*.
* **Archivos removidos:** Los archivos removidos de manera soft se muestran en color rojo y entre paréntesis al ejecutar *lsRemoved.*
* **Version history:** Version history presenta una visualización tabular intuitiva y colorida, diferenciando con un color la versión y datos actuales del archivo y con otro las versiones anteriores.

## Consideraciones a tener en cuenta

Comprendemos que hubiera sido deseable realizar un mejor manejo de comandos en la consola. En vez de tener un comando **ls** y otro **ls -removed** ambos podrían haber sido resumidos en **ls** y **ls --removed** (pasando como parámetro la especificación "removed" al ls en vez de generar un comando completamente distinto). Lo mismo sucede con **rm**, **rmHard** y **rmRec**: UNIX resuelve esto simplemente pidiendo *rm -r* para borrados recursivos.

Por cuestiones de tiempo se decidió optar por la alternativa simple y crear distintos comandos para las distintas acciones disponibles independientemente de si estas refieren o no a la misma clase de acciones/comportamiento. No sería difícil adaptar nuestra implementación para que los comandos soporten múltiples parámetros (es cuestión de implementar el comando *strtok* para poder hacer un split del string recibido en tokens y luego parsear cada parte de este).

Lo mismo sucede con los paths relativos y absolutos: nuestro SO opera de manera local; sólo se pueden copiar, mover y consultar archivos y carpetas presentes en el pwd. Esto se debe al mismo motivo explicitado en el punto anterior. Lograr hacer **cd dir1/dir2/dir3** o **rm dir1/file1.txt** no tiene ninguna complejidad una vez implementada la función *strtok*: basta con validar que todos los archivos o carpetas detallados en el string existan y luego proceder de la forma que se procede actualmente.

Considerábamos importante destacar estas cuestiones para que así quede reflejado que las decisiones de implementación tomadas fueron meramente basadas en el tiempo disponible para realizar el trabajo y no en la complejidad de las acciones que éstas deben realizar. El desarrollo está suficientemente modularizado como para de manera muy sencilla implementar los comandos con múltiples parámetros y los paths para la operación sobre carpetas distintas a la pwd.

# Conclusiones

## Trabajo base

Dado que la consigna del trabajo establecía retomar un desarrollo previo de la misma asignatura (Sistemas Operativos), parte de las ventajas de mantener el grupo en el presente trabajo fue que todos conocíamos a la perfección las estructuras y la diagramación del sistema operativo implementado.

Es importante destacar la importancia de la buena documentación del código como así también el buen estilo y modularización del mismo para poder seguir un desarrollo viejo y agregarle nuevas funcionalidades.

## Assembler

Si bien la realización del trabajo no fue sencilla, y volver a programar en bajo nivel luego de programar en Java fue algo extraño, al programar en assembler uno tiene control total sobre el micro procesador y puede realizar tareas que con otros lenguajes resulta imposible. Además al realizar el trabajo se pudo aprender mucho del funcionamiento interno de la PC y de su interacción con los dispositivos más importantes de I/O, como así también fijar el concepto de sistemas multi-tarea, nociones de scheduling y de persistencia de datos.

## Conceptos aprendidos

Una vez finalizado el trabajo entendimos qué era realmente un sistema operativo básico multitarea y la complejidad de su desarrollo. Asimismo comprendimos las ventajas de un sistema con memoria paginada, y la seguridad que brinda esta característica. Aunque lograr un eficiente y seguro manejo de memoria es muy complejo. Por último destacamos la versatilidad otorgada por un sistema operativo que logra persistir sus datos en el disco rígido ya que lo hace mucho más funcional que un sistema operativo cuyas operaciones son volátiles.

Nos resultó algo dificultosa la tarea de *debugging*  en el proceso de desarrollo debido a que no encontramos herramientas que la faciliten. Una de las problemáticas de la programación a tan bajo nivel es que, por ejemplo, ante un herror la máquina virtual (bochs) automáticamente se cuelga y/o se reinicia. Este error puede ser tan "sencillo" como estar escribiendo en una página ausente o estar desreferenciando un puntero mal asignado. La estrategia que encontramos para subsanar esta problemática fue la implementación de una función *kprintf()* la cual funciona de manera similar a printf() pero escribe directamente en video (0xB8000); es decir que no se fija ni tiene en cuenta las TTYs, los buffers, los procesos y demás sino que directamente imprime en pantalla. De esta forma podíamos ir imprimiendo los valores de las variables y/o los registros e ir evaluando por qué es que las cosas no se ejecutaban de la forma esperada.

A su vez, otra estrategia utilizada fue la de crear una función *halt* la cual explícitamente ejecutaba la instrucción *hlt* del procesador y hacia por ende que este detenga su ejecución. La técnica consitía en ir "bajando" la línea del halt hasta encontrar la línea de código que producía que la máquina virtual se reiniciara. Una vez encontrada esta línea se analizaba el problema puntual y se seguía desplazando la línea de halt hacia adelante hasta encontrar un posible nuevo foco de conflicto.

## Manejo de disco

Se logró entender la complejidad que implica un buen manejo del disco. Es de suma importancia proveer métodos de corrección automática de errores ya sea para la escritura como así también para la lectura de disco. Es bastante sencillo, en caso de no implementarlos, corromper el filesystem y/o obtener archivos defectuosos con información no consistente.

Se entendió la estrategia utilizada por UNIX en el manejo de archivos y se valoró la técnica de referenciar a los arhivos con una estructura externa (inodos); esto le da mucha versatilidad al filesystem y permite hacer operaciones de manera sencilla que, en caso de no pensarlo así, serían complejas.