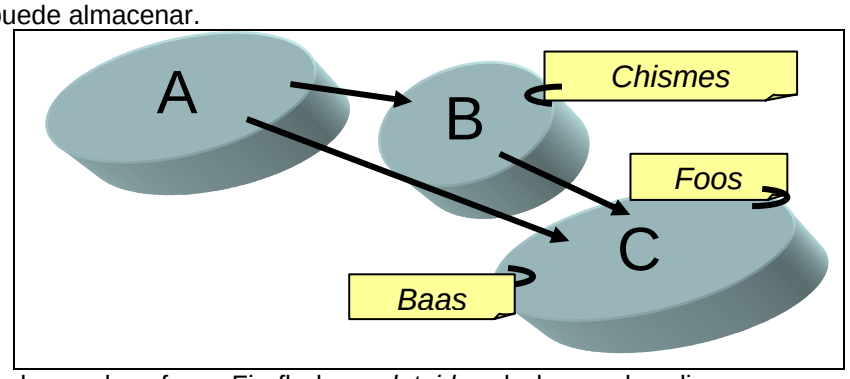
SISTEMA DE FICHEROS – FEBRERO 2006

En el año 2525 el desarrollo software ha alcanzado cotas de gran evolución, permitiendo la construcción de sistemas, por medio de piezas modulares, denominadas *pelotoides*, que se encajan desde una interfaz de desarrollo tridimensional. Realizando un estudio histórico, el profesor Rufus T. Firefly, arqueólogo informático, encuentra un arcaico libro editado a comienzos del siglo XXI en la UPM en el que se describe el diseño de sistemas de ficheros basados en sistemas de archivos virtuales (VFS).
A lo largo de su investigación, el Prof. Firefly consigue recopilar diversos módulos de la época, encapsulados como *pelotoides*, que son:

- Servidor de bloques
- Lógica del manejador del SF ext3
- Caché de bloques
- Sistema de archivos virtual (VFS)
- *Driver* de un dispositivo (disco)
- Servidor de archivos

Sabiendo que la construcción de sistemas por medio de la interfaz tridimensional, permite enlazar los *pelotoides*, lo cual indica que el *pelotoide* originario del enlace solicita servicios ofertados por el *pelotoide* destinatario del mismo. Además del enlace de módulos, la interfaz permite asociar a cada *pelotoide* uno o varios espacios donde almacenará datos propios, denominado *espacioide*. Cada uno de los *espacioides* debe tener asociado un nombre que indique qué tipo de dato puede almacenar.
Ejemplo: *El pelotoide A usa los servicios del pelotoide B. El pelotoide C da servicio, indistintamente a A y B. El pelotoide B tiene un espacioide llamado Chismes (almacena estructuras Chisme). Y C tiene otros dos, llamados Foos y Baas.*



a)- ¿Cómo debería colocar el profesor Firefly los *pelotoides* de los cuales dispone para diseñar un sistema que emulase un servicio de archivos de un sistema UNIX? **[1/10 punto]**

b)- ¿Qué *espaciodes* se deberían crear en los módulos anteriores? ¿Cuáles serían los datos que se tendrían que almacenar en cada uno de ellos? Todas las estructuras de datos principales de cada módulo deberían estar recogidas. **[2/10 puntos]**

Para poder probar su simulador, el Prof. Firefly pide a su aventajado alumno Kenny Bell que proponga un juego de pruebas con el que evaluarlo. Antes de comenzar, Kenny se da cuenta de que debe agenciarse uno de esos vetustos discos duros de 300GB e introducir datos en el mismo sobre un directorio raíz y un archivo:

c)- ¿Cuál será el contenido inicial del disco (datos y metadatos)? **[1/10 punto]**

Una de las características más avanzadas de la programación basada en *pelotoides* es que permite trazar todas las solicitudes que se hacen a uno de estos elementos. Para evaluar su primera prueba, Kenny decide trazar las solicitudes que se lanzan al *driver* del dispositivo. Para ello diseña un sencillo programa que lee un fichero (de 100bytes) y lo escribe en otro nuevo:

d)- Si es la primera prueba que se realiza (no hay nada cargado en memoria) ¿Cuáles serían las peticiones que llegan al *driver*? NOTA: Para mostrarlo hay que dibujar una tabla, que en su columna izquierda muestra el código del programa y en la derecha las peticiones que se hacen al *driver* por cada sentencia. **[2/10 puntos]**

El Prof. Firefly, pide a Kenny una variante de la prueba anterior. En este caso, hay dos procesos que ejecutan el mismo programa. La ejecución del uno se detiene en el momento que realiza una llamada al sistema y se reanuda el otro. Los dos procesos leerán del mismo fichero pero escribirán ficheros diferentes (con distintos nombres).

e)- ¿Cuáles serían las modificaciones sobre los datos de cada *espacioide*? NOTA: Al igual que en el caso anterior el Prof. Firefly solicita una tabla con el código que se ejecuta a la izquierda y el o los datos que se modifican a la derecha **[3/10 puntos]**

Durante la realización de sus pruebas Kenny se da cuenta de una extraña anomalía, en una de ellas. Su programa de prueba abría un fichero existente, y se quedaba esperando. Por error, en ese instante otro proceso sobrescribe el fichero que el primero tiene abierto. Al reanudar su ejecución el primer proceso lee los datos que el segundo proceso ha escrito. Pero, si el segundo proceso, no sobrescribe, sino que borra primero el fichero y luego crea otro con el mismo nombre, el resultado es que el programa inicial lee el contenido original del fichero borrado, y no lo que ha escrito el segundo proceso.

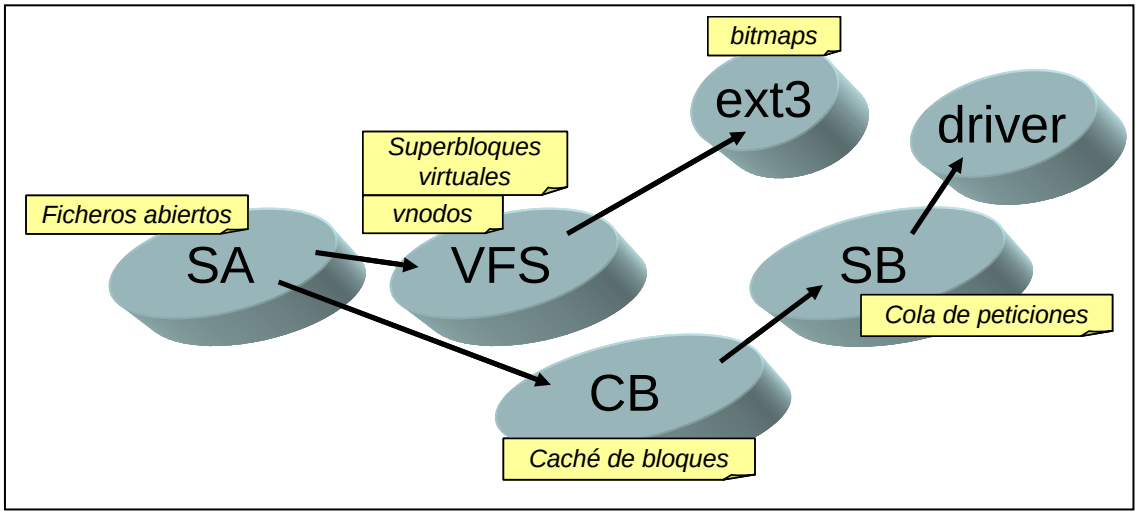
f)- Al plantear su duda al Prof. Firefly, ¿cuál sería su explicación? **[2/10 puntos]**

SOLUCIÓN

a)- ¿Cómo debería colocar el profesor Firefly los *pelotoides* de los cuales dispone para diseñar un sistema que emulase un servicio de archivos de un sistema UNIX? **[1 punto]**

b)- ¿Qué *espaciodes* se deberían crear en los módulos anteriores? ¿Cuáles serían los datos que se tendrían que almacenar en cada uno de ellos? Todas las estructuras de datos principales de cada módulo deberían estar recogidas. **[2 puntos]**

Un diseño razonable por medio de esta fabulosa programación orientada a *pelotoides* (POP), podría ser el siguiente:



El servidor de archivos (SA) se encarga de recibir las llamadas al sistema relacionadas con el servicio de archivos. Este elemento, convierte estas peticiones en operaciones sobre ficheros, es por eso que es el lugar razonable donde ubicar la tabla de ficheros abiertos (tabla a la que apuntan los BCPs de los procesos por medio de sus descriptores de ficheros, fds). Esta tabla contendrá la posición donde se esta leyendo/escribiendo en el fichero y el contador de cuantos fds apuntan a esa estructura. El SA invocaría servicios del VFS (*Virtual File System*), este módulo se encarga de gestionar los diferentes manejadores de sistemas de ficheros concretos, como es el caso de ext3. VFS debe mantener información de los i-nodos cargados en memoria (por medio de la abstracción de datos de los v-nodos. Además puede tener una tabla (opcional) con los superbloques de los sistemas de ficheros montados. Por su parte el manejador de ext3 tendrá que gestionar los metadatos específicos que dicho SF tiene para gestionar, por ejemplo el espacio libre, tal sería el caso de los *bitmaps* en memoria.

El SA y el servidor de bloques (SB) interactúan por medio de la caché de bloques (CB), que evidentemente tendrá que tener espacio para almacenar los bloques de datos cargados en memoria. Si es muy importante que la CB esté entre los dos servidores. El SB, en realidad sirve las peticiones que la caché no puede proporcionar. El SB por su parte solicita peticiones al driver del dispositivo, para poder satisfacer la cola de peticiones de lectura/escritura de bloques.

La solución, no es necesariamente única, las diferencias que pueden darse es que el VFS o incluso el propio manejador de ext3 acceda directamente a la caché de bloques. Evidentemente, configuraciones en las cuales el SA está por debajo de VFS, que la caché de bloques no sea el interfaz de acceso al SB o que el manejador de ext3, está junto al driver no son admisibles.

c)- ¿Cuál será el contenido inicial del disco (datos y metadatos)? **[1 punto]**

En el disco reside el layout del sistema de ficheros, es decir los datos estabilizados en el dispositivo. Este layout, es el formato (contenido de una partición), es decir:

- Superbloque
- Bitmaps (i-nodos y bloques)

- i-nodos
- Bloques de datos

Sobre los datos en concreto, se tendrán ocupados 2 i-nodos (uno para el directorio raíz, y otro por el fichero). Además un bloque de datos (para el contenido del directorio), y tantos como necesiten el archivo (si fuese muy grande además se necesitarían bloques de indirección: simple, doble, etc.).

d)- Si es la primera prueba que se realiza (no hay nada cargado en memoria) ¿Cuáles serían las peticiones que llegan al *drive*? **[2 puntos]**

INSTRUCCIÓN	**PETICIONES AL DRIVER**
`fd_orig=open("/orig",O_RDONLY);`	Leer_bloque(i-nodo-dir-raíz); Leer_bloque(datos-dir-raíz); Leer_bloque(i-nodo-fich-orig); [1]
`fd_dest=creat("/dest",0666);`	[2] Leer_bloque(i-nodo-fich-dest); [3] Escribir_bloque(i-nodo-fich-dest); [4]
`read(fd_orig,buff,100);`	Leer_bloque(bloque-fich-orig);
`write(fd_dest,buff,100);`	[5]
`close(fd_orig);`	Escribir_bloque(i-nodo-fich-orig); [6]
`close(fd_dest);`	Escribir_bloque(i-nodo-fich-dest); [6]

[1] Esta operación podría no darse si el bloque donde reside el i-nodo del directorio raíz y el del fichero es el mismo. Hay varios i-nodos por bloque de i-nodos.
[2] La caché de bloques contendrá ya el i-nodo y el contenido del directorio raíz.
[3] Se tiene que crear un nuevo i-nodo, pero para ello hay que traerse el bloque donde se va a ponerse el i-nodo. Al igual que en [1], este i-nodo puede caer en el mismo bloque.
[4] Habitualmente los i-nodos tiene política de escritura *write-through*. También se escribiría la entrada entre los datos del directorio, pero como datos que son se aplica la política *delayed-write*.
[5] La escritura de los bloques de datos siempre es *delayed-write*.
[6] Al hacer un *close* hay que actualizar las fechas de último acceso/modificación del fichero, por lo demás ídem a [4].

e)- ¿Cuáles serían las modificaciones sobre los datos de cada *espacioide*? **[3 puntos]**

INSTRUCCIÓN	**P**	**MODIFICACIONES DE LOS DATOS**
`fd_orig=open("/orig",O_RDONLY);`	A	1. Se crea una nueva entrada en ficheros abiertos 2. Se crea un nuevo v-nodo 3. Se ocupan varias entradas de la caché de bloques (dir. raíz y bloque del i-nodo del fichero). 4. Varias entradas en la cola de peticiones
`fd_orig=open("/orig",O_RDONLY);`	B	1. Se crea una nueva entrada en ficheros abiertos (es otro *open*) 2. Se comparte el mismo v-nodo (incrementa contador de usos (#opens)).
`fd_dest=creat("/dest",0666);`	A	1. Se crea una nueva entrada en ficheros abiertos 2. Se crea un nuevo v-nodo 3. Se puede ocupar una nueva entrada de la caché de bloques (si el bloque del i-nodo del fichero es diferente de los anteriores). 4. Varias entradas en la cola de peticiones
`fd_dest=creat("/dest",0666);`	B	1. Se crea una nueva entrada en ficheros abiertos 2. Se crea un nuevo v-nodo 3. Se puede ocupar una nueva entrada de la caché de bloques (si el bloque del i-nodo del

		fichero es diferente de los anteriores). 4. Varias entradas en la cola de peticiones
`read(fd_orig,buff,100);`	A	1. Modificación del campo de posición de lectura en la tabla intermedia 2. Una entrada de la cola de peticiones
`read(fd_orig,buff,100);`	B	1. Modificación del campo de posición de lectura en la tabla intermedia
`write(fd_dest,buff,100);`	A	1. Modificación del campo de posición de lectura en la tabla intermedia
`write(fd_dest,buff,100);`	B	1. Modificación del campo de posición de lectura en la tabla intermedia
`close(fd_orig);`	A	1. Se elimina la entrada en la tabla intermedia de ficheros abiertos y se reduce el número del contador del v-nodo (#opens).
`close(fd_orig);`	B	1. Se elimina la entrada en la tabla intermedia de ficheros abiertos y se reduce el número del contador del v-nodo (#opens). 2. Se elimina el v-nodo. En algunos sistemas los v-nodos se conservan en memoria par agilizar accesos futuros.
`close(fd_dest);`	A	1. Se elimina la entrada en la tabla intermedia de ficheros abiertos y se reduce el número del contador del v-nodo (#opens). 2. Se elimina el v-nodo. En algunos sistemas los v-nodos se conservan en memoria par agilizar accesos futuros.
`close(fd_dest);`	B	1. Se elimina la entrada en la tabla intermedia de ficheros abiertos y se reduce el número del contador del v-nodo (#opens). 2. Se elimina el v-nodo. En algunos sistemas los v-nodos se conservan en memoria par agilizar accesos futuros.

f)- Al plantear su duda al Prof. Firefly, ¿cuál sería su explicación? [2/10 puntos]

El proceso inicial abre el fichero, por lo tanto crea o referencia un v-nodo en memoria. Los datos del fichero, como no se ha hecho ninguna lectura no están cargados en memoria.

En uno de los casos otro proceso sobrescribe el fichero. Este comportamiento se explica en base a la semántica Unix de co-utilización. La implementación de la misma es la siguiente: Este proceso referenciará al mismo v-nodo (al tratarse del mismo archivo). Después de eso, los bloques de datos de dicho fichero se traen a memoria y se modifica su contenido. El proceso original, tras despertarse, al realizar la lectura del fichero accederá a los bloques de datos, que estarán ya en la caché, pero que tendrán el contenido escrito por el segundo proceso.

El otro escenario es diferente, el proceso original está esperando con un v-nodo abierto. En ese instante otro proceso solicita el borrado del fichero. El fichero se borra, es decir su entrada desaparece del directorio que lo contiene, pero el i-nodo del disco no se libera, ni tampoco los bloques de datos. Eso se debe a que se encuentra en utilización. Su liberación queda diferida al momento en el cual el proceso original cierre el fichero (*close*). De esta forma la situación se queda con que el v-nodo en memoria y el i-nodo en disco siguen existiendo, así como los bloques de datos. La explicación se debe a que la creación de un segundo fichero (aunque sea con el mismo nombre) utilizará otro i-nodo de disco. Este i-nodo nada tendrá que ver con el del fichero original y su proyección en memoria será otro v-nodo y sus bloques otros también.