El Buffer Cache

El Kernel puede leer y escribir directamente del y en el disco como respuesta a los accesos del sistema, pero el tiempo de respuesta y el throughput serian pobres debido a lo lento que es el acceso a disco y la transferencia de informacion.

Debido a esto, el Kernel utiliza un pool de buffers para minimizar los accesos a disco, llamado **buffer cache,** que contiene los bloques de datos recientemente utilizados.

Cuando el Kernel debe acceder a un bloque de datos entonces, lo que va a hacer es fijarse si el bloque no se encuentra ya en el buffer cache, si esta, se ahorro una lectura, si no esta entonces accede a disco y leer la informacion y esta queda en el buffer cacheada, para futuras lecturas. El Kernel tambien trata de minimizar las escrituras en disco, utilizando el estado del buffer **DELAYED WRITE**, para hacer que el bloque se grabe a disco justo antes de ser reasignado.

Headers

El buffer consta de dos partes: una constituye los datos en si, y la otra el header, que identifica al buffer.

El buffer no es mas que una copia del bloque de datos de disco en memoria. El header contiene:

- Numero de device
- Numero de bloque de datos
- Estado del buffer
- Puntero al proximo en la hash queue
- Puntero al anterior en la hash queue
- Puntero al proximo en la lista de libres
- Puntero al anterior en la lista de libres
- Puntero al area de datos

El estado del buffer puede ser:

- Lockeado
- Habilitado
- Delay write
- On demand
- Dormido

El Kernel cachea datos de acuerdo a la politica LEAST RECENTLY USED. Mantiene una lista de bloques libres que se rige por esta politica. Esta lista es una lista doblemente enlazada que contiene un buffer dummy para marcar el comienzo y el fin de la misma.

Todos los bloques del buffer cache son puestos en esta lista cuando se bootea el sistema. El Kernel toma un buffer de esta lista cuando desea cualquier buffer libre del buffer cache. Tambien puede tomar un buffer en particular si lo identifica. En ambos casos saca a este buffer de la lista de libres. Cuando el Kernel devuelve un buffer, en general se agrega al final de la lista de buffers libres.

Tambien tiene organizados los buffers en una hash queue. Es decir, cuando el Kernel necesita acceder aun buffer en particular lo identifica por numero de device y numero de bloque. En vez de buscar en todo el buffer cache si el bloque se encuentra en memoria, hashea estos numeros accediendo luego a la hash queue donde se espera encontrar al buffer buscado. Existen varias posibilidades de aquí en mas, dependiendo si encuentra o no el buffer en la cola a la que accede.

Cada buffer siempre existe en una hash queue, sin tener importancia el orden que tengan dentro de cada una. Dos buffers no pueden contener al mismo tiempo los datos del mismo bloque de disco. Sin embargo un mismo buffer puede estar en una hash queue y en la lista de libres.

El algoritmo que realiza la busqueda de un bloque de datos en el buffer cache es **GETBLK** y cubre las siguientes situaciones:

1) Cuando el Kernel busca un bloque en el buffer cache, primero hashea el numero de device donde encuentra al archivo y el numero de bloque que desea acceder. Accede entonces a una hash queue, entra a la misma y busca en forma secuencial el numero de bloque. En este caso el Kernel encuentra el bloque en la hash queue. Se fija si el estado es libre, como lo es, entonces marca el buffer como ocupado y procede a sacarlo de la lista de libres. Devuelve entonces el buffer lockeado.

2) Cuando el Kernel busca un bloque en el buffer cache, primero hashea el numero de device donde encuentra al archivo y el numero de bloque que desea acceder. Accede entonces a la hash queue, entra a la misma, busca en forma secuencial y en este caso no encuentra el buffer en la cola. El buffer no puede estar en otra hash queue, entonces accede a la lista de libres y toma el primer buffer libre que encuentre en ella. El paso siguiente es mover el buffer de la hash queue donde se encuentre y reasignarlo a la hash queue correspondiente. Devuelve entonces el buffer lockeado.

3) Cuando el Kernel busca un bloque en el buffer cache, primero hashea el numero de device donde encuentra al archivo y el numero de bloque que desea acceder. Accede entonces a la hash queue, entra a la misma, busca en forma secuencial y en este caso no encuetra el buffer en la cola. El Kernel accede entonces a la lista de buffers libres y toma el primer buffer de la misma, el mismo tiene como estado "DELAYED WRITE", y en este caso el Kernel procede a solicitar una escritura asincronica del buffer en cuestion, y luego toma el segundo buffer libre de la lista, verificando que el estado sea libre. Si el estado es "DELAYED WRITE" entonces vuelve a pedir la escritura asincronica. El paso siguiente es mover el buffer de la hash queue donde se encuentre y reasignarlo a la hash queue correspondiente. Devuelve entonces el buffer lockeado. Cuando el sistema termina la escritura asincronica, cabe destacar que el buffer es colocado al prinicipio de la lista de libres para evitar que tenga una vuelta "gratis".

4) Cuando el Kernel busca un bloque en el buffer cache, primero hashea el numero de device donde encuentra al archivo y el numero de bloque que desea acceder. Accede entonces a la hash queue, entra a la misma, busca en forma secuencial y en este caso encuentra al buffer pero el estado del mismo es ocupado. En este caso el proceso que quiere acceder al buffer se duerme, es decir, queda esperando que el proceso que actualmente esta utilizando el buffer lo libere. Cuando esto ocurra, se despertaran todos los procesos que esten en estado dormido, para continuar con lo que estaban haciendo. El Kernel en este caso debera chequear que el buffer al cual esta queriendo acceder sea el que realmente busca, en caso de que otro proceso haya podido acceder antes que el al buffer y haya cambiado su contenido.

5) Cuando el Kernel busca un bloque en el buffer cache, primero hashea el numero de device donde encuentra al archivo y el numero de bloque que desea acceder. Accede entonces a la hash queue, entra a la misma, busca en forma secuencial y en este caso no encuentra el buffer. Entonces accede a la lista de buffers libres, pero no hay mas buffers en la lista. El Kernel entonces procede a poner a dormir al estado, esperando que algun proceso libere algun buffer y despierte a los demas procesos. El proceso una vez que se despierta deberà volver a buscar al buffer en cuestion, porque es posible que otro proceso haya asignado un buffer al bloque de datos mientras el proceso que estamos analizando haya estado dormido.

Leyendo y escribiendo bloques del disco.

Cuando se quiere leer un bloque del disco se utiliza el algoritmo getblk para buscarlo primero en el buffer cache, si el algoritmo lo encuentra entonces hemos ahorrado un acceso a disco, de lo contrario el Kernel debera pedir al controlador la lectura de un bloque de disco a memoria.

El algoritmo que realiza esto se llama **BREAD.** Se encarga de pedir la lectura a disco en caso de que el getblk no devuelva un buffer con datos validos (en caso de que no haya encontrado el buffer en la hash queue correspondiente). Tambien existe el algoritmo **BREADA**, que se diferencia del anterior, en el hecho de que se fija si el proximo bloque a leer se encuentra o no en el buffer cache. Si esta entonces continua con la lectura del primer bloque, suponiendo que cuando necesite acceder al segundo, este aun se encontrara en el buffer cache. Si no se encuentra alli, solicita la lectura asincronica del bloque, de manere de ahorrar tiempo en la lectura, y que cuando lo necesite, el bloque ya se encuentre en el buffer cache.

El algoritmo para escribir el contenido de un buffer a disco se llama **BWRITE**. El Kernel informa al controlador que tiene un buffer el cual desea guardar, el controlador programa el bloque para I/O. Hay ocasiones en las que el Kernel no escribe los datos inmediatamente a disco, sino que lo marca como "delayed write" y libera el buffer por medio del algoritmo **BRELSE**. En este caso el Kernel escribe el buffer a disco antes de volver a asignarlo a otro bloque y le pone como estado "OLD", de manere de que cuando termine la escritura el Kernel pueda identificarlo y ponerlo al comienzo de la lista de buffers libres.

El algoritmo **BRELSE** despierta a los procesos dormidos para que continuen con su ejecucion. Si el buffer a liberar contiene datos validos entonces lo coloca al final de la lista de buffers libres, en caso contrario, lo coloca al principio. Temina cuando desbloquea el buffer.

INODOS

Cada archivo dentro del fs se identifica por medio de un inodo. El inodo contiene la siguiente informacion:

- Identificacion del dueño
- Tipo de archivo
- Permisos de acceso
- Tiempos de acceso, modificacion del inodo y el archivo
- Cantidad de links que tiene el archivo
- Punteros a los bloques de datos
- Tamaño del archivo

El Kernel para poder manipular la informacion del inodo, debe levantarlo a memoria, y ademas de esta informacion contiene lo siguiente:

- Estado del inodo
- Numero de device
- Numero de inodo
- Puntero al proximo en la hash queue
- Puntero al anterior en la hash quere
- Puntero al proximo en la lista de libres
- Puntero al anterior en la lista de libres
- Contador de referencias

Asignacion de inodos

Se utiliza el algoritmo **IGET** para asignar una copia en memoria de un inodo dado. El Kernel identifica a un inodo por el device y el numero dentro del array de inodos del fs. Mapea estos numeros en una hash queue y busca dentro de la cola el inodo. Si no lo encuentra, asigna un inodo de la lista de inodos libres.

Para poder leer el inodo a memoria, debe leer el bloque de disco que contenga el inodo, esto lo calcula sabiendo cuantos inodos entran por bloque, y el numero de inodo que se quiere acceder (**BREAD**). Calcula el offset dentro del bloque y lee el inodo, incrementando su contador en uno.

Si no encuentra el inodo en la hash queue correspondiente y no hay inodos en la lista de inodos libres, entonces el algoritmo reporta un error.

Para liberar un inodo se utiliza el algoritmo **IPUT**. Este algoritmo decrementa el contador de referencias del inodo en 1, y si luego de esto el contador queda en 0, el Kernel escribe este inodo a disco (si es que fue modificado). Luego coloca el inodo en la lista de libres

El Kernel procederá a liberar los bloques de datos del archivos en caso de que la cantidad de links al archivo sea 0.

Estructura de un archivo

Como vimos el inodo cuenta con punteros a los bloques de datos fisicos que tenga el archivo. En total son 13 punteros, enumerados del 0 al 12, de los cuales 10 punteros son punteros directos a bloques del disco, los demas son punteros indirectos.

El puntero numero 11 tiene una indireccion simple, apuntando a un bloque de disco que contiene punteros a otros bloques donde estan los datos.

El punteo numero 12 tiene una indireccion doble, apuntando a un bloque de disco que contiene punteros a otros bloques que contienen punteros a otros bloques donde estan los datos.

El puntero numero 13 tiene una indireccion triple, apuntando a un bloque de disco que contiene punteros a otros bloques que contienen punteros a otros bloques que contienen punteros a otros bloques donde estan los datos.

En un principio los 13 punteros estan inicializados en 0, sin ocupar bloques de datos que no contendrian informacion.

De esta forma cuando el Kernel debe acceder a un cierto offset dentro del archivo, debe calcular si el offset estara dentro de un bloque al cual accede en forma directa o a traves de las indirecciones. Esto lo hace a traves del algoritmo **BMAP**.

Directorios

Son archivos que le dan al fs su jerarquia. Sus datos son una serie de entradas que consisten en un numero de inodo y el nombre de un archivo contenido en ese directorio. Cada entrada consta de 16 bytes, 2 bytes para el numero de inodo y 14 bytes para el nombre del archivo.

Algoritmo NAMEI

El Kernel trabaja internamente a traves de numeros de inodos para acceder a los archivos. El algoritmo **NAMEI** parsea el path de un archivo componente a componente convirtiendo cada componente en un inodo y devuelve el inodo correspondiente al archivo en cuestion.

A medida de que va trabajando con los componentes del path va chequeando que el usuario tenga permisos de escritura en los directorios y que el componente sea en si un directorio.

El Kernel hace una busqueda lineal dentro del directorio, tratando de machear el nombre del archivo con las entradas en el mismo. Una vez que encuentra el directorio, lee el inodo correspondiente al componente encontrado y vuelve a buscar. Asi hasta llegar al nombre del archivo, y devuelve el numero de inodo correspondiente a la entrada en el directorio.

El superblock

El file system esta compuesto por:

- Boot Sector
- Superblock
- Lista de inodos
- Bloques del disco

El superblock contiene informacion del file system:

- El tamaño del fs.
- Cantidad de bloques libres en el fs
- Lista de bloques libres
- Indice del proximo bloque libre
- Cantidad de inodos libres en el fs
- Lista de inodos libres
- Indice del proximo inodo libre
- Estado del fs

Asignacion de bloques de discos

El superblock del File System contiene un array que es utilizado para cachear los numeros de bloques libres dentro del fs. El ejecutable MKFS organiza los bloques del disco en una lista linkeada, tal que cada link de la lista es un bloque del disco y contiene un array de numeros de bloques libres, y una de las entradas del array es el numero del proximo bloque linkeado del array.

Cuando el Kernel quiere asignar un bloque del file system, asigna el proximo bloque disponible en la lista del superblock (**ALLOC**). Una vez asignado, no puede ser reasignado hasta que sea liberado.

Si el bloque que lee de la lista es el ultimo bloque libre, entonces toma el contenido, que es un puntero al proximo bloque del link y lee ese bloque, llenando con los datos del mismo la lista de bloques libres del array en el superblock.

Luego trabaja con el numero de bloque que tenia libre y procede a asignarlo, luego de inicializar su contenido.

Cuando se ejecuta el MKFS, se van cargando los numeros de bloques de manera de que tengan cierta relacion que luego pueda mejorar la lectura de los bloques asignados, para aquellos archivos que contendran gran cantidad de bloques de datos.

El algoritmo **FREE** procede de manera inversa. Si la lista en el superbloque no esta llena, entonces el numero del bloque liberado se carga en ella. Si esta llena, entonces el bloque liberado es un nuevo bloque de la lista linkeada, es decir, se copia en el bloque el array del superblock en el bloque y se graba a disco. El array del superblock queda con un elemento apuntando al bloque recien grabado.

Asignacion de inodos a nuevos archivos

El algoritmo **IALLOC** asigna un inodo del disco a un nuevo archivo. El file system contiene una lista de inodos. Un inodo esta libre si el tipo especificado es 0. Cuando el Kernel tiene que asignar un inodo a un archivo nuevo, busca en la lista de inodos libres que tiene en el superblock un numero de inodo.

Si la lista tiene inodos, entonces toma el inodo, asigna una copia en memoria de este inodo por medio del algoritmo **IGET**, inicializa el inodo y lo devuelve lockeado (actualiza este inodo a disco).

Si la lista de inodos esta vacia, el Kernel busca en la lista de inodos del disco y guarda tantos inodos libres como sea posible en el array del superblock, recordando siempre cual es el mayor numero de inodo almacenado.

De esta manera, la proxima vez que necesite levantar numeros de inodos del disco, sabe que a partir del numero recordado tiene que empezar la busqueda. Y cada vez que asigna un inodo a un archivo, decrementa la cantidad de inodos libres en el array dentro del superblock.

Cuando se libera un inodo se utiliza el algoritmo **IFREE**. El Kernel chequea si hay lugar en la lista para el numero de inodo liberado, si hay lugar entonces coloca el nuimero en el array. Si no hay lugar, compara el numero de inodo con el numero del inodo recordado y solo si es menor, lo almacena dentro del array, sacando del array al inodo recordado. Este pasa a ser ahora el inodo recordado.

SYSTEM CALLS PARA EL FILE SYSTEM: SYSTEM V

OPEN

FD = OPEN (PATH DEL ARCHIVO, FLAGS, MODO DE APERTURA)

Es el primer paso cuando queremos acceder a los datos de un archivo. Este system call devuelve un nùmero llamado FILE DESCRIPTOR, que luego sera usado para manipular el archivo.
El Kernel utiliza el algoritmo **NAMEI** para convertir el path del archivo en el inodo correspondiente al mismo. Una vez cargado el inodo en memoria, puede controlar los permisos de acceso al mismo y aloca una entrada en la **FILE TABLE** para el archivo abierto. Esta entrada contiene un puntero al inodo y un offset al byte donde el Kernel pretende hacer la proxima lectura o escritura.
Si el flag utilizado indica lectura, entonces el offset se inicializa en 0, en caso de que indique escritura, se inicializa con un offset igual al tamaño del archivo.
El Kernel tambien aloca una entrada en la **USER FILE DESCRIPTOR TABLE**, tabla que se crea por proceso, y devuelve el ìndice dentro de la tabla que ocupe la nueva entrada, que luego recibe el nombre de **FILE DESCRIPTOR**. Esta entrada en la **USER FILE DESCRIPTOR TABLE** apunta a la entrada generada en la **FILE TABLE** (que es global).
Cada entrada de la user file descriptor table se corresponde con una entrada en la file table, donde se guarda el puntero a la tabla de inodos, el offset dentro del archivo y el modo de apèrtura.

READ

NUMBER = READ (FD, BUFFER, COUNT)

El Kernel toma la entrada de la File Table que corresponde al File descriptor que se le pasa al system call, siguiendo el puntero de la user file descriptor table. Setea parametros en la user area y sigue el puntero de la file table hacia la tabla de inodos, para lockear el inodo antes de leerlo.
El Kernel, para leer, convierte el offset en un bloque de datos, usando el algoritmo **BMAP**, y anota los resultados. Lee el bloque en el buffer, utilizando el algoritmo **BREAD** o **BREADA** y luego copia los datos al área que se le haya indicado al system call. Actualiza los datos para llevar un control de cuanto falta para leer, donde debe continuar, etc.
Una vez que satisfizo el pedido, actualiza el offset en el file table y devuelve el total de bytes leidos. Por último se deslockea el inodo al final del system call.

WRITE

NUMBER = WRITE (FD, BUFFER, COUNT)

Es muy similar al READ, con la diferencia de que si el offset en el cual se quiere escribir no corresponde a un bloque del archivo, se aloca uno nuevo por medio del algoritmo **ALLOC** (o si corresponde, aloca una serie de bloques, en caso de que el direccionamiento sea indirecto). Se lockea el inodo del archivo mientras dure el system call.
Cuando la escritura se completa, se actualiza el tamaño del archivo en el inodo.

Este system call procede bloque a bloque decidiendo que bloque grabar y cual dejar como *delayed write* para mejorar la performance de lectura si el bloque es mas probable de ser leido.

CLOSE

CLOSE (FD)

Este system call manipula el file descriptor, la file table y la tabla de inodos correspondientes al archivo.

Si el contador en la file table es mayor a uno, significa que otros procesos estan haciendo referencia a la entrada, entonces se decrementa el contador y el close termina.

Si el contador es 1 entonces se libera la entrada en la file table y se libera al inodo de la tabla de inodos a traves del algoritmo **IPUT**.

Si el contador de referencias de la tabla de inodos es mayor a 1 entonces solo se decrementa el contador.

Luego de esto se libera la entrada del file descriptor en la user file descriptor table.

CREAT

FD = CREAT (PATH DEL ARCHIVO, MODES)

El Kernel parsea el path que recibe usando el algoritmo **NAMEI**, para poder ubicar el inodo del directorio padre. Debe verificar la existencia o no del archivo que se esta intentando crear, de manera de crearlo o resetear los bloques que este este utilizando y dejarlo como si estuviese recien creado. Cuando llega al ultimo componente del path, este algoritmo recuerda el offset del primer slot libre dento del directorio padre.

Si el Kernel no encuentra el nombre del archivo en el directorio, entonces ya tiene el offset donde debera escribir el nuevo archivo junto con su inodo correspondiente.

Si no encuentra efectivamente el nombre del archivo, el Kernel le asigna un inodo, usando el algoritmo **IALLOC.**

Luego escribe el nombre del archivo y el numero de inodo en el directorio padre, el en offset que guardo anteriormente, y libera el inodo del directorio padre. Luego escribe el nuevo inodo a disco, por medio del algoritmo **BWRITE** y finalmente escribe el directorio a disco.

Si el archivo ya existia, el Kernel encuentra el inodo del mismo en la primer busqueda, libera los bloques del mismo por medio del algoritmo **FREE**. Modifica el tamaño del archivo a 0 bytes, y deja los permisos y autorias iguales.

De igual manera que el system call **OPEN**, se aloca una entrada el la file table y una entrada en la user file descriptor table, y se devuelve el indice de la tabla como file descriptor.

MKNOD

MKNOD (PATHNAME, TYPE AND PERMISSIONS, DEV)

Crea archivos especiales en el sistema, incluyendo pipes, devices y directorios. Es similar al system call CREAT, en el sentido de que aloca un inodo para el archivo. En el caso de que el archivo a crear sea un directorio, el directorio existe luego de la llamada al system call, pero el formato no es el correcto.

PIPES

Pipes permiten la transferencia de datos entre procesos en modo **FIFO** (first in first out), y la sincronizacion de ejecucion de procesos. Permite la comunicación de los procesos aun sin que un proceso sepa quien esta del otro lado del pipe.

Hay dos tipos de pipes, **named pipes** y unnamed pipes. La unica diferencia entre ambos es la manera que tienen los procesos de acceder inicialmente a los ultimos.

Los procesos usan el OPEN para los **named pipes**, y el **PIPE** para crear los **unnamed pipes**. Solamente los procesos descendientes del proceso que utilizo PIPE, pueden acceder al unnamed pipe. Sin embargo, cualquier proceso puede acceder a un named pipe.

PIPE

PIPE (FDPTR)

Permite la creacion de un unnamed pipe. FDPTR es un puntero a un array de enteros que contendra los dos file descriptors para lectura y escritura del pipe. El Kernel asigna un inodo para el pipe (**IALLOC**) de un device especial para pipes y aloca dos entradas en la file table y user file descriptor table.

Un named pipe es un archivo cuya semanticas son las misma que los unnamed pipes, excepto que tiene una entrada de directorio y es accedido por medio de un path. Los procesos acceden al pipe igual que a otros archivos,

Los named pipes existen permanentemente en el file system, los unnamed pipes son transitorios (cuando los procesos terminan de usarlo, el Kernel libera el espacio asignado).

DUP

NEWFD = DUP (FD)

Copia un file descriptor en el primer slot libre de la user file descriptor table, devolviendo el nuevo file descriptor al usuario. Trabaja con todos los tipos de archivos.

Como el dup duplica el fd, incrementa el contador de la correspondiente entrada en la file table. Notar que ambos apuntan a la misma entrada, con un solo offset en el archivo.

DUP2

int dup2(int oldfd, int newfd)

El descriptor antiguo es cerrado con dup2().
Con esta particular llamada, tenemos la operacion de cerrado, y la duplicacion del descriptor actual, relacionado con una llamada al sistema.

Ademas, se garantiza el ser atomica, que esencialmente significa que nunca se interrumpira por la llegada de una señal. Toda la operacion transcurrira antes de devolverle el control al nucleo para despachar la señal. Con la llamada al sistema dup() original, los programadores tenian que ejecutar un close() antes de llamarla.

Esto resultaba de dos llamadas del sistema, con un grado pequeño de vulnerabilidad en el breve tiempo que transcurre entre ellas. Si llega una señal durante ese tiempo, la duplicacion del descriptor fallaria. Por supuesto, dup2 () resuelve este problema para nosotros.

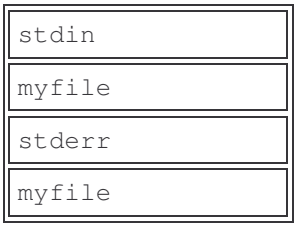
Esto es de otro apunte:

> **dup**() encuentra la primera entrada libre en la tabla de descriptores y pone su apuntador a apuntar al mismo lugar al que apunta fd. A la misma entrada en la tabla de archivos abiertos.
>
> **dup2(int oldfd, int newfd):** cierra newfd si está activo y luego pone a apuntar su entrada a donde apunta oldfd; a la misma entrada en la tabla de archivos abiertos. Es equivalente a:
>
> close(newfd)
> dup(oldfd)
>
> **dup** y **dup2** se usa para redireccionar la entrada y la salida estándar
>
> Ejemplo:

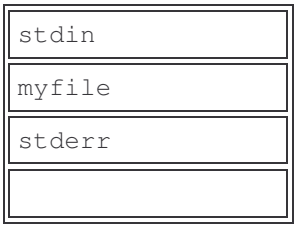
```
write(1, buff, 20)
fd = open("myfile", O_WRONLY|O_CREAT, 066)
dup2(fd, 1)
close(fd)
write(1, buff, 20)
```

Después del open:

Después del dup2:



Después del close:



LINK

LINK (FOURCE FILE NAME, TARGET FILE NAME)

Este system call linkea un archivo a un nuevo nombre en el file system, creando una nueva entrada de directorio para el nodo existente.
El file system contiene un path para cada link que el archivo tenga, un proceso puede acceder al archivo por medio de cualquiera de estos paths. El Kernel no identifica al nombre original del archivo, por consecuencia, todos los paths son tratados en forma equitativa.
Solamente el usuario superuser tiene permiso para linkear directorios, de manera de evitar un loop en los links.

El Kernel utiliza el **NAMEI** para localizar el inodo del archivo fuente, e incrementa el contador de links, actualiza el inodo a disco. Luego utiliza el mismo algoritmo para ubicar el archivo destino. Si lo encuentra, entonces el LINK falla, sino, busca un slot libre en el directorio padre, escribe el nombre del archivo y el # de inodo del archivo fuente, finalmente libera el inodo por medio del IPUT.

UNLINK

UNLINK (PATHNAME)

Remueve una entrada de directorio para un archivo. Si el path al cual se aplica el unlink es el ultimo link al archivo, el Kernel libera los bloques de datos. Si no es el ultimo, solo baja la cantidad de links al archivo.
El Kernel utiliza una variante del algoritmo **NAMEI** para ubicar el archivo a modificar. En vez de devolver el inodo del archivo, devuelve el inodo del directorio padre. Utiliza el **IGET** para subirlo a memoria y luego escribe un 0 en el dato de # de inodo del archivo. El Kernel graba el directorio a disco, decrementa el contador de links y por medio del **IPUT** libera los inodos del directorio padre y del archivo.
Si al hacer el **IPUT** el contador de links es 0, se procede a liberar los inodos y bloques de datos del archivo.

Consistencia del File System

Cuando el File system remueve un archivo de su directorio padre, realiza una escritura sincronica a disco del directorio, antes de liberar el inodo y los bloques correspondientes al archivo en cuestion.

Si el File System sufre algun problema antes de remover el contenido del archivo, los daños serian minimos: el inodo del archivo tendria un contador de links mayor al actual, pero el los demas paths serian legales.

Si la escritura del directorio no fuese sincronica, podria ser que la entrada en el directorio apuntase a un inodo libre o realocado despues del fallo del sistema.

El Kernel tambien libera los inodos y bloques de disco en un orden especial, Cuando libera el contenido de un archivo.

Supongamos que el Kernel libera los bloques del archivo primero, y luego sufre un fallo. Cuando el sistema rebootea, el inodo contiene referencias a los bloques liberados, los cuales pudieron haber sido asignados a otro archivo.

Es por eso que primero modifica el inodo del archivo, lo graba a disco y luego trabaja liberando a los bloques de disco.

Race Conditions

Los race conditions pueden presentarse en system calls como UNLINK, por ejemplo cuando se ejecuta el comando RMDIR, para eliminar un directorio, antes de eliminarlo, el Kernel debe asegurarse de que este vacio, es decir, que todos los paths que contenga, tengan como # de inodo un 0. Para esto debe leer el bloque que contiene los datos del directorio.

Puede ser entonces que cuando el comando RMDIR termino de verificar que todas referencias son al inodo 0, un CREAT cree un archivo en este directorio.

Para evitar esto, se deben utilizar lockeos en los inodos y los bloques de los archivos. Una vez que un proceso comenzo a ejecutar el UNLINK, ningun otro proceso puede acceder al archivo, ya que los inodos del directorio padre y del archivo en si estan lockeados.