Tiposdatos.h

Escribir esta estructura con los datos en el sector de arranque.

struct SECBOOT {

char jump[4];

char nombre\_disco[8];

unsigned char sec\_res; // 1 sector de arranque

unsigned char sec\_mapa\_bits\_nodos\_i; // 1 sector

unsigned char sec\_mapa\_bits\_bloques; // 4 sectores

unsigned short sec\_tabla\_nodos\_i; // 1 sector

unsigned short sec\_log\_unidad; // 54400 sectores

unsigned char sec\_x\_bloque; // 4 sectores por bloque

unsigned char heads; // 20 superficies

unsigned char cyls; // 160 cilindros

unsigned char secfis; // 17 sectores

char restante[487];

};

Formatear el disco consiste en:

1. Escribir el sector de arranque con los parámetros del disco en el sector físico 1, cilindro 0, superficie 0.
2. Reiniciar los mapas de bits (todos en 0)
3. Reiniciar la tabla de nodos i.

Estructura de datos que vamos a usar para los nodos i.

struct INODE {

char name[20];

unsigned short uid;

unsigned short gid;

unsigned short perms;

unsigned int datetimecreat;

unsigned int datetimemodif;

unsigned int size;

unsigned short blocks[10];

unsigned short indirect;

unsigned short indirect2;

};

Tabla de archivos abiertos

struct OPENFILES {

int inuse;

unsigned short inode;

int currpos;

int currbloqueenmemoria;

char buffer[2048];

unsigned **short** buffindirect[1024];

};

Funciones para el manejo de los mapas de bits (area de datos)

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Para el mapa de bits del área de de datos

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int isblockfree(int block)

{

int offset=block/8; // Obtener en que byte

int shift=block%8; // y bit, que indica si el bloque está libte

int result;

int i;

// Es importante tenerlo en memoria porque ahí es donde tenemos

// la información del disco, cuáles son los sectores donde hay qué

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

// Calculo el sector donde está el mapa de bits para los bloques

mapa\_bits\_bloques= secboot.sec\_res+secboot.sec\_mapa\_bits\_nodos\_i;

// ¿Está en memoria el mapa de bits de bloques?, si no, cargarlo a // memoria

if(!blocksmap\_en\_memoria)

{

// Leer todos los sectores del mapa de bits a memoria

for(i=0;i<secboot.sec\_mapa\_bits\_bloques;i++)

result=**vdreadseclog**(mapa\_bits\_bloques+i,blocksmap+i\*512);

blocksmap\_en\_memoria=1;

}

if(blocksmap[offset] & (1<<shift))

return(0);

else

return(1);

}

// Buscar en el mapa de bits cuál es el siguiente bloque libre

int nextfreeblock()

{

int i,j;

int result;

// Ya quedó explicado arriba

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

mapa\_bits\_bloques = secboot.sec\_res+secboot.sec\_mapa\_bits\_nodos\_i;

if(!blocksmap\_en\_memoria)

{

for(i=0;i<secboot.sec\_mapa\_bits\_bloques;i++)

result=vdreadseclog(mapa\_bits\_bloques+i,blocksmap+i\*512);

blocksmap\_en\_memoria=1;

}

// Buscar el primer byte en el mapa de bloques donde hay al menos un

// bloque libre

i=0;

while(blocksmap[i]==0xFF && i<secboot.sec\_mapa\_bits\_bloques\*512)

i++;

// Si no llegamos al final

if(i<secboot.sec\_mapa\_bits\_bloques\*512)

{

j=0;

while(blocksmap[i] & (1<<j) && j<8)

j++;

return(i\*8+j); // Regresando cual es el primer bloque libre encontrado

}

else

return(-1); // Llegamos al final del mapa y no encontramos

// un bloque libre

}

Poner un bloque como no disponible

int assignblock(int block)

{

int offset=block/8;

int shift=block%8;

int result;

int i;

int offset;

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

mapa\_bits\_bloques= secboot.sec\_res+secboot.sec\_mapa\_bits\_nodos\_i;

if(!blocksmap\_en\_memoria)

{

for(i=0;i<secboot.sec\_mapa\_bits\_bloques;i++)

result=vdreadseclog(mapa\_bits\_bloques+i,blocksmap+i\*512);

blocksmap\_en\_memoria=1;

}

// Poner en 1 el bit en el byte que corresponde al número de bloque

blocksmap[offset]|=(1<<shift);

// Escribir ese sector en el disco

offset=(offset/512)\*512;

// 0 .. 511 = 0

// 512 .. 1023 = 512

// 1024 .. 1535 = 1024

// offset=offset-(offset%512);

vdwriteseclog(mapa\_bits\_bloques+offset,blocksmap+sector\*512);

return(1);

}

Establecer el bloque como libre

int unassignblock(int block)

{

int offset=block/8;

int shift=block%8;

int result;

char mask;

int sector;

int i;

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

mapa\_bits\_bloques= secboot.sec\_res+secboot.sec\_mapa\_bits\_nodos\_i;

if(!blocksmap\_en\_memoria)

{

for(i=0;i<secboot.sec\_mapa\_bits\_bloques;i++)

result=vdreadseclog(mapa\_bits\_bloques+i,blocksmap+i\*512);

blocksmap\_en\_memoria=1;

}

blocksmap[offset]&=(char) **~**(1<<shift);

sector=(offset/512)\*512;

vdwriteseclog(mapa\_bits\_bloques+sector,blocksmap+sector\*512);

return(1);

}

Funciones para manipular los mapas de bits de los nodos i

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Para el mapa de bits del área de nodos i

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int isinodefree(int inode)

{

int offset=inode/8;

int shift=inode%8;

int result;

// Checar si el sector de boot está en memoria

if(!secboot\_en\_memoria)

{

// Si no está en memoria, cárgalo

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

mapa\_bits\_nodos\_i= secboot.sec\_res; //Usamos la información del sector de boot para

//determinar en que sector inicia el

// mapa de bits de nodos i

// Está el mapa está en memoria

if(!inodesmap\_en\_memoria)

{

// Si no está en memoria, hay que leerlo del disco

result=vdreadseclog(mapa\_bits\_nodos\_i,inodesmap);

inodesmap\_en\_memoria=1;

}

if(inodesmap[offset] & (1<<shift))

return(0);

else

return(1);

}

int nextfreeinode()

{

int i,j;

int result;

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

mapa\_bits\_nodos\_i= secboot.sec\_res;

if(!inodesmap\_en\_memoria)

{

result=vdreadseclog(mapa\_bits\_nodos\_i,inodesmap);

inodesmap\_en\_memoria=1;

}

// Recorrer byte por byte mientras sea 0xFF sigo recorriendo

i=0;

while(inodesmap[i]==0xFF && i<secboot.sec\_mapa\_bits\_nodos\_i)

i++;

if(i<secboot.sec\_mapa\_bits\_nodos\_i)

{

j=0;

while(inodesmap[i] & (1<<j) && j<8)

j++;

return(i\*8+j);

}

else

return(-1);

}

int assigninode(int inode)

{

int offset=inode/8;

int shift=inode%8;

int result;

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

mapa\_bits\_nodos\_i= secboot.sec\_res;

if(!inodesmap\_en\_memoria)

{

result=vdreadseclog(mapa\_bits\_nodos\_i,inodesmap);

inodesmap\_en\_memoria=1;

}

inodesmap[offset]|=(1<<shift);

vdwriteseclog(mapa\_bits\_nodos\_i,inodesmap);

return(1);

}

int unassigninode(int inode)

{

int offset=inode/8;

int shift=inode%8;

int result;

char mask;

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

mapa\_bits\_nodos\_i= secboot.sec\_res;

if(!inodesmap\_en\_memoria)

{

result=vdreadseclog(mapa\_bits\_nodos\_i,inodesmap);

inodesmap\_en\_memoria=1;

}

inodesmap[offset]&=(char) ~(1<<shift);

vdwriteseclog(mapa\_bits\_nodos\_i,inodesmap);

return(1);

}

Lectura y escritura de bloques

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Lectura y escritura de bloques

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int writeblock(int block,char \*buffer)

{

int result;

int i;

// Verificar si el sector de boot está en memoria y si no, cárgalo

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

// Calcula cuál es el sector donde inician los bloques

inicio\_area\_datos=secboot.sec\_res+secboot.sec\_mapa\_bits\_nodos\_i +secboot.sec\_mapa\_bits\_bloques+secboot.sec\_tabla\_nodos\_i;

// Escribir todos los sectores lógicos que corresponden al bloque

for(i=0;i<secboot.sec\_x\_bloque;i++)

vdwriteseclog(inicio\_area\_datos+(block-1)\*secboot.sec\_x\_bloque+i,buffer+512\*i);

// Escribimos cada uno de los sectores que corresponden al bloque

// buffer+512\*i hacemos que cada iteración del for un subbuffer de 512

// bytes.

return(1);

}

int readblock(int block,char \*buffer)

{

int result;

int i;

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

inicio\_area\_datos=secboot.sec\_res+secboot.sec\_mapa\_bits\_nodos\_i+secboot.sec\_mapa\_bits\_bloques+secboot.sec\_tabla\_nodos\_i;

for(i=0;i<secboot.sec\_x\_bloque;i++)

vdreadseclog(inicio\_area\_datos+(block-1)\*secboot.sec\_x\_bloque+i,buffer+512\*i);

return(1);

}

Funciones para el empaquetamiento de fecha y hora en los archivos

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Funciones para el manejo de inodos

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Convierte la fecha que está en una estructura fecha a un entero de 32 bits

unsigned int datetoint(struct DATE date)

{

unsigned int val=0;

val=date.year-1970;

val<<=4;

val|=date.month;

val<<=5;

val|=date.day;

val<<=5;

val|=date.hour;

val<<=6;

val|=date.min;

val<<=6;

val|=date.sec;

return(val);

}

// Extraer la fecha y hora que está empaquetada en un entero de 32 bits

// Almacena los resultados en una estructura

int inttodate(struct DATE \*date,unsigned int val)

{

date->sec=val&0x3F;

val>>=6;

date->min=val&0x3F;

val>>=6;

date->hour=val&0x1F;

val>>=5;

date->day=val&0x1F;

val>>=5;

date->month=val&0x0F;

val>>=4;

date->year=(val&0x3F) + 1970;

return(1);

}

// Fecha y hora actual empaquetada en un entero de 32 bits

// Las funciones de creación y escritura de archivos usan esta función.

unsigned int currdatetimetoint()

{

struct tm \*tm\_ptr;

time\_t the\_time;

struct DATE now;

(void) time(&the\_time);

tm\_ptr=gmtime(&the\_time);

now.year=tm\_ptr->tm\_year-70;

now.month=tm\_ptr->tm\_mon+1;

now.day=tm\_ptr->tm\_mday;

now.hour=tm\_ptr->tm\_hour;

now.min=tm\_ptr->tm\_min;

now.sec=tm\_ptr->tm\_sec;

return(datetoint(now));

}

Una parte fundamental para gestionar los archivos e implementar las funciones de creación, borrado y búsqueda en el directorio son las funciones que nos sirven para buscar, eliminar o crear inodos en el directorio raíz.

// A partir de una posición en el archivo determina la dirección de memoria

// donde está el apuntador en el nodo i que está cargado en memoria.

unsigned short \*postoptr(int **fd**,int pos)

{

int currinode;

unsigned short \*currptr;

unsigned short indirect1;

// El número de inodo actual lo obtenemos de la tabla de archivos abiertos

currinode=openfiles[**fd**].inode;

// Está en los primeros 10 K

if((pos/2048)<10)

// Está entre los 10 apuntadores directos

currptr=&inode[currinode].blocks[pos/2048];

else if((pos/2048)<1034)

{

// Si el apuntador a bloque indirecto está vacío, asígnale un bloque

if(inode[currinode].indirect==0)

{

// El siguiente bloque que disponible de acuerdo al mapa de bits

indirect1=nextfreeblock();

assignblock(indirect1); // Asígnalo

inode[currinode].**indirect**=indirect1;

}

// En la tabla de archivos abiertos tenemos el buffer que almacena

// el bloque de apuntadores

currptr=&openfiles[fd].buffindirect[pos/2048-10];

}

else

return(NULL);

return(currptr);

}

unsigned short \*currpostoptr(int fd)

{

unsigned short \*currptr;

currptr=postoptr(fd,**openfiles[fd].currpos**);

return(currptr);

}

Funciones básicas para la implementación del sistema de archivos

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Funciones del sistema de archivos

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Tabla de archivos abiertos

int openfiles\_inicializada=0;

struct OPENFILES openfiles[16];

// En un inodo específico escribe el nombre, atributos y usuario y dueño

// del nuevo archivo.

// Usada por la función vdcreate

int setninode(int num, char \*filename,unsigned short atribs, int uid, int gid)

{

int i;

int result;

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

inicio\_nodos\_i=secboot.sec\_res+secboot.sec\_mapa\_bits\_nodos\_i+secboot.sec\_mapa\_bits\_bloques;

if(!nodos\_i\_en\_memoria)

{

for(i=0;i<secboot.sec\_tabla\_nodos\_i;i++)

result=vdreadseclog(inicio\_nodos\_i+i,&inode[i\*8]);

nodos\_i\_en\_memoria=1;

}

strncpy(inode[num].name,filename,20);

if(strlen(inode[num].name)>19)

inode[num].name[19]='\0';

inode[num].datetimecreat=currdatetimetoint();

inode[num].datetimemodif=currdatetimetoint();

inode[num].uid=uid;

inode[num].gid=gid;

inode[num].perms=atribs;

inode[num].size=0;

for(i=0;i<10;i++)

inode[num].blocks[i]=0;

inode[num].indirect=0;

inode[num].indirect2=0;

// Optimizar la escritura escribiendo solo el sector lógico que

// corresponde al inodo que estamos asignando.

// i=num/8;

// result=vdwriteseclog(inicio\_nodos\_i+i,&inode[i\*8]);

for(i=0;i<secboot.sec\_tabla\_nodos\_i;i++)

result=vdwriteseclog(inicio\_nodos\_i+i,&inode[i\*8]);

return(num);

}

// Buscar a partir del nombre del archivo el número de inodo correspondiente

int searchinode(char \*filename)

{

int i;

int free;

int result;

if(!secboot\_en\_memoria)

{

result=vdreadsector(0,0,0,1,1,(char \*) &secboot);

secboot\_en\_memoria=1;

}

inicio\_nodos\_i=secboot.sec\_res+secboot.sec\_mapa\_bits\_nodos\_i+secboot.sec\_mapa\_bits\_bloques;

if(!nodos\_i\_en\_memoria)

{

for(i=0;i<secboot.sec\_tabla\_nodos\_i;i++)

result=vdreadseclog(inicio\_nodos\_i+i,&inode[i\*8]);

nodos\_i\_en\_memoria=1;

}

if(strlen(filename)>19)

filename[19]='\0';

i=0;

while(strcmp(inode[i].name,filename) && i<8)

i++;

if(i>=8)

return(-1);

else

return(i);

}

// Elimina un inodo de la tabla de nodos i

// Util para borrar un archivo

int removeinode(int numinode)

{

int i;

unsigned short temp[1024];

for(i=0;i<10;i++)

if(inode[numinode].blocks[i]!=0)

unassignblock(inode[numinode].blocks[i]);

if(inode[numinode].indirect!=0)

{

// Leer el bloque

readblock(inode[numinode].indirect,(char \*) temp);

for(i=0;i<1024;i++)

if(temp[i]!=0)

unassignblock(temp[i]);

unassignblock(inode[numinode].indirect);

inode[numinode].indirect=0;

}

unassigninode(numinode);

return(1);

}

Funciones de alto nivel.

Posicionamiento del apuntador del archivo.

// fd = Descriptor del archivo

// Offset = Cuanto hay que mover el apuntador

// Whence = A partir de dond

// 0 = A partir del inicio.

// 1 = A partir de la posición actual del puntero.

// 2 = A partir del final del archivo.

//

// devuelve la posición donde queda el puntero del archivo después de moverlo.

int vdseek(int fd, int offset, int whence)

{

unsigned short oldblock,newblock;

// Si no está abierto regresa error

if(**openfiles[fd].inuse**==0)

return(-1);

oldblock=\*currpostoptr(fd);

// A partir de la posición actual del archivo, me regresa un apuntador

// a una dirección de memoria que contiene el bloque actual.

// Puede ser el inodo o un bloque de apuntadores.

if(whence==0) // A partir del inicio

{

// Si el offset es negativo o excede el tamaño del archivo regresa

// error

if(offset<0 ||

openfiles[fd].currpos+offset>inode[openfiles[fd].inode].size)

return(-1);

openfiles[fd].currpos=offset;

} else if(whence==1) // A partir de la posición actual

{

// Validar que posición actual - offset no vaya antes del principio

// también que posición actual + offset no exceda el tamaño del archivo

if(openfiles[fd].currpos+offset>inode[openfiles[fd].inode].size ||

openfiles[fd].currpos+offset<0)

return(-1);

openfiles[fd].currpos+=offset;

} else if(whence==2) // A partir del final

{

// Validar que no sea positivo

// Si es negativo, que el valor absoluto no exceda el

// tamaño del archivo

if(offset>inode[openfiles[fd].inode].size ||

openfiles[fd].currpos-offset<0)

return(-1);

openfiles[fd].currpos=inode[openfiles[fd].inode].size-offset;

} else

return(-1);

// Una vez cambiada la posición, obtenemos el nuevo bloque actual

newblock=\*currpostoptr(fd);

// Si después del movimiento hay un cambio de bloque

if(newblock!=oldblock)

{

// Escribir el bloque que ya no estoy usando

// Si quieren en la tabla de archivos abiertos poner una bandera que

// indica si el bloque fue modificado y si es así escríbelo

// Una vez que se escribe, poner esa bandera en 0.

writeblock(oldblock,openfiles[fd].buffer);

// Cargar el nuevo bloque a memoria

readblock(newblock,openfiles[fd].buffer);

openfiles[fd].currbloqueenmemoria=newblock;

}

return(openfiles[fd].currpos);

}

Escritura.

// fd es el identificador en la tabla de archivos abiertos

// buffer un apuntador al area de memoria donde está lo que vamos a escribir

// bytes cuántos bytes vamos a escribir

int vdwrite(int fd, char \*buffer, int bytes)

{

int currblock;

int currinode;

int cont=0;

int sector;

int i;

int result;

unsigned short \*currptr;

// Si no está abierto, regresa error

if(openfiles[fd].inuse==0)

return(-1);

// Determinar cuál es el inodo de la tabla de nodos i del archivo que

// vamos a escribir

currinode=openfiles[fd].inode;

// Ciclo para recorrer byte por byte del buffer a escribir

while(cont<bytes)

{

// Obtener la dirección de donde está el bloque que corresponde

// a la posición actual, si no hay bloque asignado a la posición

// actual del archivo, regresamos error

currptr=**currpostoptr(fd)**;

if(currptr==NULL)

return(-1);

// Obtener el número de bloque actual

currblock=\*currptr;

// Si el bloque está en 0, aún no hay bloque para

// escribir este carácter del archivo

// hay que darle uno

if(currblock==0)

{

// Busca un bloque libre en el mapa de bits

// si no hay bloques libres, regresa -1

currblock=nextfreeblock();

if(currblock==-1)

return(-1);

// El bloque encontrado ponerlo en donde

// apunta el apuntador al bloque actual

**\*currptr**=currblock; // aquí ya lo estoy escribiendo en el nodo i.

assignblock(currblock); // Poner ese bloque como asignado

// Los cambios se hicieron en el nodo i en memoria, ahora

// hay que escribirlos en el disco

// Escribir el sector de la tabla de nodos i

// En el disco

// Recordar que los nodos i son de 64 bytes y caben 8 en

// un sector

sector=(currinode/8)\*8;

result=vdwriteseclog(inicio\_nodos\_i+sector,&inode[sector\*8]);

}

// Si el bloque de la posición actual no está en memoria

// Lee el bloque al buffer del archivo

if(openfiles[fd].currbloqueenmemoria!=currblock)

{

// Leer el bloque actual hacia el buffer que

// está en la tabla de archivos abiertos

readblock(currblock,openfiles[fd].buffer);

openfiles[fd].currbloqueenmemoria=currblock;

}

// Copia el caracter del buffer que recibe la función

// vdwrite, al buffer donde tenemos el bloque actual

openfiles[fd].buffer[openfiles[fd].currpos%2048]=buffer[cont];

// Incrementa posición actual del archivo

openfiles[fd].currpos++;

// Si la posición es mayor que el tamaño, modifica el tamaño

if(openfiles[fd].currpos>inode[currinode].size)

inode[openfiles[fd].inode].size=openfiles[fd].currpos;

// Incrementa el contador

cont++;

// Si se llena el buffer, escríbelo

if(openfiles[fd].currpos%2048==0)

writeblock(currblock,openfiles[fd].buffer);

}

return(cont);

}