21-11621
DON

tesis
KFN - 3931

CINVESTAV-IPN
Biblioteca de Ingeniería Eléctrica

FB0000008701

Rojo

CENTRO DE INVESTIGACION Y ESTUDIOS AVANZADOS DEL

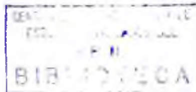
INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERIA ELECTRICA

SECCION COMPUTACION

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

SISTEMA DE ARCHIVOS PARA EL SISTEMA OPERATIVO XINIX



Tesis que presenta la Ing. Ruth Elizabeth Delgado Moreno para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de INGENIERIA ELECTRICA con opción en COMPUTACION.

Trabajo dirigido por el M. en C. Jorge Buenabad Chávez.

Becario de CONACYT México D.F. Abril 1990.

XM
CLASIF.: 90.18
ADQUIS.: 81-11687
FECHA: 10-VII-90
PROCED.: DOJ
$

AGRADECIMIENTOS

Al M. en C. Jorge Buenabad por su dirección, apoyo y paciencia en el desarrollo de este trabajo.

Al Dr. Manuel Guzmán Rentería por su apoyo,
Al Dr. Jan Janecek por sus valiosos comentarios,
y a ambos el tiempo dedicado a la revisión de este trabajo.

A todos mis amigos y compañeros por su grata compañía, y en especial a: Hildelisa Preciado, Octavio Juárez, Vicki Sailer, Zandra Navarro, Marco Rocha, Hirán Cruz.

Al CINVESTAV, y en particular a la Sección de Computación con cuyos recursos se realizó este trabajo.

Al CONACYT por financiarme durante la maestría.

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

A mis padres María de la Luz y Pascual

A mis hermanas Claudia, Judith, Lilian

A la memoria de Don Ramón Gómez

INDICE

Introducción.

Capítulo 1 Sistema Operativo XINIX.

1.1 Características Generales 1

1.2 Organización Lógica 3

1.3 Tipos de Servicios 5

1.4 Manejadores de Dispositivos 7

Capítulo 2 Sistema de Archivos Original de XINIX.

2.1 Organización Física 9

2.2 Organización Lógica 12

2.3 Manejo de Archivos 14

2.4 Comandos del Interprete sobre archivos. . 16

2.5 Uso de servicios del S.A. en programas de usuario 17

Capítulo 3 Características del nuevo Sistema de Archivos para XINIX.

3.1 Organización Física 19

3.2 Organización Lógica 20

3.2.1 Buffer Cache. 20

3.2.2 Nodos Indice (nodos-i) 24

3.2.3 Superbloques. 30

3.3.3 Comparación del sistema de archivos . . 33 original (SAO) con el nuevo sistema de archivos (NSAX).

Capítulo 4 Modificación de XINIX para la implantación del Sistema de Archivos.

4.1 Cuando XINIX inicia 37

4.2 Manejo de procesos 38

4.3 La tabla de dispositivos 41

4.4 La Interfaz del Sistema de Archivos. . . . 42
4.5 Manejador de terminal y disco. 45
4.6 Breve descripción funcional de los servicios del NSAX. 46

Apéndice A. Uso de los servicios del SA en programas de usuario 52
Apéndice B. Uso de comandos en el Intérprete 66
Apéndice C. Código de errores, significados 70
Referencias 73

Introducción

El Sistema Operativo XINIX fue desarrollado como trabajo de tesis en la Sección de Computación del CINVESTAV_IPN. Su estructura básica fue tomada del sistema operativo XINU. XINU corre en una computadora LSI-11 de DEC, motivo por el cual, se propuso transportarlo a la IBM PC-XT, incluyendo en dicha transportación, partes de los manejadores de terminal y de disco de otro sistema operativo, MINIX. EL resultado fue la primera versión de XINIX. Cabe mencionar que tanto XINU como MINIX fueron diseñados para la enseñanza de sistemas operativos. XINIX tiene también ese propósito, pues se ha utilizado para los cursos de Sistemas Operativos y de Programación en tiempo real en la mencionada sección.

El medio ambiente que provee XINIX en una computadora IBM PC-XT y dos terminales RS-232, es muy adecuado para el desarrollo de tareas que requieren el enfoque de multiprogramación, excepto cuando éstas requerían mantener información organizada en diskette (subdirectorios), pues XINIX sólo proveía una estructura plana de archivos, y con una deficiencia adicional: sólo se permitían hasta 28 archivos en cada diskette. Es por ello que surgió la propuesta de implantar una nueva versión de XINIX, en la que el Sistema de Archivos proporcionara una estructura más flexible, y facilidades para la organización de información en disco, según los requerimientos de los usuarios. El Nuevo Sistema de Archivos (NSAX) permite tener una cantidad de archivos limitada sólo por la capacidad de almacenamiento en un diskette.

En los cuatro capítulos que forman este trabajo, esta implícita

la referencia a la tesis "XINIX sistema operativo para computadora personal".

En el capitulo 1 se da un breve panorama acerca del Sistema Operativo XINIX, sus caracteristicas generales, su organización lógica, tipos de servicios.

En el capitulo 2 se explica el sistema de archivos original de XINIX, su organización fisica y lógica, el manejo de los archivos, y los servicios que prestaba.

En el capítulo 3 se da una explicación acerca de las estructuras básicas sobre las que se construye el Nuevo Sistema de Archivos, su organización física y lógica, la interrelación entre ellas.

En el capítulo 4 se presentan las modificaciones hechas al resto de XINIX para poder implantar el sistema de archivos, la comunicación entre XINIX, el sistema de archivos y el usuario, así como una breve descripción del funcionamiento de cada servicio.

Capítulo 1

Sistema Operativo XINIX

Un sistema operativo tiene las siguientes funciones:

- Proveer una interfaz para el usuario, que oculta los detalles del manejo del hardware.
- Ejecutar los programas de los usuarios.
- Administrar los recursos de la computadora: CPU, memoria, dispositivos de E/S, sistema de archivos.

1.1 Características Generales de XINIX.

XINIX es un sistema operativo de tiempo compartido hecho para una computadora PC-XT, a la que pueden conectarse 2 terminales através de puertos RS-232. XINIX maneja sólo el drive A de la PC. El diseño de XINIX está basado en 2 sistemas operativos XINU [COMER84], y MINIX [TANEN87].
De XINU se tomó el manejo, la coordinación y la comunicación entre procesos, el manejo de memoria y el sistema de archivos; de MINIX se tomó el manejador de disco.

XINIX tiene un sistema de archivos plano: no permite manejar subdirectorios; tiene capacidad limitada en el número de archivos en disco: solo 28; tiene intérprete de comandos y es posible correr programas de manera interactiva. XINIX puede ejecutar varias tareas o procesos concurrentemente, esto significa que comparten el procesador y tal vez datos e instrucciones durante su ejecución. Esta concurrencia permite optimizar el uso de los recursos de un computador.

En XINIX, un proceso es un programa en ejecución que utiliza: tiempo de CPU, memoria, dispositivos de E/S, etc. y que puede estar

en uno de los siguientes estados:

READY.- el proceso esta listo para recibir el procesador.
SUSPEND.- el proceso se encuentra en el limbo, ni compite por el procesador, ni espera por algún evento, sólo esta latente, otro proceso debe sacarlo de su letargo.
WAITING.- el proceso espera por un evento; mientras tanto no compite por el procesador; el evento lo provoca otro proceso, y cuando ocurre pasa al estado READY.
RECEIVING.- el proceso espera por un evento en particular: un mensaje por parte de otro proceso; mientras espera no compite por el procesador; cuando recibe el mensaje pasa a READY.
SLEPPING.- el proceso espera a que transcurra un intervalo de tiempo para volver a pasar al estado READY.
CURRENT.- es el estado del proceso que se ejecuta actualmente.

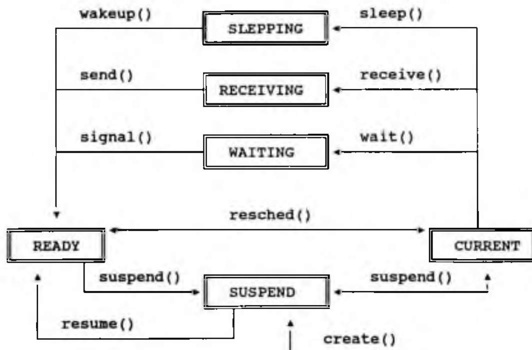


Figura 1.1 Estados de los procesos en XINIX.

Los estados están indicados dentro de los cuadros, las flechas indican el sentido en que los procesos pasan de un estado a otro, y los nombres seguidos de paréntesis "()" son los procedimientos servicio que realizan la transición.

XINIX asigna el procesador (ó intervalo de tiempo), sólo a aquellos procesos cuyo estado es READY. La asignación se hace de acuerdo a la prioridad de los procesos, y sólo se da a aquellos procesos con mayor prioridad, la prioridad varía de 0 a 32767, siendo 0 la menor.
Si varios procesos tienen la prioridad más alta, éstos comparten los intervalos de tiempo equitativamente: un proceso recién ejecutado debe esperar a que los otros procesos, con su misma prioridad, reciban un intervalo antes de él recibir otro; para esto, los procesos se manejan en una lista con política "primero en entrar, primero en salir". A éste esquema de asignación se le llama "Round Robin", y en este caso se ignora totalmente a procesos con menor prioridad: si sólo existe un proceso con la prioridad más alta, sólo él se ejecuta.
Cuando no existe ningún proceso en estado ready, quien se encarga de consumir el tiempo de procesador es el proceso "nulo", cuya prioridad es 0. Este proceso no hace absolutamente nada, es un ciclo infinito.

1.2 Organización lógica de XINIX

La organización lógica de XINIX se refiere a las capas (o componentes) de XINIX por el tipo de servicios que ofrecen: manejo de memoria, manejo de procesos, sistema de archivos, etc.(fig.1.2). Cada capa constituye u ofrece servicios para las capas superiores incluyendo a los programas del usuario. Por ejemplo, el intérprete de comandos utiliza al servicio create(), de la capa manejador de procesos, para ejecutar (crear un proceso a partir de) un programa

en disco de usuario. El programa a su vez puede, utilizando también a create(), crear procesos a partir de procedimientos encadenados con él.

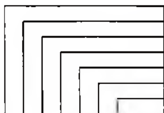


Figura 1.2 Organización lógica de XINIX. Capas.

Cada capa se forma de procedimientos y datos globales. Un servicio es un procedimiento en una capa, más no todos los procedimientos en ésta son servicios; p.ej., create() llama a newpid(), otro procedimiento del manejador de procesos, para obtener la identificación que se asignará al proceso a crear; newpid() busca una entrada libre en el arreglo global proctab[] (donde se encuentran los atributos de cada proceso existente en el sistema), y si la encuentra devuelve su número. La capa "coordinación de procesos" utiliza el arreglo semaph[] para implementar los semáforos. Para crear un semáforo, screate() llama a newsem(), el equivalente de newpid(), para buscar una entrada libre en semaph[]. Los procedimientos en otras capas también utilizan variables y arreglos globales para manejar dispositivos, bloques de memoria, archivos, almacenar nombres de comandos, etc. En general, los procedimientos en cada capa realizan una función definida sobre un tipo de objeto en particular: un proceso, un semáforo, un archivo, un diskette, una terminal, etcétera. Por su parte los objetos, sus atributos, son los campos que forman una entrada de un arreglo global en una capa; mientras que los valores en estos campos constituyen el "estado" actual del objeto: un proceso suspendido, un archivo abierto, un mensaje enviado; son estados que se

detectan, por los servicios de una capa, al preguntar por el valor de un campo-atributo.

1.3 Tipos de servicios.

La utilidad de un sistema operativo depende de la cantidad y calidad de los servicios que ofrece a los programadores. Hay dos maneras básicas de proveerlos: como "llamados al sistema" y como "programas de sistema", los que en XINIX se refieren como comandos internos y comandos externos respectivamente. Esta definición de los servicios considera la ubicación del conjunto de intrucciones, (en memoria ó dispositivo) que los realizan al momento de solicitarlos.

Todos los servicios que XINIX provee son "comandos internos", y son los siguientes (+ comando del intérprete, * servicio a programas de usuario):

1) Servicios para el control de procesos.

- Carga y ejecución de programas
  - + **exec** programa
  - +* **create** programa
  - +* **resume** programa
- Terminación o cancelación de un proceso.
  - +* **kill** id-proceso
- Obtención y modificación de los atributos de un proceso.
  - +* **chprio** id-process newprio
  - +* **get_priority** id-proceso
  - +* **get_status** id-proceso
  - * **getpid**
- Suspensión temporal de un proceso.
  - +* **suspend** id-proceso
  - +* **sleep** n
- Manifestar un evento en el sistema.
  - +* **send** id-proceso mensaje

+* **signal** id-semáforo

- Suspensión de un proceso hasta que ocurra un evento particular.
  - * **wait** id-semáforo
  - * **receive**

2) Servicios para el manejo de archivos.

* **open** id-dispositivo, nom-arch, modo

+* **close** id-archivo

* **read** id-archivo, buffer, num-chars
* **write** id-archivo, buffer, num-chars
* **seek** id-archivo, offset

+	**cp** archivo1 archivo2	(copia de archivos)
+	**mv** oldname newname	(renombra archivos)
+	**rm** archivo	(borra archivos)
+	**cat** archivo	(despliega contenido)
+	**fmt**	(formatea un diskette)
+	**vrf**	(verifica un diskette)
+	**ls**	(listar el directorio)

3) Servicios para el manejo de memoria.

* **getmem** nbytes
* **mkpool** tamaño-buffer, num-bufs, areapoolblk
* **getbuff** id-pool
* **freemem** buffer
* **freebuf** id-pool tamaño

4) Servicios para el manejo de dispositivos.

* **control** dispositivo, función, arg1, arg2
* **breakt** puerto
* **disablet** puerto
* **initport** puerto, vel, bpc, bp, p
* **enablet** puerto
* **nobreakt** puerto

* **getc** dispositivo
* **putc** dispositivo carácter

5) Servicios para mantener la información del sistema.
   + **devs** (lista los dispositivos)
   + **who** (lista la ident. de los usuarios en terminales)
   + **mem** (muestra el estado de la memoria)
   + **bpool** (muestra el estado de las despensas de bloques)
   + **ps** (muestra el estado de los procesos)

1.4 Manejadores de Dispositivos.

XINIX tiene dos dispositivos para E/S: el disco "A" y terminales, los cuales se manejan de acuerdo a sus caracteristicas y dentro del ambiente de concurrencia, esto es muy importante y donde más claramente se ve la diferencia entre un S.O. monousuario y uno de tiempo compartido. En el primero, como en el SO DOS, cuando un programa lee datos del usuario através de la terminal, el procesador se mantiene ocupado en un ciclo hasta los datos son tecleados, lo que puede durar bastante tiempo si el usuario está distraido; esto, sin embargo, no importa mucho pues dicho programa es el único que está ejecutándose. En XINIX esta situación se maneja en forma muy diferente, pues en éste puede haber más de un proceso requiriendo tiempo de procesador, por lo que éste no puede gastarse en esperar por un evento aleatorio como es el tecleo de carácteres. En XINIX, cuando un proceso lee datos de la terminal, éste es bloqueado (estado waiting) si no los hay, y el procesador se asigna al proceso en estado READY con la más alta prioridad. Cuando los datos se introducen, el proceso bloqueado pasa de nuevo al estado READY. Existen otras situaciones en las que DOS utiliza

al procesador en checar la completitud de una operación con dispositivos, la lectura y escritura a disco es una de éllas. Para leer o escribir un sector de datos en el diskette, se programan los chips Floppy Disk Controller (FDC) y el DMA: se escriben en sus puertos asociados varias secuencias de bits que constituyen el comando. Al FDC se le indica la pista, el lado, el sector y si éste va a leer o escribir. Mientras que al DMA se le indica la dirección de memoria, a partir de donde se ha de leer lo que se escribirá en el diskette, ó donde se escribirá lo que se lea de diskette. La última secuencia de bits escrita en el FDC, según el comando, inicia la operación de transferencia. DOS espera a que la operación termine: **el procesador se utiliza en esta espera.** El manejador de diskette de XINIX programa los chips FDC y DMA, pero en lugar de esperar el término de la operación ejecuta **receive()**: espera, sin competir por el procesador, por el mensaje que indica la terminación de la operación, enviado por el proceso que atiende la interrupción del FDC.

Capitulo 2

Sistema de archivos original de XINIX

La forma más usual y práctica de tener almacenados programas y datos es tener "archivos" en algún dispositivo de almacenamiento secundario, tales como discos, diskettes ó cintas. Un archivo es un objeto que perdura aún cuando el programa que lo creó termina.
El sistema de archivos como parte del sistema operativo es el encargado de proveer operaciones que crean, abren, leen, escriben y borran archivos. Los archivos son referidos con un nombre. En algunos sistemas el nombre indica el contenido del archivo: datos, programa ejecutable, código fuente ó código objeto.

2.1 Organización Física del Sistema de archivos XINIX.

El sistema de archivos XINIX está diseñado para trabajar con un solo diskette en el drive A de una PC. Para usar un diskette con el sistema de archivos primero se debe formatear. El formato de un diskette se obtiene con el comando **fmt** de XINIX:

```
CONSOLE>>fmt
     Terminando requisiciones previas...
Insert new diskette for drive A
and strike Abort, Continue when ready? c
Head: 0 Cylinder: 1
```

El formateo se realiza en 2 partes: el formateo físico y el formateo lógico.

Formateo Físico.

Cuando un diskette está nuevo no tiene ninguna información acerca de pistas, sectores ó lados. La unidad básica de almacenamiento es un sector, 512 bytes. Los sectores se distribuyen en el diskette en pistas concéntricas, las cuales tienen 2 lados. Aunque es posible utilizar un sólo lado generalemente se usan los dos.

El formateo físico consiste en poner identificación a los sectores, para poder referirlos después. Los diskettes que XINIX maneja son de 5¼ pulg., doble lado y doble densidad. La densidad ó número de pistas en un diskette está determinado por el número de desplazamientos que puede realizar el brazo de la cabeza lectora. Para los manejadores de disco de una PC-XT normal, dicho brazo puede desplazarse a 40 posiciones diferentes en un diskette (las PC's que tienen manejadores de alta densidad tienen un brazo con mayor número de desplazamientos). En cada pista-lado los sectores se identifican con un número entre 1 y 9:

pista	lado	sector
0	0	1
.	.	.
.	.	.
.	.	9
0	1	1
.	.	.
.	.	.
.	.	9
1	0	1
.	.	.
.	.	.
.	.	.
39	1	9

Hay en total 720 sectores en un diskette (40 x 2 x 9), cuya identificación lineal va de 0 a 719. El manejador de diskette, al recibir una operación de lectura ó escritura de un sector, recibe su identificación lineal, y la traduce a su respectiva identificación de pista-lado-sector que usa para programar al FDC. Hecho el formateo físico, ya es posible referir sectores en una pista-lado.

Formateo lógico.

Este consta de los siguientes pasos:

- verificar cada pista del diskette.
- identificar las pistas que tengan sectores malos (que no se puedan leer).
- enlazar los bloques del diskette.
- grabar una estructura de datos para el manejo del sistema de archivos.

El enlazamiento se hace para tener la identificación de los bloques que pueden ser usados como bloques de datos. Para enlazarlos se ponen en los dos primeros bytes de cada bloque el número del siguiente bloque disponible, fig.2.1.

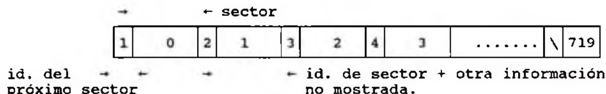


Figura 2.1 Enlazamiento de bloques.

No se puede asumir que todos los bloques están disponibles: los bloques que no se pudieron leer, no se enlazan.

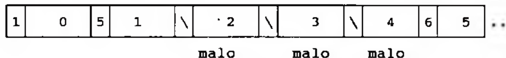


Figura 2.2 Evitando bloques malos.

Una vez enlazados los bloques, se procede a grabar el sistema de archivos en el diskette, él cual ocupa los primeros seis bloques; ver la distribución final de los sectores en la fig. 2.3. El bloque de boot se deja reservado para el programa que pueda "levantar" al sistema operativo desde diskette; el bloque 1 es usado para el

directorio de archivos XINIX; los bloques 2, 3, 4 y 5 son usados para bloques indice, y el resto de los bloques son para archivos del usuario.

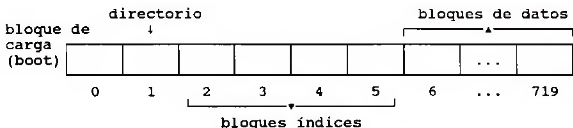


Figura 2.3 Distribución del sistema de archivos en diskette.

Cabe mencionar que es posible manejar sectores lógicos más grandes que los sectores físicos; sin embargo, en XINIX los sectores lógicos son del mismo tamaño, 512 bytes, que los sectores físicos.

2.2 Organización lógica del Sistema de archivos XINIX.

Como ya se explicó, cuatro sectores de un diskette están ocupados por bloques indice (i-blocks). Un i-block es una estructura cuya utilidad consiste en almacenar los números de bloques de datos pertenecientes a un archivo (29 en cada i-block); por lo que cada i-block indexa 29 x 512 = 14848 bytes de un archivo. Los i-blocks en disco, al igual que los bloques de datos también están enlazados. Cada uno de éstos se define:

```
struct iblk{
   long ib_byte;     /* número del byte en todo el archivo
                        que direcciona este i-block       */
   int  ib_next;     /* número del siguiente i-block */
   int  ib_dba[29];  /* arreglo para los números de bloques
                        de datos */
}
```

A su vez, el sector directorio se compone de 2 partes:

- El encabezado que describe al sistema de archivos.
- Las entradas para 28 archivos.

A continuación se muestran las partes del sector directorio:

```
struct fdes{                    /* descripción de cada archivo */
    char fdname[10];            /* nombre del archivo */
    long fdlen;                 /* longitud en bytes */
    int  fdiba;                 /* número del primer i-block */
};

struct dir{                     /* directorio */

    int  d_fblst;               /* apun. a la lista de bloques
                                   libres */
    int  d_filst;               /* apun. a la lista de i-blocks
                                   libres */
    int  d_iblks;               /* No.iblocks en el diskette */
    int  d_id;                  /* identificación de disco */
    int  d_nfiles;              /* número actual de archivos */
    struct fdes  d_files[28];     /* descripción de los
                                     archivos */
}
```

La organización lógica completa aparece en la figura 2.4.

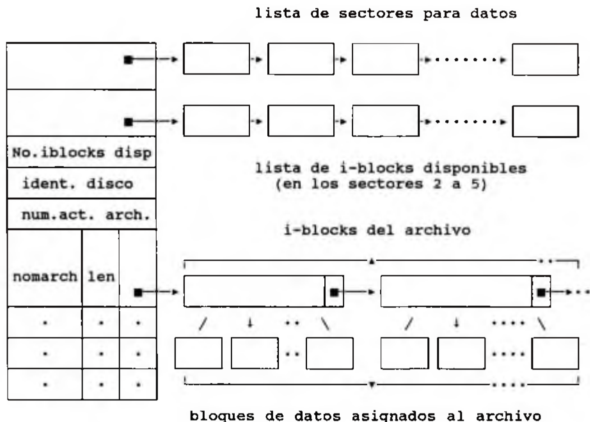


Figura 2.4 Organización lógica del Sistema de archivos XINIX.

2.3 Manejo de archivos.

El sistema de archivos XINIX ofrece siete servicios básicos: open, read, write, putc, getc, seek y close. Estos servicios son de la misma naturaleza que los usados para los dispositivos (terminales y disco). Todos los dispositivos tienen un nombre mnemónico, y asociado con éste, un número que los identifica. XINIX

maneja los archivos fueran dispositivos, por lo que se designaron 5 mnemónicos e identificadores asociados con éstos. De ésta manera, las capas superiores utilizan dispositivos o archivos, usando los mnémonicos correspondientes, con los servicios antes mencionados, por ejemplo:

putc(id-dispositivo, carácter)

es un llamado que se hará específico de acuerdo a la identificación del dispositivo ó archivo., como puede ser: ttyputc(), si es una terminal; lfputc() si es un archivo. ¿ Cómo es posible ? Através de la tabla **devtab[]**, cuyas entradas se accesan según la identificación del dispositivo ó archivo. Cada entrada en devtab[] tiene la dirección de las funciones reales que ejecutan los servicios específicos ya mencionados, para ese dispositivo.
Por ejemplo, el llamado read(DISK0, buffer, 512) se traduce a un llamado a la función dsread(DISK0, buffer, 512) del manejador de disco; y read(FILE4, buffer, 25) se traduce a un llamado a la función lfread(FILE4, buffer, 25) del sistema de archivos.

Cuando un programa de usuario abre un archivo se debe establecer una conexión entre éste y una entrada en el directorio de disco. Esta conexión es una tabla en memoria, donde cada entrada está definida así:

```
struct flblk {
        int  fl_id;     /* iden como dispositivo del archivo*/
        int  fl_dev;    /* iden drive donde está el archivo */
        int  fl_pid;    /* iden proceso q' accesa el archivo*/

        struct fdes  *fl_dent; /* entrada del directorio en
                                  memoria */

        int  fl_mode;   /* modo: read, write o ambos */
        int  fl_iba;    /* número de iblock en fl_iblk */

        struct iblk  fl_iblk;  /* iblock actual para el
```

```
                                   archivo */
        int  fl_ipnum; /* núm. actual bloque de datos en
                                   fl_iblk */
        long fl_pos;    /* posición actual en archivo(bytes)*/
        int  fl_dch;    /* ha sido fl_buff cambiado? */
        char fl_bptr;   /* apun. sig. carácter en fl_buff */
        char fl_buff[512]; /* bloque de datos actual para el
                                archivo */
}
```

Así, para cada archivo abierto existe suficiente información para habilitar las funciones que extraen ó modifican datos del archivo.

2.4 Comandos del Intérprete sobre archivos.

El intérprete de comandos provee los siguientes comandos sobre archivos:

>>**fmt**

Formatea un diskette y le graba el sistema de archivos XINIX.

>>**ls**

Lista el directorio de un diskette XINIX: el nombre y longitud en bytes para cada archivo.

>>**mv**

Cambia el nombre de un archivo.

>>**rm**

Borra un archivo del diskette.

>>**cat**

Concatena (copia) uno ó más archivos en uno solo, ó despliega el contenido de un archivo por la terminal.

>>**cp**

copia un archivo a otro, no existe sobreposición: no debe existir el archivo destino.

>>**cdxf**

Copia un archivo de un diskette DOS a un archivo en un diskette XINIX.

>>**devs**

Lista los dispositivos que maneja el sistema. Incluye el número

de archivos que pueden estar abiertos simultáneamente y su identificación "NUM" para utilizarla con el comando close.

>>**close**

Cierra un archivo. Util cuando un proceso de usuario no lo cierra antes de terminar.

2.5 Uso de servicios del sistema de archivos en programas de usuario.

Un programa que ha de correr en XINIX, debe incluir el archivo "xinixh.h" para tener acceso a las macros y servicios de XINIX. Es responsabilidad del usuario verificar el valor que devuelva cada servicio.

Open. EL servicio open no se relaciona con un identificador de archivos en si, sino con un manejador de diskette (donde se encuentra el sistema de archivos), el llamado en programas de usuario debe ser:

```
int file_id;
file-id = open(DISK0, nombre-arch, modo);
```

DISK0 es el identificador del manejador. Nombre-arch es el nombre (no más de 9 carácteres + "\0") del archivo a crear ó abrir. Modo, deberá ser una cadena de los carácteres "o"ld, "r"ead, "w"rite, "n"ew, para indicar la creación o apertura del archivo, p.ej: "nw" ó "orw". Open devuelve un entero, el identificador para el archivo recién abierto que se usará para los siguientes servicios.

Getc. Dado un identificador de archivo devuelve el siguiente carácter.

```
int file_id;
char c;
c = getc(file_id);
```

Putc. Dado un identificador de archivo escribe un carácter en él.

```
int file_id;
char c;
putc(file_id, c);
```

Read. Este servicio está construído en base a putc.
Write. Este servicio está construído en base a getc.

```
int file_id, numchars;
char buffer[n];     /* ó equivalente */
read(file_id, buffer ,numchars);
write(file_id, buffer, numchars);
```

Seek. El servicio seek mueve la posición del archivo a la posición offset, no transfiere datos.

```
int file_id;
long offset;
seek(file_id, offset);
```

Close. Cierra un archivo dado su identificador.

```
int file_id;
close(file_id);
```

Capítulo 3

Características del nuevo Sistema de Archivos de XINIX

En este capítulo se explican las estructuras básicas sobre las cuales se construyen los servicios que presta el nuevo Sistema de Archivos XINIX (NSAX), su organización en diskette, y su organización lógica en memoria.

3.1 Organización física.

La figura 3.1 muestra la organización física del NSAX después de haber formateado un diskette.

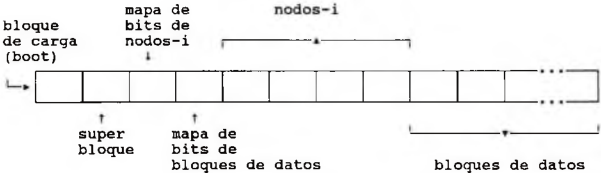


Figura 3.1 Organización en diskette del Sistema de Archivo.

En el SA anterior de XINIX, el tamaño de los bloques en diskette eran de 512 bytes, en este NSAX es de 1024 bytes, lo que se traduce en una optimización (disminución) de accesos a disco. Un diskette tiene con 720 bloques físicos tiene entonces 360 bloques lógicos de los cuales se usan 8 para el NSAX, quedando 352 bloques

disponibles para los archivos del usuario.
El bloque de carga (boot block) no se utiliza, pero se deja reservado para el programa que "levante" al SO desde diskette.
El superbloque mantiene información que describe al NSAX en diskette, su función principal consiste en indicar el tamaño de las partes (mapas de bits, nodos-i, etc.) de la figura 3.1.
El control de la asignación de nodos-i y de bloques de datos se lleva usando mapas de bits. Los bits se manejan agrupados en números enteros de 16 bits sin signo. Cada entero expresa el estado, asignado ó no, de 16 bloques de datos ó nodos-i. Hay 127 nodos-i disponibles en un diskette (ocupando 4 bloques) para ser usados por archivos de datos y directorios. Su número puede variar de acuerdo a las necesidades de los usuarios del SO, menos si hay pocos archivos de gran tamaño, ó más si hay muchos archivos pequeños.

3.2 Organización lógica.

Cuando XINIX se inicia, se crean varias tablas y estructuras en memoria, así como en diskette hay superbloque, mapas de bits, nodos-i, y bloques de datos, al montarse un SA para trabajar con archivos, la información contenida en éstos debe ponerse en dichas tablas además de información que no es necesario tener en disco pero sí en memoria mientras se utilizan. Esto implica hacer lecturas a disco, las cuales suelen ser bastante lentas, con respecto a la velocidad del procesador. Para minimizar la frecuencia de accesos a disco y mantener la mayor información de disco en memoria se creó el buffer cache.

3.2.1 Buffer Cache.

El buffer cache es una estructura en memoria que se crea cuando XINIX se inicializa. Dicha estructura la forman dos partes: un arreglo de buffers que contiene datos leídos de disco, y una tabla

de hash por medio de la cual se controla el arreglo de buffers.

Cada buffer a su vez está formado de dos partes: un encabezado que identifica al buffer y un arreglo del mismo tamaño que los bloques de disco (1024 bytes), que contiene la información leída de uno de éllos. Así, cada buffer en memoria tiene una copia de un bloque de disco, el cual se mantiene ahí hasta que el NSAX decida poner la información de otro bloque de disco en ese buffer. La información de un bloque de disco nunca está mapeada en dos buffers a la vez, pues entonces el NSAX ya no sabría cual información es la más actualizada para un bloque de datos, y si hubo modificaciones en ambos buffers seguramente cualquiera que se escriba a disco hará que ese bloque de datos tenga información incorrecta. Enseguida se define la estructura de un buffer.

```
struct buf {
     char datos[1024];   /* Parte de datos del buffer */
             /* Parte del encabezado */
     struct buf *next;
     struct buf *prev;
     struct buf *hash;
     int  num_bloque;   /* num. del bloque de datos de disco
                         * al que esta asignado el buffer */
     int  dev;          /* identificacion del drive donde
                         * reside el bloque */
     char dirt;         /* indica si la parte de datos del
                         * buffer ha sido modificada */
     int  count;        /* num. de usuarios de este buffer*/
}
```

En esta implantación el buffer cache consta de 30 buffers. Cuando inicia XINIX, cada uno de estos buffers se enlazan en una lista doblemente ligada, y todos los buffers están disponibles (forman una lista de buffers libres). La cabeza de la lista libre se asocia con el buffer menos recientemente usado (LRU), y el final de la lista con el buffer usado más recientemente (MRU).

El método que usa el NSAX para manejar buffers es el de los "menos recientemente usados" (LRU): cuando se busca un buffer (uno

libre ó uno cargado previamente) se comienza por la cabeza de la lista, el extremo LRU, se recorre hasta encontrarlo, y se cambia de posición: se enlaza al final de la lista y se utiliza, ahora ese buffer es el extremo MRU, es el bloque "usado más recientemente". Dicho bloque va cambiando de posición según se utiliza el resto de los bloques; si el bloque estuviera en la mitad de la lista y se necesitara nuevamente se cambiaría otra vez al extremo MRU, y lo mismo sucede con el resto de los bloques. Si el bloque no vuelve a necesitarse, se estará moviendo hasta que llega nuevamente al extremo LRU, de esta manera un buffer no puede ser usado por otro bloque hasta que todos los buffers restantes hayan sido usados más recientemente, el NSAX mantiene la lista de buffers en orden LRU. Cuando el NSAX necesita accesar un bloque de disco, lo busca primero en el buffer cache, por si el bloque fue leído anteriormente, sino lo encuentra, hace la petición de lectura a disco. El buffer cache se accesa por medio de una tabla de hash. Cada entrada en la tabla corresponde a una lista de bloques leidos de disco cuya identificación de sus 5 bits menos significativos, es la misma. Existen 2^5 = 32 entradas en la tabla de hash. La función de hash usada consiste en extraer los 5 bits menos significativos del número de bloque, de manera que los bloques de diferentes diskettes con igual número de bloque, aparecerán en la misma lista de hash. Por ejemplo: todos los bloques con los 5 bits menos significativos iguales a 000110 estarán enlazados en la misma lista de hash. Cada buffer existe siempre en una lista de hash, y como ya se había mencionado, dos buffers no pueden contener el mismo bloque de disco, por lo tanto cada bloque de disco existe solo en una lista de hash y solo una vez en ella; un buffer está siempre en una lista de hash y puede ó no estar disponible, en cuyo caso es probable que cuando se utilice se cambie de lista de hash; la figura 3.2 aclarará esto. Al solicitar un bloque de disco, se sabe en qué dispositivo está y también su número de bloque; en base a éste, se obtiene la respectiva lista de hash, si el bloque ya había sido cargado se encontrará ahí y se incrementa el contador

del encabezado del buffer indicando que hay un usuario más de ese bloque, (lo cual evita un acceso a disco); si no se encontró en la lista de hash, se busca un buffer en la lista libre en modo LRU cuya cuenta de uso sea cero (nadie lo usa actualmente), se elige el primero que encuentre con esas condiciones; además se verifica si el buffer fue modificado, en cuyo caso primero se escribe a disco para almacenar las modificaciones hechas y evitar inconsistencias en el NSAX, si no ha sido modificado, el buffer puede utilizarse enseguida. Hay bloques, como los mapas de bits que deben permanecer en el buffer cache sin importar el poco uso que tengan mientras esté cargado el sistema de archivos al que pertenecen, y hay otros, como los bloques indirectos de nodos-i (de los que se habla más adelante), cuya probabilidad de uso próximo es muy poca, lo cual se ponen a la cabeza de la lista LRU para que sean elegidos la siguiente vez.

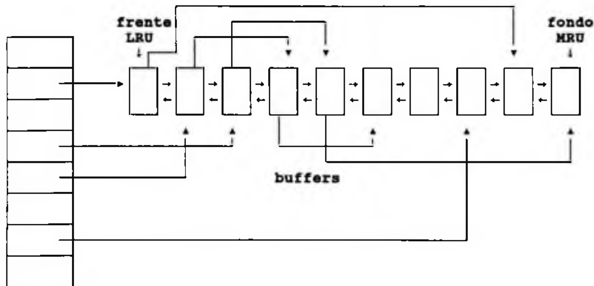


Tabla de hash

Figura 3.2 Buffer Cache.

Cabe mencionar que al inicializar el buffer cache, el espacio de

cada buffer donde va la información de disco, se verifica que no atraviese una frontera física de 64k en memoria para evitar problemas de transferencia con el DMA; el buffer que atraviese una frontera no se utiliza, por lo que no se encadena, y se continúa con el siguiente. En cuanto a las peticiones de lectura o escritura en disco, cabe mencionar que cuando se hacen lecturas, el NSAX se bloquea hasta que la petición se satisface, es síncrona; no así la escritura, pues el NSAX encola la petición y continúa ejecutandose, el manejador de disco realizará la escritura poco después o inmediatamente, según haya ó no peticiones ya encoladas. El NSAX recibe el resultado, correcto ó erróneo, de una petición de lectura, pero no de una petición de escritura. Si en ésta hubo algún error, el manejador de disco lo notifica en la terminal principal (CONSOLA).

3.2.2 Nodos-i.

Todo SA controla la asignación de espacio en disco para los archivos. En el NSAX el control se realiza usando nodos-i.
Un nodo-i (nodos índices) es una estructura que contiene las características de un archivo, y la identificación de sus bloques. ES importante mencionar que tanto archivos de datos como subdirectorios se tratan de la misma manera, para ambos se piden nodos-i, y solo se distinguen por el **modo** en que se crean, pero sus entradas son exactamente iguales. Veamos en la figura 3.3 una entrada en un directorio.

nombre del archivo	num. nodo-i

Figura 3.3 Estructura de las entradas en directorio

La importancia de los nodos-i radica en mantener la identificación de los bloques de datos que han sido asignados al archivo. Cada nodo-i tiene la identificación directa de 7 bloques de datos, y dos

identificaciones más que corresponden respectivamente, a un bloque indirecto sencillo, y a un bloque indirecto doble. Un bloque indirecto tiene en lugar de datos de archivos, las identificaciones de tantos bloques de datos como pueda direccionar, ya que cada bloque es de 1024 bytes y la identificación de un bloque es de tipo entero (2 bytes), así que cada bloque indirecto contiene la identificación de 512 bloques de datos; un bloque indirecto doble

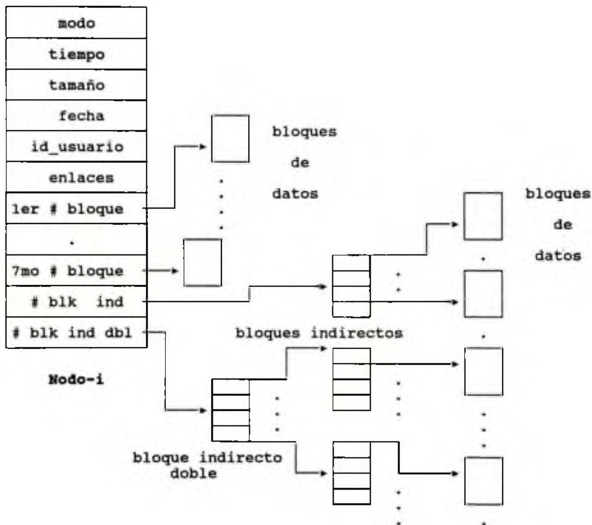


Figura 3.4 Estructura de un nodo-i.

tiene la identificación de 512 bloques indirectos sencillos, como

la figura 3.4 lo muestra.

Cuando el tamaño de un archivo sobrepasa los 7k se utilizan los bloques indirectos. En realidad, cuando se trabaja con diskettes de 360k bastaría con tener el bloque indirecto sencillo, pues 1024 bytes * 512 bloques direccionan ½ megabyte, pero si XINIX tuviera manejador de disco duro se haría indispensable el bloque indirecto doble si hubiera archivos demasiado grandes.

En el NSAX se distinguen 3 tipos de archivos: archivos de tipo regular (de datos de usuario), archivos de tipo directorio, y archivos especiales. Para el primero es el usuario quien se encarga de crearlos y controlarlos, para el segundo el usuario los manda crear pero es el NSAX quien lleva el control de ellos: el usuario no puede cambiar la estructura de un directorio, solo puede agregar y/o borrar archivos; los archivos especiales son únicamente nodos-i (no ocupan bloques de datos), por medio de los cuales se puede hacer referencia y uso de un dispositivo, principalmente manejadores de disco (ver comando mount del apéndice A), éstos se crean al inicializar XINIX; no es recomendable que un usuario común intente crear alguno de éstos, esa tarea correspondería a quien le interese modificar XINIX.

Control de acceso a archivos.

El tipo de archivo se indica en el **modo**, así como también, los derechos de acceso a archivos. Sin embargo en el NSAX no es posible hacer uso de los derechos de acceso porque trabajando en diskettes no es posible tener un archivo de identificaciones de los usuarios del sistema, lo cual es la base para discriminar **quién** tiene **qué** derechos en el sistema; de cualquier manera se explica para cuando sea posible manejar el NSAX en disco duro.

Para cada archivo y cada directorio en el NSAX hay 3 clases de usuarios:

- Dueño (owner). Es el usuario quién inicialmente lo creó.

- Grupo (group). Es un grupo de usuarios con los mismos derechos sobre ciertos archivos y directorios.
- Otros (public). Son el resto de los usuarios del sistema.

Cada tipo de usuario tiene 3 tipos de acceso a los archivos:
- permiso de lectura (r).
- permiso de escritura (w).
- permiso para ejecución ó búsqueda si es directorio (x).

El campo "modo" es de 16 bits, en donde los 3 bits menos significativos, indican los permisos de acceso para "Otros", los siguientes 3, indican los permisos de acceso para el "Grupo", los siguientes 3, es el permiso de acceso para el "Dueño". Al listar un directorio aparecen representados así: -rwx rwx rwx el "-" al comienzo indica que es un archivo de datos, cuando es directorio se pone "d", cuando algún permiso no es permitido se representa con "-", p.ej. el permiso: -rw-rw-rw indica permisos de sólo lectura y escritura para el dueño, el grupo y otros. A continuación se muestra como se pueden hacer las diferentes combinaciones, considere que son valores octales, cada dígito se forma con 3 bits.

00400	permiso de lectura para el dueño.
00200	permiso de escritura para el dueño.
00100	permiso de ejecución (búsqueda) para el dueño.
00070	permisos de lectura, escritura y ejecución para el grupo.
00007	permisos de lectura, escritura y ejecución para otros.

00777	

Así, un archivo con modo 0777 indica permiso de lectura, escritura y ejecución para cualquier usuario, y un modo 0444 indica permiso de sólo lectura para cualquier usuario. En el bit menos significativo del cuarto dígito de derecha a izquierda indica el SET_UID del archivo pero el NSAX tampoco lo usa, actualmente lo

pone a 0 (ver [KERN87] pags. 57-58). Los siguientes 3 bits del cuarto dígito (los dígitos siguen siendo octales) indican el tipo de archivo :

02000	archivo especial de caracteres (iden. terminales).
04000	el archivo es un directorio.
06000	archivo especial de bloque (iden. manejadores de disco).
10000	archivo ordinario.

Así, los modos de un archivo normal y uno directorio con todos lo permisos permitidos serían (en octal) respectivamente: 100777 y 040777.

Tipos de archivos y manejo de directorios.

Cuando un disco se formatea, queda grabado el SA e inicializado su directorio raíz. La inicialización del directorio (y de cualquiera se cree subsecuentemente en un SA) consiste en incluir dos entradas en el directorio, la primera de ellas corresponde al directorio creado (el actual) y se indica con ".", y la segunda es la entrada del directorio "padre" del directorio actual, y se indica con "..". Cada entrada tiene su respectivo número de nodo-i, y con excepción del directorio raíz todos los directorios tienen un directorio "padre". En el caso del directorio raíz ambas entradas tienen el mismo número de nodo-i. El "padre" del directorio raíz es él mismo.

En vista de que pueden existir varios niveles de subdirectorios hay dos maneras de hacer referencia a un archivo en cualquier directorio: usando trayectorias absolutas ó relativas; una trayectoria absoluta es aquella que comienza en el directorio raíz, y una relativa es la que comienza en el directorio actual. En ambos casos son los nodos-i asociados a los nombres de directorios y archivo en la trayectoria, los que se usan para localizar un archivo dado, como lo ejemplifica la fig. 3.5.

directorio raíz		nodo-i 5 /usr_a	bloque 20 directorio de /usr_a		nodo-i 9 /usr_a/dwork	bloque 33 directorio /usr_a/dwork	
.	1	modo tamaño	.	5	modo tamaño	.	9
..	1		..	1		..	5
dev	2	1er blk=20	xyz.c	8	1er blk=33	arch.c	56
usr_a	5		dwork	9		dinic.c	75
usr_b	6		abc.c	7		graf.c	81

Figura 3.5 Conversión de trayectorias a nodos-i.

Si la trayectoria es /usr_a/dwork/dinic.c, se busca primero a usr_a en el directorio raíz, se toma su número de nodo-i, se localiza y de él se toma el bloque de datos, el cual contiene el directorio usr_a. En éste se busca la siguiente parte, dwork,y así sucesivamente hasta llegar al archivo buscado, en este caso dinic.c.

Si al requerir un nodo-i, el bloque en el que se encuentra no se ha cargado de disco, se hace la petición, y cuando ésta se completa, ya se encuentra dicho bloque en el buffer cache; el paso siguiente es copiar el susodicho nodo-i a la tabla de nodos-i en memoria, en donde cada entrada además de tener la información de un nodo-i de disco, tiene campos adicionales que sirven para llevar el control de su uso en memoria. Veamos la definición de un nodo-i.

```
struct inode{
    unsigned int modo; /* tipo archivo, protección */
    unsigned int time; /* tiempo de ultima actualización */
    long size;         /* tamaño actual del archivo bytes*/
```

```
    long date;          /* fecha ultima actualización */
    int  uid;           /* id del usuario propietario */
    char links;         /* número de enlaces del archivo */
    int blk[9];         /* núms. de blks dir, ind, dbl_ind*/
    /* los siguientes campos no aparecen en disco */

    int  dev;           /* dispo.donde se encuentra nodo-i*/
    int  num;           /* núm de nodo-i en ese dispo. */
    int  count;         /* veces q' es usado el nodo-i;
                         * 0 significa q' esta libre */
    int  dirt;          /* ha sido modificado el nodo-i? */
    int  pipe;          /* 1 es pipe 0 no lo es */
    int  mount;         /* si el nodo-i esta montado */
    int  seek;          /* fue seek la última operación? */
}
```

El campo links indica cuantas referencias hay al nodo-i de un archivo, cada nodo-i debe tener por lo menos una: la del archivo al que pertenece. Cuando en un "directorio" se crea un archivo ó un subdirectorio, el campo links del nodo-i de "directorio" debe incrementarse en uno, porque ya hay una referencia más a él, ahora es un "directorio padre". Los campos que sólo existen en memoria sirven para indicar si ha sido modificado, si el nodo-i está montado, si el nodo-i es de un pipe, si el número de referencias actuales que tiene un nodo-i es 0, la entrada de la tabla en memoria se considera libre. La tabla en memoria tiene capacidad para 32 entradas de nodos-i. Los procedimientos del NSAX refieren ó modifican nodos-i, por medio de un apuntador a la entrada en la tabla donde éstos se encuentran. La modificaciones a los nodos-i se hacen en las entradas de la tabla, más cuando se requiere escribir el bloque de nodos-i del buffer cache a disco, éste se actualiza con las modificaciones de la tabla, antes de escribirle, lo que evita inconsistencias en el NSAX.

3.2.3 Superbloques.

La función principal de un superbloque es la de indicar el tamaño de las partes que forman un SA en disco, más cuando un SA se monta,

su superbloque se carga primero en el buffer cache y de ahí a una tabla de superbloques en memoria, que al igual que para los nodos-i, se tienen campos adicionales para el control de todo el SA que fue montado. Veamos la definición de un superbloque.

```
struct superblock{
     int  s_ninodes;         /* núm. de nodos-i en disco */
     int  s_size;            /* total de bloques en disco */
     int  s_imap_blk;        /* núm.blks mapa bits nodos-i */
     int  s_bmap_blk;        /* núm.blks mapa bits bloques */
     int  s_firstdatablk;    /* núm. 1er bloque de datos */
     long s_max_size;        /* tamaño máximo de un archivo*/
     int  s_magic;           /* id. superbloques XINIX */

/* los sig. campos sólo se usan cuando esta en memoria */
     struct buf *s_imap[4]; /* apun.mapa bits nodos-i en
                               memoria */
     struct buf *s_bmap[6]; /* apun.mapa bits bloques en
                               memoria */
     int  s_dev;             /* en qué drive se monto este
                               superbloque */
     struct inode *s_isup;  /* nodo-i del directorio raíz
                               del superbloque montado */
     struct inode *s_imount; /* nodo-i montado */
     int  s_time;            /* tiempo última actualización*/
     long s_date;            /* fecha última actualización */
     char s_rdonly;          /* montado sólo para lectura */
     char s_dirt;            /* se ha modificado? */
}
```

Después de cargar el superbloque se cargan los bloques de mapas de bits de nodos-i y de bloques de datos, y permanecen en el buffer cache hasta que el SA de disco se desmonta (adelante se explican los montajes). Los mapas de bits se refieren através de los arreglos de apuntadores s_imap y s_bmap en la entrada del superbloque en la tabla en memoria. Estos apuntadores contienen la dirección de los bloques en el buffer cache que contienen los mapas de bits.

Después de inicializar el buffer cache se crea el directorio raíz de XINIX, con los subdirectorios **dev**, **usr_a**, **usr_b**; las entradas del subdirectorio dev identifican a los manejadores de

disco como **fd0** y **fd1** (A: y B: en DOS), mientras que los directorios usr_a y usr_b están vacíos, no se puede ni debe crearse nada en ellos, se usan para montar sistemas de archivos de disco, **el montaje sirve para poder hacer referencia a sus archivos y directorios.** El montaje de un SA en disco se hace con el comando **mount**; la sintaxis del comando mount es la siguiente:

mount id_manejador directorio_vacío

Se sugiere que el manejador de disco en el que se inserte el diskette, coincida con la terminación del nombre del directorio donde se va a montar,p.ej.

mount /dev/fd0 /usr_a

indica que deseamos montar un SA del diskette insertado en el drive 0 (a: en DOS), en el subdirectorio usr_a. De ésta manera se mapea el directorio raíz del SA de disco con el subdirectorio usr_a, ahora se puede hacer referencia a cualquier archivo o subdirectorio en disco. Por último, recuerde que un archivo en un subdirectorio puede ser accesado usando trayectorias absolutas o relativas. Al usar trayectorias absolutas accesamos un archivo desde cualquier otro directorio en el SA, p.ej.:

/usr_a/directorio/..../archivo

Esto es posible gracias al campo s_isup que apunta al nodo-i del directorio raíz del SA montado, el campo s_imount apunta al nodo-i del directorio sobre el que se hizo el montaje. Usando ambos es como se pueden utilizar las trayectorias absolutas. He ahí la utilidad de los superbloques.

3.3 Comparación del sistema de archivos original (SAO) con el nuevo sistema de archivos (NSAX).

En este capitulo y en el anterior se explicó como están organizados lógica y fisicamente, el sistema de archivos original y el NSAX, las diferencias más notables y las que determinan el funcionamiento y eficiencia son:
-- Tamaño lógico de bloque.
-- Organización Lógica de bloques libres en disco.
-- Estructura de cada archivo.
-- Estructura de directorios.

Tamaño lógico de bloque.

En un sistema de archivos el tamaño lógico de bloque se determina en función del tamaño físico, ya sea igual ó un múltiplo de éste. En el SAO el tamaño de bloque lógico es igual al físico: 512 bytes. Además sólo un bloque puede mantenerse en memoria. Según el tamaño del archivo se tendrán (tamaño/512=n) n accesos a disco para poder leerlo completamente en forma secuencial, ahora bien, si se está accesando informanción directamente sin ningún tipo de orden, el número de accesos puede incrementarse notablemente, si se accesa información aleatoriamente, seguramente cada vez se tendrá que leer el bloque correspondiente de disco, sólo cuando lea información del mismo bloque, se evitará cargar el bloque de disco, pues este habia sido cargado previamente, recuerdese que sólo se mantiene en memoria el último bloque leido. Como se puede observar esto no es muy eficiente. Observese también que cuando en el último bloque de un archivo sólo se usa 1 byte se desperdician 511 bytes lo cual es un problema de fragmentación interna, pero no es muy grave (como se podra comparar más adelante).
En el NSAX el tamaño de bloque lógico es de 1024 bytes lo cual se realiza usando 2 bloques físicos consecutivos. El NSAX tiene un

buffer cache en el cual se pueden tener hasta 30 bloques lógicos leídos de disco lo cual significa que sólo la primera vez que un bloque es requerido, se transfiere de disco, pues en las siguientes ocasiones, dicho bloque ya se encuentra en el buffer cache y por lo tanto no se hace necesario hacer una tranferencia de disco. Existe una desventaja pues cuando se usan muy pocos bytes del último bloque, el resto se desperdicia y es espacio que no puede ser usado por ningún otro archivo, pues no hay manera de recuperarlo, este es un caso más grave de fragmentación interna.

Organización lógica de bloques disponibles en disco.

En el momento del formateo de un diskette se establece la organización lógica de los bloques disponibles.
EL SAO enlaza ó liga todos los bloques disponibles en un diskette (se crea una gran lista ligada de bloques) los cuales conforme se crean archivos se van asignando a ellos, eliminandose de la lista de disponibles, y se asignan a la lista de bloques que el archivo ocupa (usando i-blocks). Cuando se borra información de un archivo, ó se borra el archivo completo, los bloques que pertenecían a ese archivo se liberan (se desenlazan de la lista del archivo) y se reincorporan a la lista de archivos disponibles. Como puede verse la asignación y desasignación de bloques se lleva a cabo manejando listas ligadas en ambos casos, la desventaja es que la lectura y/o escritura de archivos se vuelve bastante lenta por el manejo de los apuntadores lo cual no es muy conveniente para un sistema de archivos.
El NSAX mantiene en mapas de bits tanto los bloques disponibles en diskette como los nodos-i. En un mapa de bits cada bit representa un bloque y ya sea éste 0 ó 1 el bloque se encuentra libre ó asignado. En el NSAX un bloque de disco es el mapa de bits para bloques disponibles en diskette, los bloques son de 1024 bytes o sea (1024 x 8) 8192 bits para el manejo de bloques. Como en un diskette sólo se tienen 360 bloques de 1024 bytes, el mapa de bits

es por mucho suficiente, al menos cuando se trabaja en discos flexibles, en disco duro se necesitaría más de un bloque para el mapa de bits; algo similar pasa con el mapa de bits de nodos-i. La ventaja del uso de mapas de bits es que, los mapas de un SA de disco permanecerán en memoria mientras está montado el SA. su manejo es muy sencillo, y en conjunto funciona bastante rápido.

Estructura del directorio.

El SAO tiene un único directorio con 28 entradas para archivos, en donde cada entrada define las características del archivo correspondiente, tiene además los apuntadores correspondientes a la lista de bloques e i-blocks libres.
EL NSAX tiene un directorio raiz con 64 entradas para archivos u otros directorios, una entrada consiste en el nombre del archivo y un número de nodo-i que tiene las características del archivo, generalmente el bloque del directorio raiz ya se encuentra en memoria, no así el de los nodos-i, pero al leer un bloque de nodos-i ya se tiene la información necesaria para poder accesar 32 archivos, y una vez en memoria ya no es necesario accesar a disco.

Estructura de los archivos.

La estructura de archivos que maneja el SAO se basa en el uso de listas ligadas usando i-blocks. Cuando un archivo se crea se pone su nombre en una entrada del directorio, se le asigna un i-block, lo cual significa que debe ser desenlazado de la lista de i-blocks disponibles, y enlazarlo a dicha entrada por un apuntador, un i-block es capaz de direccionar a 29 bloques de datos, cuando un archivo es más grande que eso, se asignará otro i-block el cual es enlazado al primero, y asi sucesivamente.
El NSAX basa la estructura de sus archivos en nodos-i. En un nodo-i se tienen las características que describen a un archivo, y además las identificaciones de los bloques de datos que ocupa.

Al crearse un archivo se le asigna una entrada en el directorio, se lee el mapa de bits de nodos-i y según el bit que se encuentre disponible será el número de nodo-i que se asigne al archivo. Cuando se empieza a escribir en él es necesario asignarle un bloque de datos libre, y al igual que con el nodo-i es necesario leer el mapa de bits de bloques disponibles, según el bit que se encuentre disponible será el bloque que se asigne al archivo y cuya identificación se pone en el nodo-i. En el NSAX los mapas de bits se encuentran siempre en memoria lo que evita hacer accesos a disco cuando se requiere asignar nodos-i ó bloques disponibles lo que representa un gran ahorro de tiempo cuando se leen o escriben archivos.

Capitulo 4

Modificación de XINIX para la implantación del Sistema de Archivos

En este capitulo se explican los cambios a XINIX requeridos por el nuevo sistema de archivos (NSAX), las modificaciones al sistema de archivos de MINIX [TANEN87] para adecuarlo al medio ambiente XINIX, y la interfaz de éste con aquél.

4.1 Cuando XINIX inicia.

XINIX es un programa ejecutable que corre en DOS. Sin embargo una vez en ejecución, XINIX toma el control de la PC: carga los vectores de interrupción con la dirección de los manejadores de reloj, de terminales y de diskette; inicializa las variables y tablas del sistema, y las listas de procesos; obtiene la memoria disponible, sobre la que actuará el manejador de memoria; establece el tamaño de los diferentes pools que puede crear el usuario; inicializa los manejadores de disco y crea el proceso DISK_OPERATOR, encargado de llevar a cabo las peticiones a disco; se inicializan las terminales y los procesos shell asociados a ellas; finalmente se crea el puerto de mensajes para el NSAX, y su proceso: **el Sistema de Archivos es un proceso en XINIX.** Cabe mencionar que el main() de XINIX se convierte en el proceso nulo del SO, y que al oprimir las teclas CTL ALT DEL simultáneamente se restablecen los vectores de interrupción y se regresa al medio ambiente DOS tal y como estaba antes de ejecutar XINIX.

El sistema de archivos inicializa sus variables, las tablas en memoria de superbloques, de nodos-i, de procesos y de archivos abiertos; después inicializa el buffer cache, enlazando los buffers

en una lista doblemente ligada e inicialmente todos en la misma lista de hash (libres). Durante el enlazado se verifica que cada buffer no quede entre una frontera física de 64k, el que la atraviese, se elimina: no es enlazado. Enseguida se crea el directorio raíz (/) con 3 subdirectorios: dev, usr_a, usr_b, los dos últimos vacíos, no se puede ni deben crearse archivos ahí, sólo se utilizan para montar SA de disco; en dev se encuentran las identificaciones de los drives fd0 y fd1 para el montaje de los SA.

El NSAX cuenta con su propia tabla de procesos, pues cada proceso en el sistema es un usuario en potencia del NSAX. Las tablas de procesos XINIX y NSAX tienen el mismo número de entradas, ambos asignan a un proceso dado, la misma entrada en su respectiva tabla de procesos. Así, los procesos nulo, disk_operator y los shells de las terminales se incluyen en la tabla de procesos del NSAX; La inclusión de las terminales tiene razón práctica: los cambios de directorio se asocian al proceso que los ejecuta, y es lo que permite la independencia de cambios de directorios de cada terminal en el NSAX. El proceso del sistema de archivos se incluye también en ambas tablas.

4.2 Manejo de procesos.

El servicio create() crea un proceso: pide memoria para la pila, asigna un pid al proceso y usando éste como índice se incluye tanto en la tabla de procesos del SO como en la del NSAX. El intérprete de comandos de una terminal es un proceso, si ejecuta un programa (con el comando exec, p.ej.), éste se convierte en su proceso hijo, y si a la vez, el programa crea un proceso, se convierte en el proceso hijo del programa (ahora proceso). Como todo proceso se genera de un proceso padre, es necesario éste herede al hijo ciertos atributos tales como: el directorio en el que se encuentra, los apuntadores a las ranuras de archivos abiertos, etc. Veamos la definición cada ranura de la tabla de procesos del NSAX.

```
struct fproc {
     unsigned int fp_umask; /* lo coloca el servicio umask*/
     struct inode *fp_workdir; /* apun. al directorio actual
                                  de trabajo */
     struct inode *fp_rootdir; /* apun. al directorio raíz
                                  del SA */
     struct filp  *fp_filp[20]; /* tabla de apun. a ranuras
                                  descriptores de archivos*/
     int fp_realuid;  /* id. real de usuario */
     int fp_effuid;   /* id. efectiva de usuario */
     param *fp_bufp;  /* parámetros del proceso cuando se
                         desbloquea por uso de pipes */
     char fp_suspend; /* indica si proceso suspendido */
     char fp_revived; /* indica si hay q' revivir proceso */
     char fp_task;    /* en qué tarea se suspendió el proc*/
}
```

Los campos *fp_workdir y *fp_rootdir apuntan a los nodos-i de los directorios raíz y actual respectivamente, cuando un usuario se cambia de directorio, lo que cambia es el apuntador fp_workdir al ahora directorio actual. Ambos apuntadores sirven también para saber en qué nodo-i inicia la búsqueda de un archivo para el que se ha dado una trayectoria: si es absoluta comienza en el nodo-i apuntado por fp_rootdir, si es relativa en el apuntado por fp_workdir.

Cada entrada en la tabla de procesos hace referencia, por medio de un arreglo de apuntadores a otra tabla: la de archivos abiertos. La utilidad de ésta tabla consiste en mantener la posición actual y el apuntador al nodo-i de cada archivo abierto. Es por medio de los apuntadores a la tabla de archivos abiertos, por cuales es posible que dos procesos compartan un archivo. Por ejemplo: un proceso abre un archivo, escribe en él, crea otro proceso que debe escribir información adicional en el archivo, al que hereda explícitamente el descriptor del archivo, e implícitamente el apuntador a la ranura en la tabla de archivos abiertos; éste proceso-hijo escribe en el archivo usando la posición de la ranura del archivo abierto a la que ambos tienen referencia. Con ese

esquema la información del proceso-hijo no se sobrepone en el comienzo del archivo, sino que escribe a partir de la última escritura que hizo el proceso-padre, y cuando la escritura del proceso-hijo termina, el proceso-padre puede continuar escribiendo a partir de donde el proceso-hijo terminó. La figura 4.1 muestra más claro lo anterior.

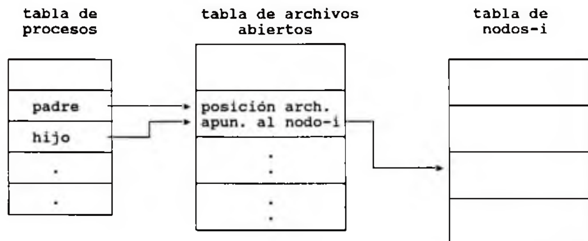


Figura 4.1 Compartición de un archivo entre procesos padre-hijo.

La tabla de archivos abiertos tiene 64 ranuras, lo que significa que puede haber 64 archivos abiertos en el sistema, pero cada proceso sólo puede accesar a 20 de esas ranuras. Veamos la definición de una ranura de archivo abierto.

```
struct filp{
     unsigned int  filp_mode; /* bits RW indican cómo se
                                 abre el archivo */
     int filp_count;           /* cuantos descriptores de
                              archivos comparten la ranura */
     struct inode *filp_ino;  /* apun. al nodo-i */
     long filp_pos;           /* posición del archivo */
     int file_lock;           /* indica si el archivo esta
}                                bloqueado para lec/escr */
```

El uso de un archivo se lleva a cabo por su identificación en memoria: el descriptor del archivo; como ya vimos cada proceso puede tener hasta 20 apuntadores a ranuras de archivos abiertos, el descriptor de archivo actúa como índice en la tabla de apuntadores, el cual no tiene nada que ver como índice en la tabla de archivos abiertos, cuyas ranuras se asignan según se encuentren disponibles.

La última instrucción en cualquier proceso es un ")", al momento de crear un proceso se sabe el tamaño del código que lo forma, así que dicha instrucción es reemplazada por la dirección de un procedimiento que llama a otro el cual se encarga de eliminar el proceso de la tablas, así como de liberar el código correspondiente si procede. El procedimiento que lo hace es kill(), el que también eliminar al proceso de la tabla de procesos del NSAX.

4.3 La tabla de dispositivos.

El SA original de XINIX tenía solo 4 identificadores para archivos los cuales se trataban como identificadores de dispositivos cuando se ejecutaba un servicio que los requerían (p.ej. putc(id_dispositivo, carácter)), con el nuevo SA habiendo tantos descriptores para archivos, y para mantener el mismo procedimiento de funciones comunes, se eliminaron los 4 identificadores y se decidió dejar sólo uno, por medio del cual se pudiera saber que la función requerida trabaja sobre un servicio del NSAX; en general la tabla de accesa por cualquiera de los siguientes identificadores de dispositivos ó archivos:

CONSOLE	0
OTHER_1	1
OTHER_2	2
DISK0	3
DISK1	4
FILE1	5

De ésta manera los descriptores de archivos devueltos al usuario nunca serán menores que 5, pues de 5 en adelante es como se identifica a los archivos y con éstos a las funciones que se requieren.

De los servicios que se accesan usando la tabla de dispositivos (devtab[] ver archivo xconf.c en fuentes de XINIX), para el SA sólo se usan: close, read, write, seek, getc, putc. Sería lógico que la función open también se usara, pero en el NSAX para abrir un archivo se necesita el nombre y modo del archivo, y es ésta función la que devuelve el descriptor del archivo con el que se pueden usar los otros servicios.

4.4 La interfaz del sistema de archivos.

El sistema de archivos es un proceso, el uso de los servicios que ofrece se lleva a cabo por medio de un **puerto**. Un puerto es un medio de comunicación (y coordinación) entre procesos por medio de mensajes. Un puerto se crea con el servicio **pcreate(nm)**, que recibe el número máximo de mensajes que el puerto puede tener a la vez. Pcreate() devuelve la identificación del puerto. Un proceso recibe un mensaje de un puerto con el servicio **preceive(portid)**; si existe el mensaje preceive() regresa enseguida, devolviendo el mensaje en el nombre de la función, en caso contrario, preceive() bloquea al proceso que lo llamó. Los mensajes a un puerto se envían con el servicio **psend(portid, mensaje)**. Si hay espacio, psend() deposita el mensaje y regresa enseguida, en caso contrario bloquea al proceso que lo llamó, el cual se desbloqueará cuando otro proceso obtenga un mensaje, del mismo puerto con preceive().

El sistema de archivos, utiliza un puerto de mensajes, para recibir las peticiones de un servicio. Este puerto tiene capacidad para diez mensajes, cada uno de los cuales es una estructura que tiene los parámetros necesarios para la realización del servicio, la identificación del proceso que lo pide y la identificación del

servicio.

El procedimiento-servicio que se llama desde un programa de usuario no es realmente el que lo realiza, sino un procedimiento que con los parámetros que recibe, forma y envia un mensaje. Veamos la definición de un mensaje.

```
struct mensaje{
            int    lock;
            int    addr;
            char   *buffer;
            int    cd_flag;
            int    co_mode;
            int    eff_grp_id;
            int    eff_usr_id;
            int    dev;
            int    fd;
            int    fd2;
            int    group;
            int    real_grp_id;
            int    ls_fd;
            int    mk_mode;
            int    mode;
            char   *name;
            char   *name1;
            char   *name2;
            char   *name3;
            int    name_length;
            int    name1_length;
            int    name2_length;
            int    nnbytes;
            long   offset;
            int    owner;
            int    parent;
            char   pathname[14];
            int    pro;
            int    rd_only;
            int    real_usr_id;
            int    request;
            int    whence;
            int    who;
            int    id_servicio;
}
```

Este mensaje contiene todos los parámetros se utilizan todos los servicios; sin embargo, un servicio en particular usa sólo los que

necesita. Veamos ahora como se define un puerto y un ejemplo de cómo es un procedimiento que prepara un mensaje (la petición real) para el NSAX.

```
/* Creación del puerto de mensajes para el  SA */

int flmsgport,  flmsgbufp;

flmsgport = pcreate(10);
flmsgbufp = mkpool(10 * sizeof(struct mensaje), 10,
                   NCHK64KB);   /* pool de mensajes */
```

```
int fread(fd, buff, numbytes)  /* prepara mensaje para el*/
int fd, numbytes;              /* SA, solicitan serv.READ*/
char *buff;
{

    struct mensaje bufp;
    int stat;

    bufp = (struct mensaje *) getbuf(flmsgbufp);
    bufp->fd = fd;
    bufp->buffer = buff;
    bufp->nnbytes = numbytes;
    bufp->whos = getpid();
    bufp->id_servicio = READ;

    recvclr();
    psend ( flmsgport, bufp);  /* envía el mensaje al SA */
    stat = receive();
    freebuf(bufp);
    return(stat);
}
```

Cuando NSAX recibe un mensaje, el campo bufp->id_servicio indica a qué procedimiento debe llamar, y le pasa la dirección del mensaje como único parámetro: ahí van los parámetros reales. El procedimiento al que se llamó devuelve alguna respuesta al NSAX, y éste a su vez envía el mensaje de respuesta (se recibe en stat en el cuadro superior) al procedimiento que originalmente había

sido llamado por el usuario.

4.6 Manejadores de terminal y de disco.

La modificación en estos manejadores son principalmente detalles de implantación, cómo y para qué fueron hechos ayudará a quién se interese en hacer algún otro manejador de dispositivo o nivel lógico para XINIX.

En lo que al manejador de disco se refiere, se hizo una reestructuración de la versión original; aún cuando realiza las mismas funciones, ahora el programa es más legible; se eliminaron ciertas funciones que eran llamadas como de "control" (ver definición de devtab[] archivo xconf.c en los fuentes de XINIX) para el manejador de disco, pues la estructura del sistema de archivos original no las proveía como tales, por lo que fueron implantadas dentro de este manejador.
En la versión original del manejador de disco, y por el diseño del SA anterior los bloques físicos y lógicos eran de 512 bytes, así que las transferencias que hacía el manejador de disco eran del mismo tamaño y no había problema cuando se trasladaban archivos DOS a XINIX, pues el tamaño de las transferencias de disco para ambos SO's era el mismo: 512 bytes. Con el nuevo NSAX las transferencias se hacen de 1024 bytes (el tamaño de dos bloques físicos en disco), y cuando se trasladan archivos DOS se necesitan de 512 bytes así que en cada petición al manejador de disco se incluye el tamaño de la transferencia. De esta manera se puede indicar dinámicamente al manejador de disco 4 tamaños de transferencias: 512, 1024, 2048, 4096 bytes.

Modificaciones al manejador de terminal como tales no hubo, pero se añadió una pequeña inicialización, explico primero el motivo. Cuando XINIX comienza a ejecutarse crea los procesos de los

intérpretes de comandos para cada terminal; estos procesos shell eventualmente se convierten en los procesos "padres" de los programas que ejecute el usuario, y a su vez "abuelos" de los procesos que sean creados dentro de esos programas. Se supone que para cada proceso su ancestro más antiguo es el intérprete donde comenzó todo, y que es éste quién se encarga de heredar ciertos atributos a sus hijos, entre ellos la identificación de la terminal que controla, la cual viene a ser la identificación de los dispositivos stdin, stdout, stderr de cada proceso hijo. Sin embargo, cuando un proceso utiliza **printf()** para imprimir resultados, éstos se imprimian siempre en la terminal de memoria mapeada, la CONSOLA, aún cuando el proceso que escribe tenga asignada otra terminal como **stdout**. El problema era que al iniciar los procesos shells de las terminales se "olvidó" actualizar la identificación real de la terminal asociada a cada proceso shell. Ya que printf() la buscaba en la tabla de procesos considerando la identificación del proceso actual que ejecutaba. En pocas palabras, stdin, stdout, stderr, se encuentran en la entrada que corresponde en la tabla de procesos, más al estar incorrecta la identificación de tales dispositivos en el shell de una terminal, todos los hijos la heredaban mal. Puede parecer trivial, pero son detalles que toman tiempo, y espero que escribir esto aqui ayude un poco a quienes modifiquen a XINIX, por lo menos para entender porque suceden ciertas cosas. La mencionada inicialización se hizo en ttyinit() del manejador de terminal archivo x2ttyio.c de fuentes de XINIX.

4.7 Breve descripción funcional de los servicios del SA.

En este trabajo, no se incluyen los listados de los programas, sin embargo éstos y el resto de los fuentes de XINIX se encuentran en diskettes que están disponibles en la coordinación académica de la Sección de Computación en el CINVESTAV-IPN. En esta parte se

explica el funcionamiento de los servicios y se indica el archivo en donde se encuentra cada servicio, y cuando se haga referencia a alguna función que no se encuentre en el mismo archivo será indicado.

En general, para todo servicio se usan trayectorias, las funciones eat_path(), search_dir(), get_name(), advance(), y last_dir(), se encargan de transformar una trayectoria en el nodo-i del archivo deseado, de buscar un archivo en un directorio, incluir un archivo en un directorio, ó borrar un archivo de un directorio, etc. Dichas funciones están en el archivo xrpath.c. Los servicios por su naturaleza se pueden dividir en 4 tipos:

Servicios que accesan archivos existentes.
open(), read(),vwrite(), getc(), putc(), seek(), close().

Servicios que crean nuevos archivos ó directorios.
creat(), mknod().

Servicios que manipulan al sistema de archivos.
chdir(), chroot(), chmod(), stat(), fstat().

Servicios que crean referencias entre archivos o SA.
dup(), mount(), umount(), link(), unlink().

Open(). EL servicio open() abre un archivo y devuelve el descriptor de archivo para ese archivo. Open() primero llama a eat_path() usando el nombre ó trayectoria del archivo como parámetro, eat_path() devuelve el apuntador al nodo-i de dicho archivo; después se llama a get_fd() función que asigna el descriptor de archivo al archivo que se intenta abrir; después, usando el descriptor como índice se asigna una ranura de la tabla de archivos abiertos, de la cual un apuntador tiene ahora la

referencia del nodo-i en memoria del archivo. Se retorna el descriptor del archivo. Open() se encuentra en el archivo xropen.c.

Read(). El servicio read() lee una cantidad especificada de bytes de un archivo, y retorna la cantidad real de bytes leidos. Al iniciar se verifica que el descriptor del archivo sea válido, si es un número muy grande de bytes a leer se divide en grupos de transferencias que no pasen de 1 bloque lógico (1024 bytes). Las peticiones de lectura se hacen al buffer cache, si se encuentran ahí se reduce copiar datos entre buffers en memoria, de otra manera el buffer cache hace peticiones a disco, se completan, y después se transfieren al buffer del usuario, esto se realiza tantas veces como sea necesario para satisfacer la petición ó hasta encontrar fin de archivo. Read() está en el archivo xrread.c.

Write(). El servicio es similar a read(), sólo que para escritura; además, cuando ya no existe el bloque para el offset actual en el archivo se manda pedir un "bloque nuevo", los datos a escribir se ponen en ese bloque que ahora es parte del archivo. Si el valor que devuelve no corresponde con el número de bytes que se pidió escribir es porque algún error ocurrió. Se encuentra en el archivo xrwrite.c.

Seek(). El servicio seek() permite al usuario hacer acceso aleatorio a un archivo, pues éste manipula el offset del archivo. Se verifica que el descriptor se válido, en ese caso se obtiene la ranura correspondiente de la tabla de archivos abiertos y se actualiza el offset del archivo ahi y en su nodo-i en memoria. Se encuentra en el archivo xropen.c

Close(). Cierra un archivo abierto identificado por su descriptor. Se verifica la validez del descriptor, y se obtiene la ranura correspondiente como archivo abierto y se decrementa el contador de referencias a ese archivo abierto, si el contador es cero se manda a escribir su nodo-i a disco y se libera la ranura que tenía asignada el archivo. Se encuentra en el archivo xropen.c.

Creat(). Crea un archivo nuevo y retorna su descriptor de archivo quedando el archivo listo para escrituras. Se asigna un nodo-i para el archivo, implica actualizar el mapa de bits de nodos-i, y un bloque de nodos-i, después se asigna un descriptor de archivo y usandolo como indice se asigna una ranura de la tabla de archivos abiertos, de la cual un apuntador tiene ahora la referncia del nodo-i en memoria del archivo. Retorna el descriptor del archivo. Se encuentra en el archivo xropen.c

Mknod(). Como creat() crea archivos normales, mknod() crea archivos especiales, los que identifican a dispositivos. El archivo se crea igual que uno normal excepto por el modo, por la identificación del dispositivo, y porque estos archivos no usan bloques de datos. Se encuentra en el archivo xropen.c.

Chdir(). Cambia el directorio actual de trabajo. Se convierte la trayectoria del nuevo directorio o el nombre del nuevo directorio en un apuntador al nodo-i, se libera la referencia anterior y se toma este apuntador. La referencia la tiene la entrada del proceso del shell asociado con la terminal donde se hizo chdir(). Se encuentra en el archivo xrstadir.c.

Chroot(). El servicio chroot() permite asignar un nuevo directorio raiz para el proceso que hace el llamado. Es similar a chdir() excepto por que en vez de cambiar el directorio actual se cambia el directorio raiz. Después de haber hecho chroot las

trayectorias absolutas comenzarán en el nuevo directorio raiz y no en "/". Se encuentra en el archivo xrstadir.c.

Chmod(). Permite cambiar los permisos de acceso a un archivo. Se transforma la trayectoria ó el nombre del archivo, en un apuntador a un nodo-i, se cambia el modo, y se manda escribir el nodo-i a disco. Esta en el archivo xrprotect.c.

Stat() y Fstat(). Retornan la información que describe a un archivo. Stat() usa el nombre ó trayectoria del archivo y fstat() usa el descriptor del archivo. Con cualquiera de ellos se obtiene un apuntador al nodo-i del archivo, y se copia el contenido del nodo-i en el buffer del usuario. Se encuentra en el archivo xrstadir.c.

Dup(). Copia un descriptor de archivo y devuelve el nuevo, con lo que se tienen 2 descriptores para el mismo archivo. Se asigna un descriptor y obtiene una ranura libre en la tabla de archivos abiertos; y además se copian las mismas referencias del descriptor dado, y se incrementa el contador de referencias de la nueva ranura. Se encuentra en el archivo xrmixtos.c.

Mount(). Este servicio permite incorporar un SA en disco al SA raiz. Usa el nombre del dispostivo (drive) donde está el SA de disco y el directorio en donde se montará. Se transforma el nombre del dispositivo en su identificador numérico, se busca una ranura libre en la tabla de superbloques, se verifica que no se encuentre ningún sistema montado en ese dispositivo, se carga el superbloque del SA de disco, se verifica que el directorio en que se va a montar esté vacío, se cargan los mapas de bits del SA de disco, y se asignan los apuntadores de nodos-i a los campos correspondientes del superbloque. Se encuentra en el archivo xrmount.c.

Umount(). Elimina la manera de referir archivos en disco. Solo necesita el nombre del dispositivo en donde esta el SA en disco que se desea desmontar. Se transforma el nombre del dispositivo en su identificador númerico se verifica no haya ningún nodo-i en éste dispositivo que tenga referencias pues si las hay significa que **hay uno ó más procesos usando ese SA y no será posible desmontarlo**, sino se eliminan las referencias del superbloque en memoria y este a su vez es eliminado de la tabla de superbloques. Se encuentra en el archivo xrmount.c.

Link(). Crea una nueva entrada en un directorio para un archivo ya existente. Recibe como parámetros el nombre del archivo existente y el nombre de la nueva referencia. El nombre del archivo existente se traduce a un apuntador a un nodo-i en memoria, se verifica que exista el directorio donde se desea la nueva referencia del archivo, si existe se hace el enlace: se crea la entrada con el nuevo nombre en el directorio indicado usando el nodo-i del archivo existente, y se incrementa el contador de enlaces del nodo-i. Se encuentra en el archivo xrlink.c.

Unlink(). Hace la función inversa de link(): elimina una entrada de directorio para un archivo. Usa como parámetro el nombre del archivo que se desea eliminar. Verifica que exista el directorio en el que se encuentra el archivo, se elimina la entrada de ese directorio, y se decrementa el contador de enlaces al respectivo nodo-i. Se encuentra en el archivo xrmount.c.

Apéndice A

Uso de los servicios del sistema de archivos en programas de usuario

Este es el manual del usuario del sistema de archivos. El formato en que se presentan los servicios es el siguiente:

servicio()	servicio

función general

Sinopsis
Tipo de parámetros que recibe.

Descripción
Qué hace en detalle, qué regresa, sugerencias, etc.

Ver también
Servicios relacionados.

En general, el parámetro "nombre_archivo" puede ser una trayectoria que incluya el nombre del archivo deseado. En el archivo "xinixh.h", que se debe incluir en todo programa de usuario, existen unas macros que facilitan el uso de los servicios, pues sirven de parámtros bastante significativos.

access()	access

Verifica que un archivo pueda ser accesado según el modo.

Sinopsis:

```
#include "xinixh.h"

char *nombre_archivo;
int modo;

int access(nombre_archivo, modo)
```

Descripción:

Se verifica que sea válido el intento de acceso a un archivo dependiendo del modo. EL modo puede ser lectura, escritura, ó ambos, pueden usarse las macros M_READ, M_WRITE, M_RW respectivamente. Devuelve OK si se permite el acceso, SYSERR en caso contrario, ó si hubo algún error, un valor negativo que lo identifica, ver el apéndice C de errores.

Ver también: chmod().

chdir()	chdir

Cambia de directorio al proceso que lo ejecuta.

Sinopsis:

```
#include "xinixh.h"

char *trayectoria_dir;

int chdir(trayectoria_dir)
```

Descripción:

Cuando un proceso es creado tiene acceso por default al directorio en el que se creó, si éste proceso llama a chdir(), entonces el proceso tendrá acceso default al directorio que se acaba de cambiar. Devuelve OK si pudo hacerlo, SYSERR ó algún error en caso contrario, ver apéndice C de errores.

Ver también: chroot(), mkdir(), mknod().

chmod()	chmod

Cambia los permisos de acceso a un archivo.

Sinopsis:

```
#include "xinixh.h"

char *nombre_archivo;
int modo;

int chmod(nombre_archivo, modo)
```

Descripción:

Chmod() cambia el modo de un archivo, el usuario debe conocer muy bien como se construye un modo (ver capítulo 3 nodos-i). Devuelve OK si todo fue correcto, SYSERR u otro valor negativo en caso contrario (ver apéndice C).

Ver también: creat(), access(), stat(), fstat().

chroot()	chroot

Cambia el directorio raiz de un sistema de archivos en disco.

Sinopsis:

```
#include "xinixh.h"

char *trayectoria;   /* del directorio que será la raíz */

int chroot(trayectoria)
```

Descripción:

Chroot() cambia el directorio raíz de un sistema de archivos en disco, puede ser útil cuando se usan trayectorias demasiado largas, si tienen un directorio común después del raíz puede acortarse usando éste servicio. Devuelve OK si todo fue correcto, SYSERR en caso contrario (ver apéndice C).

Ver también:

chdir(), access(), stat(), fstat(), mount(), mkdir().

close()	close

Cierra un archivo.

Sinopsis:

```
#include "xinixh.h"

int descriptor_arch;

int close(descriptor_arch)
```

Descripción:

Cierra un archivo, no se pueden hacer más operaciones de lectura y/o escritura una vez que se ha cerrado el archivo, libera una ranura en la tabla de archivos abiertos en memoria. Devuelve OK si todo fue perfecto, SYSERR u otro valor negativo en caso contrario (ver apéndice C).

Ver también:

open(), creat(), read(), write(), seek().

creat()	creat

Crea un archivo.

Sinopsis:

```
#include "xinixh.h"

char *nombre_archivo;
int modo;

int creat(nombre_archivo,modo)
```

Descripción:

Crea un archivo nuevo y lo deja abierto para escritura, si el usuario desea evitarse complicaciones en cuanto al modo de creación del archivo puede usar la macro F_REGULAR como parámetro de modo; dicha macro es el modo para un archivo normal (ver el archivo include "xinixh.h"). Si el archivo ya existía se liberan los bloques de datos asignados anteriormente: se pierden los datos que tenían; se inicializa como nuevo. Devuelve un entero positivo: el

descriptor de archivo, con el cual se podrá leer y escribir del archivo. NO confundir con el servicio create(), que crea un proceso.

Ver también:

open(), read(), write(), seek(), close(), dup(), dup2().

dup()	dup

Duplica un descriptor de archivo.

Sinopsis:

```
#include "xinixh.h"

int desc_arch;

int dup(desc_arch)
```

Descripción:

Duplica un descriptor de archivo: para tener dos descriptores de archivo a uno mismo. Para que el archivo no tenga ninguna referencia como abierto deberá cerrarse tantas veces como se haya duplicado su descriptor, usando como parámetro los descriptores duplicados. Retorna el nuevo descriptor del archivo si no hubo ningún error, en caso contrario SYSERR ó algún valor negativo (ver apéndice C).

Ver también:

dup2(), open(), creat(), read(), write(), close().

dup2()	dup2

Duplica un descriptor de archivo en un entero no asignado.

Sinopsis:

```
#include "xinixh.h"

int fd1, fd2;

int dup2(fd1, fd2)
```

Descripción:

Duplica un descriptor de archivo, con la particularidad de que es el usuario el que escoge el nuevo descriptor (un entero), al que todavía no se le ha asignado un archivo. La llamada hace de fd2 un descriptor de archivo válido del mismo archivo que fd1. Devuelve el descriptor de archivo válido, si está bien, en caso contrario SYSERR ó algún error (ver apéndice C).

Ver también:

dup(), open(), creat(), read(), write(), close().

fstat()	fstat

Retorna la información que describe a un archivo.

Sinopsis:

```
#include "xinixh.h"

int desc_arch;
struct stat buffer;

int fstat(desc_arch, buffer)
```

Descripción:

La información en un nodo-i acerca del archivo que se encuentra abierto y cuyo descriptor de archivo se usa en éste servicio como parámetro, es vaciado en "buffer", la estructura "stat" está definida en "xinixh.h". Retorna OK si todo fue correcto, SYSERR ó un valor negativo en caso contrario (ver apéndice C).

Ver también: stat().

getc()	getc

Lee el siguiente carácter en un archivo.

Sinopsis:

```
#include "xinixh.h"

int desc_arch;

int getc(desc_arch)
```

Descripción:

Getc() lee el siguiente carácter del archivo identificado con el descriptor. Si todo fue correcto, getc() devuelve el carácter leído, sino se encontró fin de archivo ó hubo algún error.

Ver también:

putc(), read(), write().

link()	link

Enlaza un archivo existente con uno nuevo.

Sinopsis:

```
#include "xinixh.h"

char *arch_exs;        /* nombres de archivos ó trayectorias */
char *arch_nuev;

int link(arch_exs, arch_nuev)
```

Descripción:

Crea una nueva entrada con diferente nombre, de un nodo-i perteneciente a un archivo ya existente. Se puede accesar el mismo archivo por diferentes nombres. Devuelve OK si todo fue correcto, SYSERR u otro valor en caso contrario (ver apéndice C).

Ver también:

creat(), unlink(), dup(), dup2().

mkdir()	mkdir

Crea un directorio.

Sinopsis:

```
#include "xinixh.h"

char *trayectoria;     /* nombre del directorio a crear y/o
                                  trayectoria */
int mkdir(trayectoria)
```

Descripción:

Crea un directorio, en realidad es un procedimiento que usa los servicios mknod() y link(), para construirlo. Devuelve OK si todo fue correcto, ó un valor negativo si hubo algún error (ver apéndice C).

Ver también:

chdir(), chroot(), mknod(), link(), rd(), unlink().

mknod()	mknod

Crea un archivo especial.

Sinopsis:

```
#include "xinixh.h"

char *nom_arch;
int modo;
int ident;

int mknod(nom_arch, modo, ident)
```

Descripción:

Crea un archivo especial, en este caso son: archivos que identifican a los dispositivos, el usuario debe saber perfectamente como construir el modo adecuado (ver capítulo 3 nodos-i), "ident" es el identificador del dispositivo en particular, consultar "xinixh.h". Retorna OK si todo fue correcto, un valor negativo en caso contrario (ver apéndice C).

Ver también:

mkdir(), creat().

open()	open

Abre un archivo ya existente.

Sinopsis:

```
#include "xinixh.h"

char *nombre_archivo;                   /* trayectoria */
int modo;
int bloqueado;

int open(nombre_archivo, modo, bloqueado)
```

Descripción:

Open() devuelve el descriptor del archivo, con el cual se hacen las operaciones de lectura, escritura y posicionamiento. El modo en que se abre puede ser lectura, escritura, ó ambas las macros M_READ, M_WRITE, M_RW, los definen y hacen más fácil el uso del modo, no se debe confundir el usuario con el modo de creación que se usa en el servicio creat(), en este caso el archivo debe existir, sino devuelve SYSERR (ver apéndice C). El parámetro "bloqueado" lo usa un usuario en su proceso cuando quiere bloquear el archivo según lo necesite: si es para escritura, nadie más puede accesarlo; si es para lectura, sólo procesos que lo quieran leer podrán accesarlo, si son escritores no entrarán, sino hasta que los lectores acaben. Las macros LOCK_R, LOCK_W, NO_LOCK usadas como parámetro "bloqueado" indican que el archivo se bloqueará para lectura, escritura, ó no bloqueo respectivamente. Cuando se usen bloqueos se sugiere poner el open en un ciclo tantas veces como el usuario desee el número de intentos que se hagan por asegurar el bloqueo del archivo que se intenta abrir, y se tomen las acciones correspondientes cuando no haya sido posible abrir el archivo con la condición de bloqueo deseada. Cuando un archivo es abierto para lectura ó para lectura y escritura, el apuntador al archivo queda en el inicio; cuando es abierto para escritura el apuntador queda al final del archivo, esto es para seguridad de la información que ya había en el archivo, y para que sea el usuario el que decida cuando debe sobreescribir en él.

Ver también:

creat(), mknod(), close(), read(), write(), getc(), putc(), seek().

putc() putc

Escribe un carácter en un archivo.

Sinopsis:

```
#include "xinixh.h"

int desc_arh;
char c;

int putc(desc_arch, c)
```

Descripción:

Escribe el carácter c en el archivo identificado con ese descriptor. Retorna OK si todo fue correcto, valor negativo en caso de error (ver apéndice C).

Ver también:

getc(), read(), write(), open(), creat().

rd() rd

Borra un directorio.

Sinopsis:

```
#include "xinixh.h"

char *nombre_dir;            /* ó trayectoria */

int rd(nombre_dir)
```

Descripción:

Borra un directorio, éste tiene que estar vacío. En realidad, este servicio está formado por otros: fstat(), access(), unlink(), open(), read(), close(), ver archivo xrinter.c de fuentes de xinix. Retorna OK si pudo hacerlo, un valor negativo en caso contrario (ver apéndice C).

Ver también:

unlink(), rm().

read()	read

Lee uno ó más carácteres de un archivo.

Sinopsis:

```
#include "xinixh.h"

int desc_arch;
char *buffer;            /* ó equivalente */
int num_chars;

int read(desc_arch, buffer, num_chars)
```

Descripción:

Read() lee hasta "num_chars" carácteres del archivo identificado con ese descriptor. Si hay algún error se devuelve SYSERR ó un valor negativo (ver apéndice C). Sino, devuelve el número de carácteres leídos, ó los disponibles antes del fin del archivo, los deja en "buffer".

Ver también:

getc(), putc(), write(), creat(), open(), seek().

rm()	rm

Borra un archivo.

Sinopsis:

```
#include "xinixh.h"

char *nombre_archivo;       /* trayectoria */

int rm(nombre_archivo)
```

Descripción:

Borra un archivo del directorio actual ó del último de la trayectoria dad. Este servicio es equivalente a unlink(). Devuelve OK si pudo borrarlo, error en caso contrario ver apéndice C.

Ver también:

creat(), mknod(), unlink(), link().

seek()	seek

Cambia la posición actual en un archivo abierto.

Sinopsis:

```
#include "xinixh.h"

int desc_arch;
long offset;
int apartir_de;

int seek(desc_arch, offset, apartir_de)
```

Descripción:

Cambia la posición actual del archivo identificado por el descriptor, hasta un desplazamiento "offset", tomando en cuenta desde "apartir_de", para éste último parámetro se puede usar una de 3 macros definidas: S_BEGIN, S_CURRENT, S_END, que indican respectivamente desde el inicio del archivo, desde la posición actual, desde el fin del archivo, ésto último es posible ya que si es elegido y el offset es mayor que cero, el espacio entre lo que era el fin del archivo y la actual posición queda vacía, y es en la posición actual en donde pueden escribirse datos; después el tamaño del archivo será el número total de bytes hasta el útlimo escrito por el usuario aunque para escribirlo haya dejado muchos "vacíos". Devuelve OK si todo fue correcto, algún error en caso contrario (ver apéndice C).

Ver también:

read(), getc(), putc(), write(), creat(), open(), seek(), close().

stat()	stat

Retorna la información que describe a un archivo.

Sinopsis:

```
#include "xinixh.h"

int *nombre_archivo;
struct stat buffer;

int stat(nombre_archivo, buffer)
```

Descripción:

La información del nodo-i del archivo se pone en buffer. La diferencia con fstat es que aquí se usa el nombre del archivo y allá el descriptor del archivo. Retorna OK si todo fue correcto, algún error en caso contrario (ver apéndice C).

Ver también:

fstat().

sync()	sync

Manda escribir a disco bloques que han sido modificados en el buffer cache.

Sinopsis:

```
#include "xinixh.h"

int sync()
```

Descripción:

Escribe en disco bloques del buffer cache que han sido modificados. Cuando el usuario use archivos en sus programas ó procesos y los modifique ó cree nuevos, usar sync() antes de terminar su código con ")" para asegurarse que la información que escribió sea grabada en disco y para evitar inconsistencias en el sistema de archivos.

unlink()	unlink()

Quita un enlace a un archivo.

Sinopsis:

```
#include "xinixh.h"

char *nombre_archivo; /* trayectoria */

int unlink(nombre_archivo)
```

Descripción:

Elimina un enlace de un archivo, si era el último enlace, el archivo se elimina del directorio en donde se encuentre.

Ver también:

link(), rm(), rd(), mkdir().

write()	write()

Escribe uno ó más carácteres en un archivo.

Sinopsis:

```
#include "xinixh.h"

int desc_arch;
char *buffer;              /* ó equivalente */
int num_chars;

int write(desc_arch, buffer, num_chars)
```

Descripción:

Write() escribe hasta "num_chars" carácteres de "buffer" en el archivo identificado con ese descriptor. Si hay algún error se devuelve SYSERR ó un valor negativo (ver apéndice C).

Ver también:

creat(), open(), read(), seek(), getc(), putc(), close().

Apéndice B

Uso de los comandos del Sistema de Archivos en el intérprete de comandos (shell)

Los comandos son instrucciones que el usuario da a XINIX en forma interactiva, através del intérprete de comandos, usando el teclado de una terminal. El formato siguiente explica la formación de cada comando:

CONSOLE>> **comando arg1, arg2 .....** **"comando"**

EL prompt desplegado depende de la terminal que se esté usando: "CONSOLE>>" si es la terminal de memoria mapeada; "OTHER_1>>" u "OTHER_2>>" si son terminales RS-232. Tome en cuenta que todos los comandos, excepto los informativos y **cat**, despliegan la manera de usarlos en caso de no enviarles los parámetros necesarios. En cada comando, los parámetros entre paréntesis cuadrados ([ ]) son opcionales. Cuando se dan parámetros incorrectos ó no se puede llevar a cabo un comando se imprime en la terminal un letrero indicando el código del error, generalmente un número negativo; el apéndice C explica cada uno de ellos.
En cualquier parte donde se indique el nombre de un archivo, o de un directorio, va implicita la posibilidad de sea una trayectoria que indique donde se encuentra el archivo o directorio correspondiente. Lo anterior es válido para comandos que ya existían y manipulaban archivos como **cat, cp, mv, rm.**

Comandos para el Sistema de archivos en el Intérprete

CONSOLE>> **chmod modo file** "chmod"

Cambia el "modo" con el que originalmente fue creado un archivo, el usuario debe saber cómo construir el "modo" y debe darlo en forma octal.

CONSOLE>> **cd nombre_directorio** "cd"

Cambia el directorio actual por el último encontrado en la trayectoria si es que la hay, sino al directorio indicado. Recuérdese las trayectorias relativas y absolutas.

CONSOLE>> **chroot nuev_direcraiz** "chroot"

Cambia el directorio raiz de un sistema de archivos montado. Sirve para no hacer referencia a archivos con trayectorias demasiado largas.

CONSOLE>> **date [mm-dd-aa]** "date"

Despliega la fecha, la cual se tuvo que haber dado anteriormente. Si no se dió ninguna, la fecha default es 1-1-1980. Después de desplegarla "pide" la nueva fecha por si se quiere actualizar, sino basta con dar <RETURN>.

CONSOLE>> **link old_file new_file** "link"

Crea una nueva refencia de "old_file" como "new_file", el mismo archivo es referido con ambos nombres.

CONSOLE>> **ls [-l]** "ls"

Despliega el contenido del directorio actual. "ls" despliega sólo los nombres de los archivos, la opción "-l" despliega los nombres de los archivos y sus atributos.

CONSOLE>> **mkdir nombre_directorio** "mkdir"

Crea un directorio dentro del directorio actual.

CONSOLE>> **mknod nombredispo b/c numdispo** "mknod"

Crea un archivo especial. Recuerde que un archivo especial identifica a los dispositivos, así que el parámetro "b/c" debe ser "b" si el dispositivo está orientado al manejo de bloques, p.ej. los drives; "c" si el dispositivo está orientado al manejo de carácteres, p.ej. las terminales. El usuario debe conocer como están identificados los dispositivos en el sistema operativo para que sea correcto el parámetro "numdispo".

CONSOLE>> **mount nombredispo nomdirectorio [rd_only]** "mount"

Incorpora un sistema de archivos de disco al SA raíz. En este caso "nombredispo" puede ser uno de los siguientes: **/dev/fdo** ó **/dev/fd1** según en el drive que se encuentre insertado el SA en disco que se quiere montar 0 ó 1 (A: ó B: en DOS). A su vez "nomdirectorio" indica el directorio ya existente y vacío en donde se montará, por medio del cual se puede accesar el SA en disco, en este caso **/usr_a** ó **/usr_b** se sugiere coincida la terminación de éste directorio con el drive en donde se encuentre el diskette. El parámetro "rd_only" sirve cuando el usuario desea montar un SA sólo de lectura. El usuario es responsable de mantener el disco en el drive donde lo montó, si lo retira sin haber hecho **UMOUNT,** e inserta otro pueden generarse serios problemas de inconsistencia

en los SA de ambos discos.

CONSOLE>> **rd nombre_directorio** "rd"

Borra un directorio. Este tiene que estar vacio.

CONSOLE>> **sync** "sync"

Manda escribir los bloques que han sido modificados desde la última vez que se escribió en disco. Se recomienda usarlo antes de desmontar un SA.

CONSOLE>> **time [hh:mm]** "time"

Despliega la hora, la cual toma de la cuenta que llevaba DOS cuando éste inició; después pide el nuevo tiempo si se desea actualizar, sino basta dar <RETURN>; si se actualiza, el cambio también será para DOS una vez que se retorne a él.

CONSOLE>> **umount nombredispo** "umount"

Desmonta el SA montado en "nombredispo". Se elimina la referencia entre el directorio donde se montó y el SA de disco. Ahora puede ser retirado el diskette del drive y montar otro SA de disco.

Apéndice C

Códigos de Errores

En XINIX y en su Nuevo Sistema de Archivos, cuando los servicios no se pudieron llevar a cabo regresan un valor que lo indica, XINIX en general usa la macro SYSERR (-1). El NSAX usa valores negativos más específicos para los errores más comunes, dichos valores son equivalentes en mensajes del intérprete y en programas de usuario.

Nombre Mnémonico	Error	Significado
OK	1	Terminación ó salida exitosa.
SYSERR	-1	Hubo algún error en el SO.
EPERM	-1	Un usuario intenta modificar un archivo que no es de su propiedad, ó al que sólo tiene acceso el dueño ó el superusuario.
ENOENT	-2	No se encuentra archivo ó directorio en el directorio actual. Ocurre cuando el nombre del archivo dado no corresponde a ninguno de los actuales, ó no existe algún directorio indicado en la trayectoria.
E2BIG	-7	Lista de argumentos demasiado larga.
EBADF	-9	Descriptor de archivo incorrecto. Ya sea que un descriptor se refiera a un archivo no abierto, ó que se haga una petición de lectura (escritura) en un

		archivo que fue abierto sólo para escritura (lectura).
EACCESS	-13	Permiso denegado. Se intenta accesar un archivo al cual no se tiene permiso de acceso.
ENOTBLOCK	-15	Se requiere la identificación de un dispositivo de bloque. p.ej. en mount.
EBUSY	-16	Dispositivo para montaje ocupado. Se hizo un intento de montar un SA en un dispositivo en donde ya se había hecho un montaje, ó se intenta desmontar un SA el cual se encuentra activo (archivos abiertos, usuarios en algún directorio, etc).
EEXIST	-17	El nombre de archivo ó directorio ya existe.
ENODEV	-19	No es válido el dispositivo. Se hizo un intento de aplicar un servicio inapropiado para el dispositivo.
ENOTDIR	-20	No es un directorio. No se especificó un directorio donde se requería, p.ej. en una trayectoria ó como argumento para chdir.
EISDIR	-21	Es un directorio. Se intenta escribir inadecuadamente en un directorio.
EINVAL	-22	Argumento(s) inválido(s). P.ej. desmontar de un dispositivo no montado, leer ó escribir en un archivo para el seek() generó un apuntador negativo.
ENFILE	-23	Sobreflujo en la tabla de archivos abiertos. La tabla de archivos abiertos está llena, y temporalmente no pueden aceptarse más "opens".

EMFILE	-24	Demasiados archivos abiertos. Sólo se permiten 20 archivos abiertos por proceso.
EFBIG	-27	Archivo demasiado grande. El tamaño del archivo excede de 350k (condicionado por el tamaño del dispositivo en este caso, discos flexibles).
ENOSPC	-28	No hay espacio disponible en el dispositivo.
ESPIPE	-29	Llamado ilegal a seek().
EROFS	-30	Sistema de archivos de sólo lectura. se hizo un intento por modificar un SA que se montó de sólo-lectura.
EMLINK	-31	Demasiados enlaces a un archivo. Intento de hacer más de 32767 ligas a un archivo.
EPIPE	-32	Conexión rota. Se intenta escribir en un pipe para el que no hay lectores.
E_BAD_CALL	-102	Se hizo una petición de un servicio inexistente.
E_LONG_STRG	-103	Trayectoria con caracteres extraños ó demasiado grande.

REFERENCIAS

[BACH86] Bach Maurice J.
THE DESIGN OF THE UNIX OPERATING SYSTEM.
Prentice Hall Inc. 1986.

[TANEN87] Tanenbaum Andrew S.
OPERATING SYSTEMS DESIGN AND IMPLEMENTATION.
Prentice Hall Inc. 1987.

[THOMP78] Thompson K.
UNIX IMPLEMENTATION.
Bell Sys. Tech. J., vol. 57 no.6 1978.

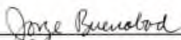
[RITCH78] Ritchie D. Thompson K.
THE UNIX TIME-SHARING SYSTEM.
Bell Sys. Tech. J., vol. 57 no. 6 1978.

[KERNG87] Kernighan B. Pike R.
THE UNIX PROGRAMMING ENVIRONMENT.
Prentice Hall Inc 1987.

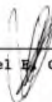
[COMER84] Comer Douglas.
OPERATING SYSTEM DESIGN THE XINU APPROACH.
Prentice Hall Inc. 1984.

[BUENB89] Buenabad Jorge.
XINIX SISTEMA OPERATIVO PARA COMPUTADORA PERSONAL.
Tesis de maestría. CINVESTAV 1989.

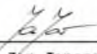
El jurado designado por la Sección de Computación del Departamento de Ingeniería Eléctrica, del Centro de Investigación y de Estudios Avanzados del I.P.N., aprobó esta tesis el 20 de abril de 1990.



M. en C. Jorge Buenabad Chávez



Dr. Manuel E. Guzmán Rentería



Dr. Jan Janecek Hyan

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITECNICO NACIONAL

BIBLIOTECA DE INGENIERIA ELECTRICA
FECHA DE DEVOLUCION

El lector está obligado a devolver este libro antes del vencimiento de préstamo señalado por el último sello.

- 1 OCT. 1990
18 OCT. 1990
05 NOV. 1990
12 NOV. 1990
2 NOV. 1990
6 DIC. 1990
28 MAR. 1991
30 NOV. 1995
13 DIC. 1995
26 ENE. 1996
22 NOV. 2000
24 OCT. 2002
- 6 ENE. 2003
ENE. 2003
15 OCT. 2003

DEVOLUCION

7/8-100-90L

AUTOR DELGADO MORENO, R.E.

TITULO SISTEMA DE ARCHIVOS PARA EL SISETMA OPERATIVO XINIX

CLASIF. XM 90.18 RGTRO. BI 11681

NOMBRE DEL LECTOR	FECHA PREST.	FECHA DEVOL.
David Leija Diaz	17/9/90	1/10/90
David Leija Diaz	4/X/90	5 DIC 90
David Leija Diaz	7/III/91	15/4/91
Maximino Peña Guerrero	8/Nov/9[illegible]	13/XII/[illegible]
Maximino Peña Guerrero	5/[illegible]	[illegible]
Juan M Correa Hdez	1/Nov[illegible]	
Victor J. Cerón [illegible]	3[illegible]	
Salvador Castro Cru[illegible]		
Rocío Gómez de F		
Salvador Castro[illegible]		
Roberto M		