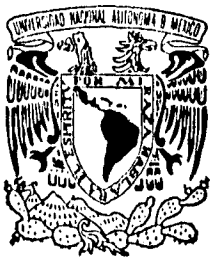
23
2Ej



UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

PORTABILIDAD DE SISTEMAS OPERATIVOS: UN CARGADOR PARA LINUX EN LA PLATAFORMA SPARC

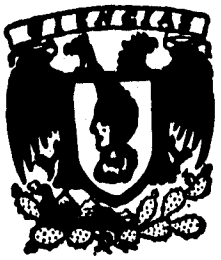
T E S I S

QUE PARA OBTENER EL TITULO DE:

M A T E M A T I C O

P R E S E N T A :

MAURICIO PLAZA VILLEGAS



TESIS CON
FALLA DE ORIGEN

TESIS

COMPLETA

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

M. en C. Virginia Abrín Batule
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
P r e s e n t e

Comunicamos a usted que hemos revisado el trabajo de Tesis:

Portabilidad de Sistemas Operativos:
Un cargador para Linux en la plataforma SPARC

realizado por
Mauricio Plaza Villegas

con número de cuenta 8836151-8 , pasante de la carrera de Matematicas

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	M. en C. Elisa Viso Gurovich	
Propietario	Dr. Mario Magidin Matluk	
Propietario	M. en I. María de Luz Gasca Soto	
Suplente	M. en C. Javier García García	
Suplente	M. en C. Manuel López Michelone	

Consejo Departamental de Matemáticas

A mi...

... familia, Ramon Plaza Mancera,

María Clemencia Villegas y

Ramon Plaza Villegas.

Portabilidad de Sistemas Operativos:
Un cargador para Linux en la plataforma SPARC

Mauricio Plaza Villegas

Abril-Mayo-Junio 1996

Índice

Introducción

El presente trabajo explicará la implementación y el funcionamiento de un cargador para Linux en la plataforma SPARC llamado **SPARC Linux Loader** (SILO). Para este efecto, primero se hará una muy breve introducción a sistemas operativos (capítulo I), sólo para familiarizarse con la terminología y detalles técnicos que fueron requeridos en la implementación del cargador. También se hará una breve descripción de la arquitectura SPARC donde se puntualizarán los elementos relacionados con el cargador.

En el capítulo II se hará una somera descripción del sistema operativo Linux y la importancia del proyecto Linux/SPARC. Este proyecto tiene como fin el portar el sistema operativo Linux (sistema nativo de los procesadores INTEL 386 y posteriores), a una arquitectura totalmente distinta como es la SPARC (procesadores RISC).

El resto de los capítulos se abocan a describir y explicar la implementación de SILO, los problemas que se enfrentaron y las soluciones que se plantearon. A SILO se le dió este nombre para tratar de mantener cierta compatibilidad con los nombres de otros cargadores para Linux en otras plataformas. El primer cargador para Linux en su plataforma nativa (PC) se llama LILO, que significa simplemente **Linux Loader**. Recientemente Linux fue portado a la arquitectura Alpha (procesadores MIPS), así que el cargador fue llamado MILO (**MIPS Linux Loader**). Y SILO, como se mencionó anteriormente, significa **SPARC Linux Loader**.

Capítulo 1

Nociones Básicas

1.1 Sistemas Operativos

1.1.1 Descripción General

Un sistema operativo es un conjunto de programas que administra los recursos de una computadora. Estos recursos son:

1. Memoria.
2. Registros.
3. Tiempo de procesador.
4. Dispositivos externos o periféricos.

Así, el sistema operativo es el responsable del movimiento, almacenamiento y proceso[1] de la información entre estos recursos y del control de las funciones mencionadas. En general, se tiene la idea de que un mecanismo de control y administración es totalmente externo al objeto controlado o administrado; en este caso, de que el sistema operativo es externo a los recursos. Sin embargo, el sistema operativo lleva a cabo sus funciones (o se ejecuta) en la misma forma que cualquier otro programa, es un programa más ejecutado por el procesador. Esto ocasiona que en algunas situaciones, el sistema operativo demande el control del procesador y deba esperar a obtenerlo casi como

[1] Literalmente: *movement, storage and procesing of data*[Stal].

cualquier otro programa.

El sistema operativo en sí, no es más que otro programa ejecutado por el procesador, estando la diferencia clave en la intención del sistema como programa, ya que éste dirige al procesador hacia el uso de otros programas y los recursos de la máquina, decide qué recurso está libre para cuál programa, cuánto tiempo de procesador le toca a cada proceso, qué memoria está libre, cuál se puede liberar, etc.

El sistema operativo puede ser uno o más programas, pero siempre hay uno distinguido que es la parte principal o medular del sistema mismo. Este programa controla la ejecución de aplicaciones y actúa como una interface entre el usuario y la computadora. El sistema operativo como una interface entre el usuario y la computadora, debe de proveer servicios en las siguientes áreas:

1. **Creación de programas:** El sistema operativo debe de proveer una variedad de servicios como editores y depuradores para ayudar en la creación de programas. Generalmente, estos servicios son en forma de utilerías que no necesariamente son parte del sistema operativo aunque son accesibles por medio de él.

2. **Ejecución de programas:** El sistema operativo debe realizar muchas tareas para ejecutar un programa, como llamar y colocar en memoria los datos y las instrucciones del programa, inicializar dispositivos de entrada y salida y la memoria.

3. **Acceso a los dispositivos de entrada y salida:** Cada dispositivo requiere de un conjunto particular de instrucciones y señales de control para su operación. El sistema operativo se encarga de las diferencias y sutilezas entre los distintos dispositivos para que los programadores piensen sólo en términos de funciones genéricas sencillas de lectura y escritura (`read, write`).

4. **Acceso controlado a archivos:** En el caso de los archivos, se debe tener no sólo el control para el dispositivo donde se encuentra el archivo (cinta, disco), sino el formato que tiene el archivo mismo; otra vez, el sistema operativo se encarga de estas pequeñas sutilezas. En los

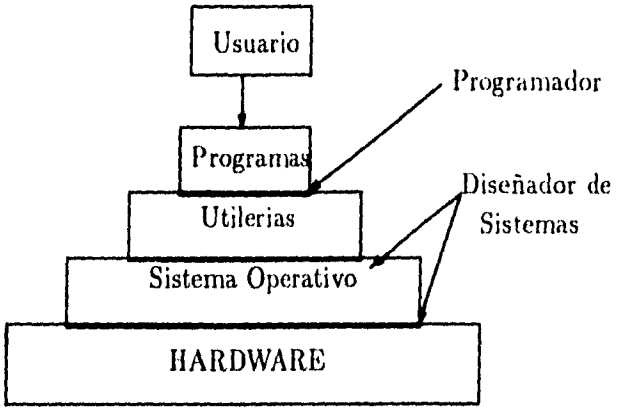


Figura 1.1: Visión general de un sistema operativo como una interface entre el usuario y la computadora

sistemas operativos multiusuario, el sistema operativo debe de proveer mecanismos de protección de la información.

5. **Acceso al Sistema:** En el caso de sistemas compartidos, el sistema operativo controla los accesos al sistema como un todo y algunos recursos en particular.

6. **Detección de errores:** El sistema operativo debe de contemplar cualquier eventualidad que pudiese ocurrir y reportarlo, desde fallas en el funcionamiento del *hardware* hasta fallas en el uso (y/o abuso) de los recursos de la máquina.

1.1.2 Sistemas UNIX

Uno de los sistemas operativos más usados, principalmente dentro de los ambientes universitarios es UNIX. Este sistema operativo se inició en una DEC PDP-7 desechada, en los laboratorios BELL durante 1969. Ken Thompson, con ideas y el apoyo de Rudd Canaday, Doug McIlroy, Joe Ossana y Denis Ritchie, escribió un sistema de tiempo real compartido y de uso general lo bastante adecuado y cómodo como para atraer a los usuarios entusiastas y con suficiente credibilidad para justificar la adquisición de una máquina más grande: una PDP-11. En 1973, Ritchie y Thompson escribieron el kernel

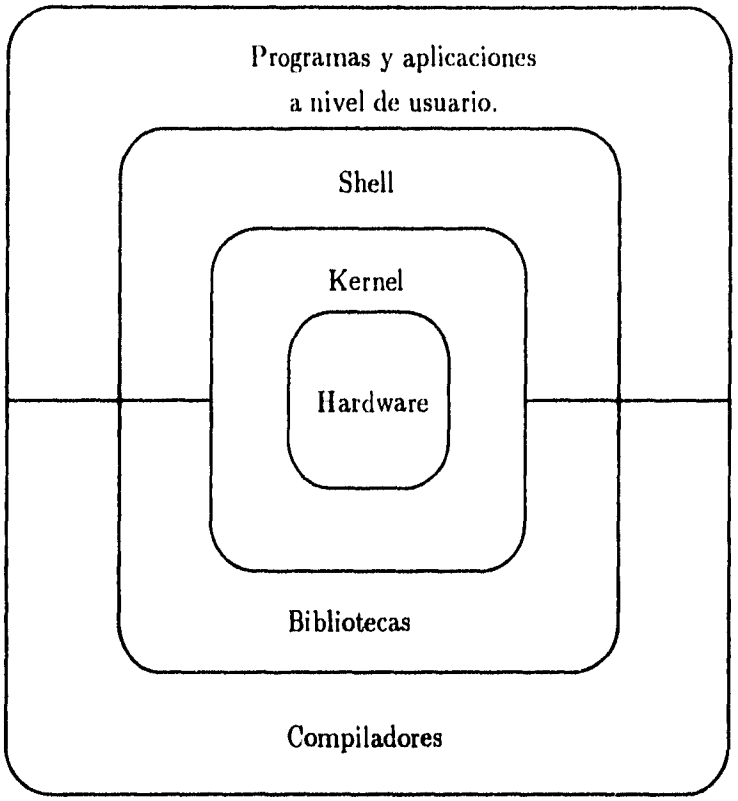


Figura 1.2: Descripción general de la arquitectura de un sistema UNIX

(núcleo) de UNIX en C, rompiendo así con la tradición de que los programas (*software*) que forman al sistema operativo (kernel) están escritos en lenguaje ensamblador.

El éxito de UNIX radica básicamente en que: primero, está escrito en C, o sea que es portátil (se ejecuta desde máquinas grandes como la CRAY hasta máquinas tan pequeñas como una PC). Segundo, el código fuente está disponible y escrito en lenguaje de alto nivel, lo cual lo hace fácil de adaptar a exigencias particulares. Por último, y esto es lo más importante, es un 'buen' sistema operativo, especialmente para los programadores. Su ambiente de programación es de extraordinaria riqueza y productividad.

Aún cuando UNIX introduce varios programas y métodos "innovadores" (para su época), su eficiencia no se debe a un programa o idea en particular, sino a su enfoque de la programación, una filosofía de cómo usar la computadora. Aunque esa filosofía no pueda describirse en una sola frase, se basa en la convicción de que el poderío de un sistema depende más de las relaciones entre los programas que de los programas mismos. Muchos programas de UNIX hacen aisladamente tareas triviales, pero al combinarse con otros se conviertenen en herramientas generales, útiles y poderosas.

La figura 1.2 describe la arquitectura general de una sistema UNIX. El hardware está cubierto en su totalidad por el núcleo o kernel del sistema operativo, este núcleo es la parte medular del sistema operativo (figura 1.3). Sin embargo, el sistema está equipado con un gran número de interfaces y servicios que son considerados parte del mismo sistema, como son los *shells*, componentes de la biblioteca de C, bibliotecas, etc.

La figura 1.3 nos da un acercamiento al núcleo del sistema operativo. Los programas a nivel de usuario invocan a servicios del sistema operativo por medio de "llamadas al sistema", o a través de una biblioteca. Esta interface de llamadas al sistema es la encargada de determinar qué tipo de recursos es el que se va a ocupar (memoria, disco), de ahí entra a un subsistema que trabaja con el recurso en sí, para finalmente trabajar con el recurso físicamente (hardware). El kernel del sistema operativo UNIX cuenta con las siguientes partes:

1. **Subsistema de archivos:** Es el encargado de intercambiar información entre la memoria y los dispositivos externos[2].

2. **Interface para llamadas al sistema:** Es la liga entre los programas a nivel de usuario y otros procesos mas específicos del kernel. Permite hacer llamadas de alto nivel que son mandadas a algún proceso de control más específico del kernel.

3. **Control del Hardware:** Es una serie de rutinas muy primitivas que accesan directamente el hardware de la máquina.

[2]Se hará una explicación más específica en la siguiente sección ya que es muy importante entender claramente cómo funciona el sistema de archivos, pues SILO basa su funcionamiento en el sistema de archivos ext2

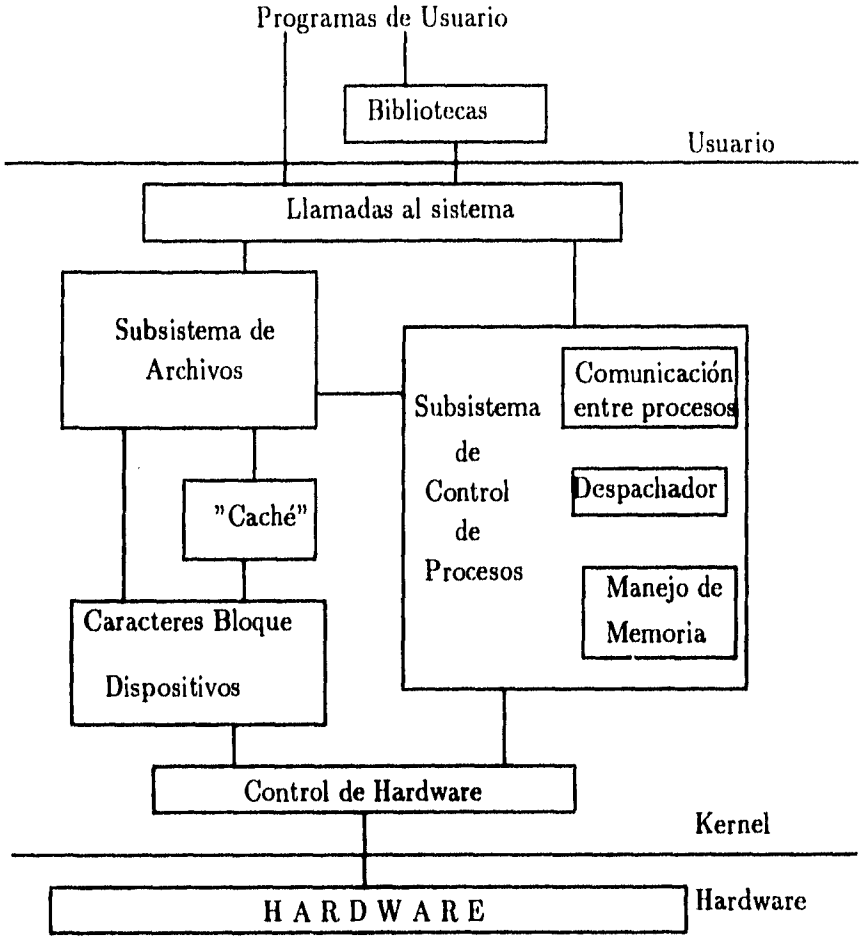


Figura 1.3: Diagrama en bloques del núcleo de UNIX

4. **Subsistema de control de Procesos:** Es el responsable del manejo de la memoria, calendarización y despacho de procesos, y de la sincronización entre los mismos.

1.1.3 Sistema de archivos

Al elemento básico y más pequeño de los datos le llamaremos **campo**. Un campo contiene uno y sólo un valor y su contenido está asignado por procesos o usuarios. Un **registro** es una colección de campos relacionados entre sí que son tratados como una unidad frente a programas o aplicaciones.

Un archivo es una colección de datos que es tratada como una sola entidad por los usuarios frente al sistema operativo. Teóricamente deben tener un nombre único de identificación mediante el cual, procesos, aplicaciones y/o usuarios hacen referencia a él. Se debe contar con operaciones básicas sobre éstos, como son:

1. **Obtener_todos**(*Retrieve_All*): Obtener todos los registros de un archivo. Esta operación es necesaria para procesar toda la información en un archivo a la vez.

2. **Obtener_uno** (*Retrieve_One*): Obtener un solo registro de un archivo. Los procesos interactivos requieren esta operación.

3. **Obtener_el_siguiente** (*Retreive_Next*): Operación que permita obtener el "siguiente" registro, en algún orden "lógico". Las búsquedas y procesos de comparación requieren esta operación.

4. **Obtener_el_anterior** (*Retreive_Previous*): Misma descripción de la anterior, pero obtiene el registro anterior en lugar del siguiente.

5. **Insertar_uno** (*Insert_One*): Agrega, ya sea al final o indistintamente en cualquier lugar del archivo, un registro al archivo.

6. **Borrar_uno** (*Delete_One*): Borra ya sea el último o indistintamente cualquier registro de un archivo.

7. **Actualiza_uno** (*Update_One*): Obtiene un registro, modifica la información de uno o más campos, y regresa el registro a su lugar en el archivo.

8. **Obtener_algunos** (*Retrieve_Few*): Obtiene un número cualquiera de registros. Se usa generalmente para obtener registros ya sea en un rango o que cumplan algún criterio.

Por lo tanto, un sistema de archivos es un conjunto de programas (*software*) del sistema que garantice se tengan las definiciones antes mencionadas y el conjunto de operaciones antes descrito. Típicamente la única manera en que un usuario o proceso accese información, es a través de un sistema de archivos.

Para un sistema operativo interactivo (UNIX), se requiere tener una organización que permita conocer los requerimientos del usuario como son:

1. Cada usuario debe ser capaz de crear, borrar y modificar sus propios archivos.
2. Cada usuario debe tener el control sobre el acceso a sus archivos.
3. Cada usuario debe poder mover información entre archivos.
4. Cada usuario debe poder respaldar y restituir su información.
5. Se debe poder accesar un archivo por un nombre simbólico.

El sistema de archivos de UNIX, en general, cuenta con todas las características antes descritas y las implementa por medio de una estructura de árbol invertido, esto es, que la raíz del árbol (**root**) está encima de las ramas y las hojas. Básicamente hay dos tipos de archivos distinguidos:

1. **Directorios o subdirectorios:** Corresponden a los nodos no terminales en esta estructura de árbol. Son archivos especiales que son usados para crear acceso a otros archivos del disco conteniendo información sobre éstos. Hay dos entradas especiales en estos directorios, el "." que es una entrada a sí mismo, y el ".." que es una entrada al directorio inmediato superior.
2. **Archivos de datos:** O simplemente **archivos**, contienen cualquier otro tipo de información y corresponden a las hojas de esta estructura.

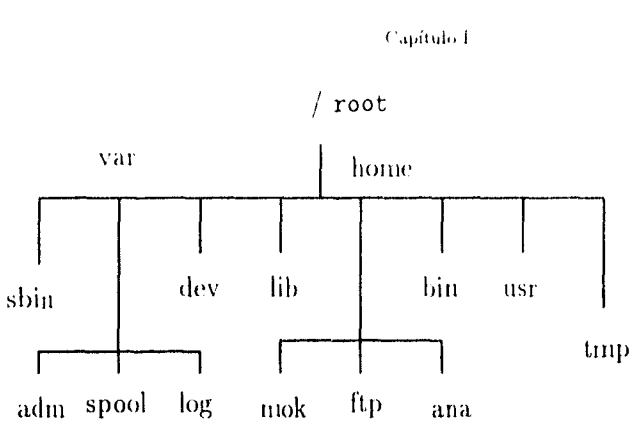


Figura 1.4: Representación gráfica de un sistema de archivos (árbol invertido)

Sin embargo, físicamente los discos están divididos en sectores de 512 bytes y el sistema de archivos debe de implementar mecanismos para que lógicamente sean vistos de otra manera. Primero se le debe dar un formato al disco para mantener homogeneidad en la estructura con la que se guarda la información. Este formato deja que lógicamente se vean bloques de diversos tamaños (1024 bytes, 2048 bytes, etc;) y tener varios pedazos o particiones en un mismo disco. Esto permite tener al disco con pedazos formateados a distintas densidades. Cada una de estas particiones[3] se monta sobre un directorio, creando así un único árbol.

```
# df
 Filesystem  1024-blocks  Used    Available  Capacity  Mounted on
 /dev/hda1   240023       224569  2645       99%       /usr
 /dev/hda2   125901       109737  9445       92%       /home
 /dev/sda1   640123       624669  2645       99%       /usr/local
 /dev/sdb5   441023       424579  2645       99%       /kernels
 /dev/sdb1   41023        42457   451        99%       /

#
```

Los bloques son la parte básica en un sistema de archivos, ya que los pedidos de lectura y escritura del subsistema de archivos del kernel, son siempre transformados y puestos en una cola con el número exacto del bloque a leer,

[3]Se puede hacer una partición de toda la capacidad del disco

el bloque del sistema de archivos, no el bloque físico del disco. Por ejemplo, para leer un carácter del disco, el subsistema de archivos tiene que leer todo un bloque y guardarlo en un *buffer*, aunque sólo sea entregado el cáracter pedido. Todas las funciones referentes al disco son hechas por medio de lectura y escritura sobre el bloque, de ahí que se les llamen dispositivos de bloque.

Cada una de estas particiones debe tener información propia, referente al espacio total en la partición, el espacio ocupado, el número de inodos[4] totales en la partición, el número de inodos usados, los archivos que pertenecen a la partición, etc. A esta información se le conoce como los *Metadatos*. Cada partición tiene un *Superbloque* y cada archivo de la partición tiene una entrada en la tabla de los inodos.

El Superbloque, es una tabla que toda partición debe tener. Este superbloque contiene la información de todo el sistema de archivos. El tamaño, el número de bloques libres, la lista de bloques disponibles, el índice del siguiente bloque libre en la lista, el tamaño de la lista de inodos, el número de inodos libres, la lista de inodos y el índice al siguiente inodo libre en la lista, entre otras cosas.

Algunos bloques del sistema de archivos son reservados para el uso exclusivo del administrador de la máquina o superusuario. Esta información es guardada en el superbloque y el número de bloques disponibles al usuario generalmente no son el total de bloques realmente disponibles. De tal modo que esto deja algunos bloques libres para uso del sistema y del administrador. Si no se reservara este espacio, el sistema podría volverse inarrancable al no tener espacio para guardar información en el momento de arrancar, y al tratar de obtener un bloque libre, el sistema se caería. Con estos bloques reservados, se asegura que hay espacio para arrancar y permitir al superusuario limpiar el disco.

Para el sistema UNIX, cada usuario debe tener un número de identificación única que le permita implementar los mecanismo de seguridad de la información. A este número se le conoce como el *User Identification* o simplemente *UID*. El UID del administrador o superusuario es el 0. De hecho,

[4]Más adelante se hará una explicación de los inodos, también llamados i-nodos, nodos-i o *Nodes-i*

el superusuario no es quien manualmente revisa y corrige algún disco al momento de arrancar, pero si lo hace un proceso cuyo dueño es el superusuario, es decir tiene el UID 0.

Metadatos

La tabla de los inodos guarda la información básica de un archivo: el UID del propietario del archivo (para saber quién es el dueño), la fecha y hora de modificación y último acceso, el tamaño del archivo, tipo de archivo, permisos, número de ligas al archivo y la dirección de los bloques en el disco, entre otras cosas.

Dentro de la tabla de inodos, los 9 primeros lugares direccionan al disco directamente, el décimo lugar contiene una tabla que apunta a otra tabla con 13 lugares (primer nivel de indirección), el décimo primer lugar apunta a una tabla que a su vez apunta a otra tabla, y ésta apunta a los bloques de disco (segundo nivel de indirección). El décimo segundo lugar contiene un apuntador a una tabla que contiene apuntadores a otra tabla de inodos que, finalmente, apuntan a un espacio físico del disco. Por último el décimo tercer lugar tiene tres niveles de indirección dándonos un total de 13^3 lugares.

Con esta organización el sistema de archivos queda completo. Sabemos claramente dónde buscar un archivo. Por ejemplo de la imagen del kernel del sistema operativo, típicamente **/vmunix**, obtenemos su inodo mediante el Superbloque de la raíz ("/") y con este inodo podemos obtener bloque a bloque la información del archivo.

1.1.4 Sistemas Operativos nativos de la SPARC

Originalmente SUN Microsystems desarrolló un sistema operativo UNIX para sus estaciones de trabajo. Este sistema operativo, llamado SunOS, incorporó todos los cambios y mejoras del UNIX desarrollado por la universidad de California. campus Berkeley[5], y esto lo hace muy estable y funcional, ya que el UNIX BSD al ser desarrollado en un ambienet académico, está planeado

[5]Todos los sistemas UNIX que incorporan estos cambios son conocidos como UNIX BSD.

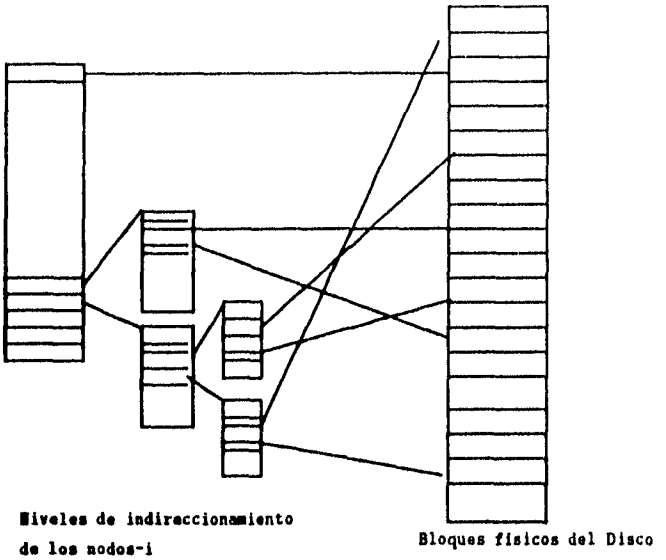


Figura 1.5: Tabla de los inodos

para trabajo más pesado, mayor carga en los servidores, cálculos numéricos fuertes y, en general, una respuesta más eficiente en todos aspectos (redes, procesamiento, acceso a disco).

Sin embargo, en un intento de Sun por competir (en cuanto a velocidad) con las estaciones de trabajo Alpha DEC, SUN creó una serie de computadoras con multiples procesadores para las cuales desarrollaron un nuevo sistema operativo (SOLARIS 2.x), que es capaz de paralelizar procesos, es decir un sistema operativo *multithread*[6].

Para poder construir tal sistema operativo se implementó el uso de semáforos[7]. El problema con estos semáforos es que cuando tenemos dos rutinas que hacen una misma llamada al sistema (system-call) surge un problema llamado *deadlock* (abrazo mortal), lo cual representa un problema mucho más

[6] ***Multithread*** significa que se tienen varios puntos de ejecución dentro del mismo espacio de direccionamiento

[7] Los semáforos siempre se han usado en el desarrollo de kernel's de sistemas operativos Unix

serio que una simple pérdida en la velocidad de respuesta del procesador; en realidad este problema generalmente causa la caída completa del sistema. No fue sino hasta mediados de 1996 cuando la última versión de SOLARIS (2.5) alcanzó la estabilidad y eficiencia deseadas.

El problema básico aquí es que SOLARIS incorporó todos los cambios y mejoras del sistema original desarrollado en los Bell-Labs (que se conoce como **System V**) y que es incompatible con el UNIX BSD. Así, la gente de SUN dejó de mejorar al antiguo sistema operativo (SunOS) limitándose a dar soporte y arreglar sus fallas. SunOS ya no tiene soporte para el nuevo hardware de SUN, tarjetas de video, multiproceso y otros periféricos. La última mejora que se le hizo a SunOS fue el poder usar dos procesadores, pero no en paralelo; esta mejora se limita a usar el otro procesador cuando el primero tiene una carga muy alta.

En resumen, tenemos dos sistemas operativos desarrollados por SUN Microsystems que no son compatibles entre sí:

1. **SunOS:** UNIX BSD, estable, ya no es soportado. No hay manejadores para este sistema con el nuevo hardware. No maneja varios procesadores.
2. **SOLARIS:** Sistema UNIX SysV[8]. Poco estable, multiproceso.

El UNIX BSD y el UNIX SysV se convirtieron rápidamente en los dos estándares más importantes en el desarrollo de cualquier sistema operativo UNIX. Poco después aparece otro estandard, POSIX, que en lugar de tratar de juntar lo mejor de los primeros dos estándares, redefine las especificaciones del sistema, además de incorporar otros cambios y mejoras al mismo. Por lo que se tienen tres estándares distintos, UNIX BSD, UNIX SysV y POSIX.

1.1.5 Arranque de un sistema operativo

Cuando encendemos una máquina, es muy común que ésta ya cuente con un sistema operativo el cual administra los recursos de la máquina, dándonos

[8]Abreviación de **System V**.

acceso a ellos (disco, teclado, consola). Sin embargo, cuando la máquina es encendida, la memoria, los registros y en general toda la máquina carece de algún tipo de información, el kernel del sistema operativo se encuentra en algún dispositivo (disco) pero no en memoria y por lo tanto no puede ser ejecutado. Sin embargo, de alguna manera el kernel del sistema operativo llega a la memoria, se ejecuta y la máquina eventualmente contesta con algún carácter diciendo que ya está activa y lista para ser utilizada (prompt).

A este proceso que se realiza, se le llama *bootstrapping*; el termino proviene de finales de la segunda guerra mundial. Lo primero que tenía que hacer un buen soldado inmediatamente después de haberse levantado, era ponerse y abrocharse las botas (*bootstrap*) como signo de que estaba listo para pelear. De una manera análoga, una máquina lo primero que debe de hacer es **levantarse** y cargar un sistema operativo.

Generalmente el levantantar una máquina (*bootstrapping*), consiste en que el ROM da la información suficiente al CPU para que baje a leer el(los) primer(os) sector(es) de algún dispositivo, como discos duros, CDROMs o discos flexibles. El ROM tiene por labor únicamente cargar esta información y ejecutarla, por lo que los bytes que están en dichos sectores no son otra cosa que un programa lo suficientemente inteligente como para que, sin haber cargado aún el kernel del sistema operativo, sepa de una manera rápida y primitiva, de qué tipo de sistema de archivos se trata, dónde está ubicado el kernel, cómo se llama, cargarlo a memoria y ejecutarlo.

A este programa que habita en estos sectores especiales, se le llama **cargador** o *bootloader*.

1.2 Arquitectura SPARC

1.2.1 Organización de la SPARC

El procesador SPARC, creado por SUN microsystems, es un procesador RISC de 64 bits, con 69 instrucciones básicas codificadas todas en 32 bits (4 bytes)[9]

[9] A diferencia de otros procesadores que codifican sus instrucciones en tamaños variables

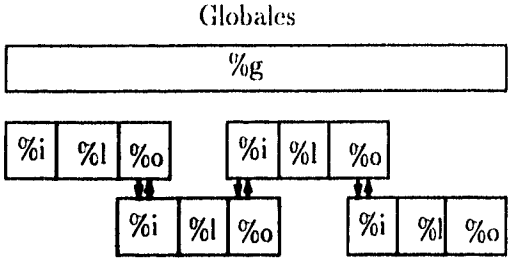


Figura 1.6: Registros deslizantes de la SPARC

con un espacio de direccionamiento de 2^{32} bytes.

El procesador consta de la Unidad Entera (UI), de la Unidad de Operaciones Flotantes (FPU) y opcionalmente de un Coprocesador. Cada unidad contiene sus propios registros y puede estar en dos modos, usuario y supervisor. Hay instrucciones que sólo pueden ejecutarse en modo supervisor; las aplicaciones se ejecutan en modo usuario.

Los procesadores RISC, al reducir su tamaño, básicamente tomaron dos tendencias, la MIPS y la SPARC, que se distinguen en la utilización de este espacio sobrante. El procesador SPARC utiliza el espacio sobrante para registros. La UI usa registros de propósito general que pueden ir de 40-520, y 8 registros *%g* globales.

El procesador SPARC usa un mecanismo de *registros deslizantes* (figura 1.6). En principio cada proceso tiene acceso a los 8 registros globales (*%g*), a 8 registros locales propios (*%l*), a 8 registros de entrada (*%i*) y a 8 registros de salida (*%o*). Los parámetros que recibe el proceso se colocan en los registros de entrada y los resultados se colocan en los registros de salida. La idea de los registros deslizantes consiste en reutilizar los registros de salida de un proceso como entrada al otro y viceversa.

El registro *%g0* es sólo de lectura y contiene al 0. Esto sirve para tener *a la mano* un 0. Generalmente se usa para comparaciones. movimientos, corrimientos (*Shift*) y demás procesos de aritmética, pero sobre todo el cero es muy usado para hacer máscaras lógicas (*OR, AND, XOR*).

El procesador cuenta con dos contadores de ejecución, el *PC (Program Counter)* que contiene un apuntador a la instrucción que se está ejecutando, y el *NPC New Program Counter* que contiene un apuntador a la siguiente instrucción a ejecutar. Esto sirve para que, mientras que se ejecuta la instrucción a la que apunta el PC, se va decodificando la instrucción que está apuntada por el NPC y este apuntador será obtenido por el PC. Esto sirve para ganar tiempo de ejecución, ya que la instrucción que está apuntada por el PC ya no tiene que ser decodificada.

Estos detalles técnicos del procesador son importantes y hay que tenerlos muy presentes en el momento de codificar en ensamblador, pues el uso de los registros deslizantes permite aprovechar mejor el tiempo de procesador y el espacio en los registros al no tener que copiar la información. Esto sirve, por ejemplo, cuando se hace una llamada a alguna subrutina y se necesita hacer algún paso de parámetros; en cualquier otro caso, éstos deben ser copiados, mientras que en el caso de SPARC no. Hay que tener mucho cuidado también con el NPC, ya que el orden de las instrucciones no es el tradicional. Si típicamente se desea hacer una suma (*SUB*), una máscara (*OR*) y luego una llamada a `_bootmain` (*CALL*), tradicionalmente el código se ve así:

```
SUB   %l4,%l1,%l2
OR    %l2,%l4,%l2
CALL  _bootmain
SUB   %l4,%l1,%l2
```

Sin embargo en la SPARC, al ejecutarse ela instrucción `CALL`, se está decodificando la segunda resta, y antes de entrar a la instrucción `CALL`, la resta será ejecutada dando como terrible resultado que el contenido de los registros no sea el deseado. Por lo tanto, el mismo código en ensamblador de la SPARC debe verse así:

```
SUB   %l4,%l1,%l2
CALL  _bootmain
OR    %l2,%l4,%l2
SUB   %l4,%l1,%l2
```

1.2.2 El Lenguaje Ensamblador

El lenguaje ensamblador de la SPARC, consta de un grupo básico de 5 tipos de instrucciones:

1. **Cargar/Guardar** - Instrucciones para accesar, guardar e interactuar con los registros y la memoria (`set, sethi`).
2. **Aritméticas/Lógicas/Corrimiento** - Operaciones básicas entre los elementos de los registros (`add, sub, or, and, subcc`).
3. **Control/Transferencia** - Operación para transferir el control de la ejecución del código. Modifica el PC (`jmp, call`).
4. **Estados/Registros/Acceso** - Para guardar y restaurar el estado de los registros, Principlamente su uso es al regresar de una llamada a subrutina y restaurar el estado original del procesador (`bne`).
5. **Punto Flotante/Coprocesador** - Conjunto de operaciones para accesar el coprocesador matemático. Operaciones complicadas y manejo de números reales.

Existe un inconveniente, dado que en la SPARC el espacio de direccionamiento es de 32 bits (2^{32}), y todas las instrucciones son de 4 bytes, no es posible cargar una dirección de memoria con una simple instrucción. Por lo menos un byte debe ser usado para la operacion (**set**) lo que nos deja 3 bytes para la dirección de memoria, esto es un espacio de direccionamiento de 2^{24} bits. La solución se hace por medio de dos llamadas, donde una carga la parte significativa de la dirección y la otra la menos significativa.

```
sethi  %hi(_bss),%l4       Cargamos la parte alta
or     %l4,%lo(_bss),%l4   Cargamos la parte baja
```

El cargador SILO sólo utiliza el lenguaje ensamblador de la SPARC para arrancarse a sí mismo y accesar la dirección del ROM de la SPARC. El resto del código está hecho en lenguaje C, por lo que SILO sólo utiliza las operaciones más sencillas y básicas de este lenguaje.

Capítulo 2

El sistema operativo Linux

2.1 Linux

Linux es un sistema operativo *multithread* que ha sido desarrollado desde principios de los 90's (Abril 1991) hasta la fecha, siendo ya un sistema estable, funcional, eficiente y poderoso en los procesadores INTEL 386 y posteriores, se ha portado a varias arquitecturas como son los chips Motorola, Amiga y Alpha MIPS. Ahora se ha planteado la posibilidad de portarlo a las plataformas SPARC (SUN) y Silicon Graphics (SGI).

Originalmente Linux fue escrito por un joven estudiante de la Universidad de Finlandia llamado Linus Benedic Torvalds. Sin embargo, han sido tantas las personas que han contribuido a lo largo del planeta, que ya no se puede decir que sea el esfuerzo solitario de una sola persona. Linux es un clon, escrito totalmente desde cero, del sistema operativo UNIX.

Linux comenzó como un proyecto de Linus para explorar el chip de INTEL 386 pero después el proyecto creció demasiado. Linus trataba de explotar hasta el máximo las ventajas de este chip, hasta que los pequeños programas de prueba que hacía se volvieron la primeras piezas funcionales de un sistema operativo UNIX, las bases de un kernel.

Al principio Linus no pensó en hacer un sistema operativo grande y de uso general y sólo era un pasatiempo [Torv]. Sin embargo, el uso de la li-

cencia GPL[1] permitió que Linux creciera y se desarrollara exponencialmente. Esto ha permitido que se tengan los manejadores para casi cualquier dispositivo, ya sean externos, internos, de video, disco o tarjetas de red. Por otro lado la mayoría del software hecho para UNIX funciona bajo Linux, y algunas compañías están desarrollando software exclusivo para este sistema operativo.

Linux fue diseñado tratando de evitar el incurrir en los errores que se cometieron en minix [Torv], y que la implementación estuviese basada solo en un estándard; se tomaron primero las definiciones de POSIX, luego de UNIX SysV y luego las de UNIX BSD, por lo que Linux es una mezcla que trata de tomar lo mejor de los tres estándares.

La mayoría de las utilerías de las distribuciones de Linux son parte del proyecto GNU[2]

2.2 El sistema de archivos ext2

2.2.1 Descripción

El primer UNIX que se implementó para la plataforma PC[3] fue **minix**; este sistema operativo tenía su propio sistema de archivos llamado *minix filesystem* (**minixfs**). Las especificaciones de este sistema de archivos se convirtieron en un estándard para cualquier UNIX en la plataforma PC, y fue el primer sistema de archivos que implementó Linux.

Sin embargo, este sistema de archivos tenía muchas desventajas como por ejemplo, el tener un límite de 64Mb en el tamaño de las particiones, 14 carácteres de límite en el nombre de los archivos y no ser extensible. Ante esta situación, Rémy Card diseñó un nuevo sistema de archivos llamado **extfs**, que estaba basado principalmente en la implementación del minixfs, pero lo había extendido (de ahí el nombre de *Extended File System, extfs*), quitando la limitante en las particiones y extendiendo el límite en el tamaño máximo para los nombres de archivos a 255 carácteres.

[1] Revisar el apéndice B

[2] Revisar el apéndice B.

[3] Procesadores INTEL 8088, 8086, 80286 y posteriores.

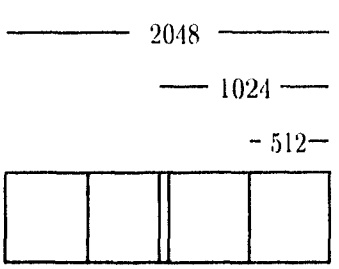
Aun así, el sistema tenía muchas deficiencias, heredadas de la implementación de minixfs; a decir verdad, minixfs fue implementado originalmente en máquinas con tecnología muy vieja (procesadores 8088), además de que Linux sólo funciona en máquinas con procesadores 386 y posteriores, así que la diferencia de tecnología de cuando fue implementado minixfs a ese momento era muy marcada, por lo que Rémy Card decidió escribir un nuevo sistema de archivos totalmente nuevo que aprovechase las ventajas de las nuevas tecnologías. A este segundo sistema de archivos que escribió, le llamo *Second Extended File System* (**ext2fs**). Este sistema de archivos no sólo incluía las ventajas del extfs, sino que además provee un mejor manejo del espacio en el disco[4], el uso de banderas especiales para el manejo de archivos, el uso de listas de control, es extensible e incluye las mejoras hechas al sistema de archivos hecho por la universidad de Berkeley. Este nuevo sistema de archivos (ext2fs) se convirtió en el estandard de Linux, no sólo por estar muy bien diseñado, sino porque fue hecho exprofeso para este sistema operativo; digamos que el ext2fs es nativo de Linux.

2.2.2 Diferencias y mejoras

Dentro de las mejoras que tiene el ext2fs están esencialmente la optimización en los recursos del disco, del tiempo y del espacio. Por ejemplo permite la fragmentación en los bloques. Generalmente un archivo tiene un cierto número de bytes, pero el sistema de archivos lo guarda en cierto número de bloques de tamaño fijo[5], así que el archivo no llena exactamente todo el espacio que ocupan los bloques. Por ejemplo, un archivo que contenga sólo 1 byte en un sistema de archivos de 1024 bytes por bloque estará ocupando los 1024 bytes y se están desperdiciando 1023. Se han hecho estudios y en promedio se desperdicia medio bloque por archivo. En la mayoría de los casos esto es mucho espacio que no se usa del sistema de archivos.

Para contrarrestar este inconveniente, el ext2fs no usa el bloque completo, sino fragmentos, donde el tamaño del fragmento debe ser el tamaño del bloque físico multiplicado por una potencia de 2. Así, un archivo se

[4] Literal de la documentación: *provides a better space allocation management*[ExtD].
[5] Toda la partición está formateada con la misma densidad, X bytes por bloque.



Físico $\leq$ Fragmento $\leq$ Lógico

Figura 2.1: Relacíon entre los tamaños.

convierte en una sucesión de bloques llenos y una pequeña sucesión de fragmentos consecutivos. Cuando se juntan suficientes de estos fragmentos se reagrupan y forman un nuevo bloque. La figura 2.1 describe la relación entre los tamaños de los bloques físicos y lógicos, y el tamaño del fragmento [ExtD].

Además, el *ext2fs* es considerablemente mas rápido que el resto de los sistemas de archivos. Cuando se hace una modificación a un archivo, se genera una llamada al sistema (*system-call*) que le avisa al subsistema de archivos que hay que modificar algo "físicamente" en el disco; el subsistema marca los bloques "sucios" (bloques que deben ser modificados) y le contesta al kernel que ya fue realizada la tarea, quien avisa a la aplicación o usuario que la información ya fue escrita, aunque la información aún no haya sido escrita. Antes de escribir la información en los bloques, los sistemas de archivos convencionales escriben la información de los metadatos y cuando reciben la confirmación de que éstos fueron modificados satisfactoriamente, entonces realiza la escritura física de la información en el disco.

El *ext2fs* realiza la escritura de los metadatos al mismo tiempo que escribe los datos, ahorrando hasta un 33% en tiempo de escritura. La idea fundamental de escribir los metadatos antes de los datos, es por si surge alguna eventualidad, como algún corte en el suministro de energía, con esto se garantiza que los metadatos (la parte medular de la información) no entra en conflicto con el resto del sistema. Sin embargo para fines prácticos, estas "eventualidades" generalmente causan la pérdida de la información y la ganancia es mínima. En Linux, se preocuparon más por hacer un verificador

del sistema de archivos (*filesystem check*) más inteligente que pudiese recuperar datos, ya que aunque se tengan los metadatos de archivo, si los datos no fueron escritos, da lo mismo no tenerlos.

El sistema de archivos ext2 incorpora nuevos campos en el superbloque de tal forma que es posible recuperar[6] archivos borrados en UNIX. Generalmente en UNIX una vez que un archivo era borrado, se perdía para siempre. Los campos que hay en el superbloque del ext2 son:

1. **s_inodes_count** : Número total de inodos.
2. **s_blocks_count** : Número total de bloques.
3. **s_r_blocks_count** : Número total de bloques reservados para uso exclusivo del superusuario.
4. **s_free_blocks_count** : Número total de bloques libres.
5. **s_free_inodes_count** : Número total de inodos libres.
6. **s_first_data_block** : La posición del primer bloque de datos en el sistema de archivos. Generalmente este número es el 1 para sistemas de archivos de 1024 bytes por bloque, y 0 en otro caso.
7. **s_log_block_size:** Usado para hacer cálculos con el tamaño de los bloques. Este valor es en realidad `1024 << s_log_block_size`.
8. **s_log_frag_size:** Usado para hacer cálculos con el tamaño del fragmento.
9. **s_mtime:** Hora en la que el sistema de archivos fue montado.
10. **s_wtime:** Hora en la que fue hecha la última escritura en el superbloque.
11. **s_mnt_count:** El tiempo que ha durado montado el sistema de archivos en modo lectura/escritura sin haberse revisado.

[6] Algunos programas ya hacen uso de esta ventaja como el Midnight Comander (clon del Norton Comander) version 3.2

12. **s_max_mnt_count:** El máximo número de veces que un sistema puede ser montado como lectura/escritura, sin ser revisado.

13. **s_state:** El estado actual del sistema de archivos. EXT2_VALID_FS (0x0001) que significa que el sistema de archivos fue desmontado limpiamente y EXT2_ERROR_FS (0x0002) que significa que hubo un error al desmontar el sistema de archivos o no fue desmontado.

14. **s_errors:** Manejo de errores.

Una vez que el superbloque es leído en memoria, el código para el manejo del sistema de archivos ext2fs calcula otro tipo de información y la guarda en otra estructura[7]; aquí es donde maneja e incorpora las mejoras al sistema de archivos, calcula el tamaño del fragmento por bloque, el número de fragmentos en un bloque, las opciones que se usaron al momento de montar el sistema de archivos, y un apuntador a un espacio reservado de memoria que contienen toda la información del superbloque.

2.3 Proyecto Linux/SPARC

2.3.1 Descripción del proyecto

Linux/SPARC es un proyecto a nivel mundial que está portando o migrando el sistema operativo Linux a la plataforma SPARC (a las estaciones de trabajo creadas por la compañía SUN Microsystems). Esto no es una nueva versión de Linux, el portar significa que el mismo código que está integrado en la distribución del kernel, compile y funcione en esta nueva arquitectura. Es un hecho que hay que hacer cambios para tal efecto, pero todas las distribuciones de Linux deben traer los cambios. El kernel y la interface a nivel de usuario en esencia son los mismos, sólo cambian las partes dependientes del hardware de la máquina.

Este trabajo comenzó a finales de 1994 y aunque hasta la fecha (1996) sigue estando bajo desarrollo[8], las pruebas preliminares de esta migración

[7]Estas estructuras se encuentran completas en el código fuente del kernel (`/usr/src/linux/include`).

[8]Cuando un trabajo se sigue desarrollando se le llama WIP (*Work in progress*).

han indicado que es más estable que SunOS.

Las razones principales para portar Linux a la SPARC, además de que es un sistema operativo gratuito (evitar pagar costosas licencias), son:

1. Acceso y control total del código fuente del kernel.
2. Uso de consolas virtuales.
3. Es más rápido que SunOS y SOLARIS[9].
4. Funcionan los binarios de SunOS (y próximamente los de SOLARIS).
5. Mejoramiento en los algoritmos.
6. Mejoramiento en el uso de los chips.

Actualmente Linux ya ha sido probado en las arquitecturas Sun4c (SPARC1, 1+, 2, IPC, IPX, SLC, ELC) y SUN4m (SPARC Classic, LX, 5, 10, 20) y se está trabajando también en el nuevo procesador del SPARC: Ultra.

El responsable a nivel mundial del proyecto es David S. Miller, quien escribió la mayor parte del código para la SPARC. Durante el verano de 1996, David fue llamado por la compañía Silicon Graphics para empezar el proyecto Linux/SGI, dejando como líder del proyecto Linux/SPARC a Eiddie C. Dost. Además de estas dos personas, el equipo lo constituyen Peter A. Zaitzev responsable de los bloques de arranque y discos de instalación, y Miguel de Icaza responsable del sistema de ventanas X, la librería standard de C, ligado dinámico de los binarios, binarios ELF[10], manejo de las tarjetas de video y compatibilidad con SunOS.

Para Junio de 1996 se tienen funcionanado las sisguientes piezas, entre las más importantes:

1. Linux version 2.0 (liberado como Linux/Sparc).

[9] Ver apéndice C.
[10] Ver sección 4.1.5

2. Cargador dinámico.
3. Bibliotecas compartidas ELF.
4. Tarjetas de video: *cgfourteen* (SX), CG3, TGX.
5. Hypersparc (modo SMP).
6. Dispositivos SCSI.
7. Sistema de ventanas X11.
8. Manejador para el disco flexible.
9. Biblioteca de C.
10. Ejecutables de SunOS.
11. NFS root[11].

Aunque funcionan los binarios de SunOS, gran parte de los programas para SPARC ya son compilados nativamente en Linux/SPARC y compañías como RedHat ya están haciendo distribuciones de Linux/SPARC que próximamente estará en CDROM listo para instalarse.

2.3.2 SILO: El Cargador

Además del equipo de personas mencionado en la seccion anterior, hay mucha gente que ha contibuido con pequeños programas y proyectos en el porte de Linux/SPARC. Uno de estos proyectos es precisamente hacer un cargador para Linux en la plataforma SPARC. El cargador no es una tarea urgente, pues cuando se hacen este tipo de desarrollos, es conveniente tener dos terminales: una donde se programa y se configura, y la otra, que arranca por red, obtiene la información relevante de la primer terminal y es en la que se hacen todas la pruebas. Por ello no es necesario cargar el sistema operativo de los discos duros locales.

Sin embargo, aunque no es una tarea urgente sí es muy importante para el desarrollo del sistema, ya que una vez que el kernel esté funcionando, la

[11]La raíz del sistema de archivos está en otro servidor vía NFS (*Network File System*).

máquina deberá arranacarlo del disco duro local y el resto del desarrollo del sistema se hace sobre el sistema operativo en pleno funcionamiento, además de que el sistema, una vez terminado, debe tener todas las piezas que lo forman incluyendo a su cargador.

Se me invitó a participar en este proyecto y al principio se empezó a trabajar con una máquina del Instituto de Ciencias Nucleares y un disco duro SCSI de 202Mb del Departamento de Matemáticas de la Facultad de Ciencias, ambas dependencias de la Universidad Nacional Autónoma de México. Peter Zaitcev ya tenía funcionando un minicargador para discos flexibles y con esa base se empezó el trabajo. El siguiente capítulo explicará en detalle cuál fue el proceso a seguir en el desarrollo del cargador y como fue implementado.

Capítulo 3

Cargadores para Linux

Para entender el funcionamiento del cargador en cuestión (SILO), primero se hará una breve descripción de los cargadores ya existentes para Linux en otras arquitecturas, los elementos a favor y en contra, y las necesidades que cada uno de estos cargadores tuvo que satisfacer dependiendo de la arquitectura. Los cargadores existentes son:

1. LILO: Cargador para Linux/i386.
2. MILO: Cargador para Linux/ALPHA.

SILO por su parte es el cargador para Linux/SPARC y su implementación no se parece a la de los cargadores antes mencionados, esto no significa que no tome ciertos elementos, sino que simplemente SILO es un cargador nuevo y escrito desde cero. Se explicará por qué SILO fue hecho de esta manera, por qué no se decidió portar alguno de los cargadores ya existentes a la plataforma SPARC, y qué elementos fueron tomadas y adaptados a SILO.

3.1 LILO: Cargador para Linux/i386

3.1.1 Descripción

Lilo es un cargador para Linux en la plataforma PC (i386) desarrollado y mantenido por Werner Almesberger. Lilo tiene la habilidad de arrancar la

máquina con otros sistemas operativos además de Linux tales como OS/2, DOS y Windows95. La idea de permitir arrancar otro sistema operativo es fundamental para atraer a la gente hacia Linux; la mayoría de los sistemas operativos para PC antes mencionados, cuentan con una popularidad muy grande, de tal forma que la gente es muy reticente a cambiar lo que ya conoce por experimentar con un nuevo sistema operativo. Para contrarrestar esta reticencia, LILO aprovecha que Linux permita que en un mismo disco duro se tengan varias particiones con distintos sistemas operativos y así, LILO cargará cualquiera de los sistemas instalados en el disco, de tal forma que se puede arrancar la máquina con cualquiera de los sistemas antes mencionados.

La manera como funciona Lilo es por medio de tablas, esto es: cuando se quiere arrancar la máquina con algún archivo, primero se obtienen todos los sectores físicos del disco que lo forman y se guarda el número de cada sector en algún espacio reservado en el disco. Cuando la máquina arranca, basta leer esta tabla o mapa de bloques para cargar cada uno de ellos a memoria y ejecutarlo. Típicamente este archivo es la imagen del kernel del sistema operativo. La tabla tiene dos entidades básicas (entre muchas otras), como son el nombre del archivo en el que se encuentra el kernel, y un nombre de configuración que se asociará a dicho kernel. Esta medida es tomada para poder arrancar con varios kernels u otro sistema operativo.

Lilo tiene dos partes fundamentales, la primera es la parte que se instala en los sectores de arranque de la PC (MBR) y a la que llamaremos simplemente **LILO**[1]. Este programa no entiende lo que es un sistema de archivos, ni siquiera el sencillo sistema de archivos de DOS. Simplemente recibe el nombre de una configuración de un kernel con el cual determina qué bloques debe cargar a la memoria para después ejecutarlos.

La otra parte de Lilo es un programa a nivel de usuario (*user-level*), que debe ser corrido después de editar un archivo llamado **/etc/lilo.conf**. Este programa, al que llamaremos **lilo**[2], lee dicho archivo de configuración y genera las tablas respectivas que LILO pueda entender. En el archivo de configuración se determinan varios aspectos importantes para el arranque de

[1]Se usan mayúsculas porque éste es el programa que es cargado al arrancar la máquina, y responde con estas siglas (en mayúsculas).

[2]Se usan minúsculas ya que es un programa que se llama así, `lilo` y típicamente está en el subdirectorio `/sbin` del sistema.

un kernel: el primero es un nombre de configuración para un kernel, que es el nombre con el que LILO sabrá cuál kernel cargar; el kernel mismo (/vmunix), que es el archivo que usará lilo para obtener la tabla de bloques del kernel, el dispositivo que se montará como directorio raíz, y las opciones de arranque. Por supuesto que el trabajo de entender sistemas de archivos, abrir inodos, obtener el número del bloque físico en disco y demás llamadas al sistema, las realiza el kernel y no lilo; éste simplemente le pide al kernel la información y la guarda en una tabla cuya localización es conocida por LILO; así, este último sólo se limita a reconocer el nombre de configuración de kernel y llamar a los bloques especificados en dicha configuración.

En este ejemplo, lilo genera la tabla de bloques a cargar basándose en el archivo "/vmlinux" y escribe el nombre "linux" en la tabla de configuraciones que LILO lee:

Ejemplo de : /etc/lilo.conf

```
#
#
# Start LILO global section
boot = /dev/hda
#
delay = 50        # Tiempo que tarda en arrancar
vga = normal      # Modo de Video
#
# Final de las secciones globales de LILO
# Empieza una partición de LINUX de arranque
image = /vmlinuz        # Archivo del kernel
root = /dev/hda1        # Dispositivo
label = linux           # Nombre de configuración para LILO
read-only               #
```

En el archivo `/etc/lilo.conf` puede haber tantas configuraciones como se quiera. Cuando la máquina arranca y LILO es cargado, éste responde con la palabra "`LILO`" y espera tanto tiempo en milisegundos como se haya especificado en la opción de retardo (`delay`) del archivo de configuración. Esta espera sirve para poder elegir el kernel a arrancar desde un sencillo menú que se invoca con la tecla de mayúsculas (*Shift*). Al ser presionada esta tecla, LILO responde con un *prompt* (`boot:`) y espera el nombre de alguna configuración: si se desconoce el nombre de las configuraciones, basta presionar la tecla de tabulador y aparecerá una lista con los nombres posibles para arrancar la máquina. Si no se presiona ninguna de estas teclas y el tiempo especificado en la opción de retardo expira, LILO carga automáticamente la

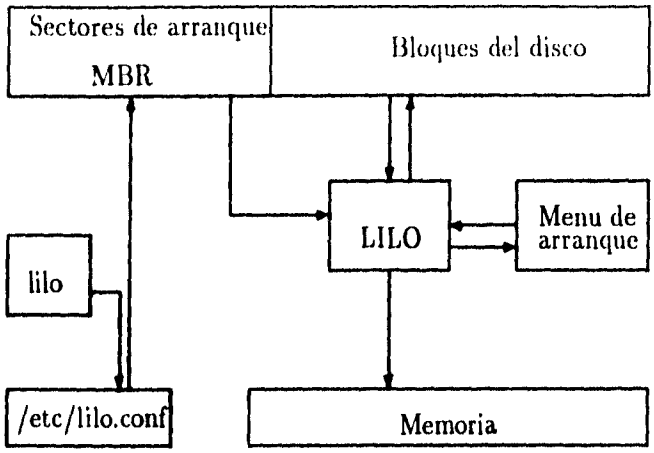


Figura 3.1: Funcionamiento esquemático de Lilo

primera configuración en la lista. Esto puede servir para que las máquinas arranquen por omisión algún sistema operativo específico.

Ejemplo de una secuencia de encendido

```
LILO
boot:
linux   dos     linux-nuevo     linux-sin-red
boot:
```

Una vez que LILO determina que se trata del nombre de una configuración válida, procede a leer cada entrada de la tabla y a cargar ese sector del disco a memoria para finalmente ejecutarlo. Así es como Lilo funciona, a base de tablas.

3.1.2 Motivaciones

El problema fundamental al que se enfrentó Werner Almesberger fue que el procesador INTEL arranca en un modo de direccionamiento de 16 bits, y aunque puede cambiar a modo de 32 bits, ya no puede regresar a 16 bits sin reiniciar el procesador[3], y Linux, y en particular la biblioteca ext2, tienen

[3]Existen varios tipos de rearranque en la PC, el primero es el "cold boot", que rearranca al procesador, limpia memoria y exige se haga una revisión de la memoria (RAM test). El

un modo de direccionamiento de 32 bits, por lo que resulta materialmente imposible el uso de esta biblioteca.

En este punto el procesador se encuentra entre el modo de direccionamiento de 16 bits del BIOS y el de 32 bits de Linux; el problema clave está en que la lectura del disco duro se hace por medio del BIOS en 16 bits, después se tendría que cambiar al procesador en modo de 32 bits para entender la biblioteca ext2fs; hasta aquí todo funciona bien, pero al tratar de hacer algún acceso al disco, leer las especificaciones de éste o cualquier otra función que utilizara el BIOS, ya no se puede regresar a 16 bits sin reiniciar al procesador. Esto es, una vez que se ha cambiado a modo de 32 bits ya no hay acceso al BIOS. De hecho, una vez que Linux ha arrancado, ya no es posible consultar información del BIOS de la PC. Esta es la razón principal para que LILO fuese hecho a base de tablas estáticas. La lectura del kernel se hace por medio del BIOS y la localización de los sectores se conoce a priori.

Otro problema que enfrentó Werner Almesberger es que en la PC existe el eterno problema de la memoria base (640Kb) y la memoria expandida, por lo que el kernel debe ir comprimido y no ocupar más de 500Kb. Una vez cargado en la memoria base, el kernel debe ser descomprimido y ya utilizar toda la memoria en bloque[4] único (como debe ser, como lo hace Linux). Por lo tanto, reconocer que se trata de un ejecutable ligado estáticamente en un archivo comprimido es bastante problemático. De este trabajo se encarga `/sbin/lilo` y el kernel se descomprime solo .

3.1.3 Elementos a favor y en contra

En realidad `lilo` no es el encargado de entender sistemas de archivos sino que simplemente hace una llamada al sistema operativo (kernel) y éste es el encargado de entender el sistema de archivos correspondiente y entregarle a `lilo` el número del bloque que forma al archivo. Esto es un punto a favor,

segundo es el "warm boot", donde el procesador y la memoria son limpiados pero sin una revisión de la memoria RAM; por ejemplo, el presionar las teclas Ctr-Alt-Del genera un "warm boot". Y en el tercero es donde el procesador rearranca, aunque sea preservada la información que hay en RAM.

[4]En un bloque único quiere decir sin distinción de memoria base, memoria expandida o memoria extendida. Simplemente un bloque de memoria.

ya que al añadirse el soporte para un nuevo sistema de archivos (ufs, WindowsNT), automáticamente Lilo está en posibilidades de poder cargar este nuevo sistema operativo.

Esta manera de resolver el problema de cargar un kernel a la memoria, por medio del uso de una tabla estática que contiene varios kernels, es muy sencilla y fácil de entender y modificar. Sin embargo, tiene algunos inconvenientes; el mas común es que si por alguna razón no se ejecutó el programa `lilo` después de instalar un kernel, éste simplemente no podrá ser cargado debiéndose a dos posibles razones:

1. No existe un nombre de configuración asociado a este kernel y por lo tanto no están mapeados los bloques que lo forman.
2. Si existiese un nombre de configuración para este kernel, los bloques mapeados son los del kernel anterior, posiblemente borrado y LILO cargará basura en lugar del kernel requerido.

Lo peor que puede pasar es que éste sea el único kernel instalado y la máquina ya no pueda arrancar. Para esto basta arrancar de disco flexible y correr el programa `lilo`.

Otro punto por el que no se optó por portar Lilo a la SPARC, es que la PC no tiene un ROM[5] inteligente como la SPARC, y es muy común que la gente arranque su sistema (SOLARIS, SunOS) tecleando el nombre del kernel y no teclee el de una configuración. Se desea mantener cierta compatibilidad con la manera tradicional de arrancar a las máquinas SPARC.

3.2 MILO: Cargador para Linux/ALPHA

3.2.1 Motivaciones

El responsable de portar Linux a la arquitectura Alpha fue Jim Paradis quien invitó a colaborar en este proyecto a David A. Rusling, autor y diseñador de

[5]El ROM de la SPARC se llama PROM y responde con el prompt **ok**, desde donde se le pueden dar instrucciones como **boot** y el nombre de un kernel.

MILO. El problema consistía en diseñar un pequeño cargador para esta plataforma lo suficientemente inteligente para entender por lo menos el sistema de archivos nativo de Linux (ext2), de tal forma que no se tuvieran que usar las tablas estáticas con mapa de bloques y se pudiera especificar directamente, en el momento de arrancar, el nombre de la imagen del kernel.

Esta idea surgió pues la arquitectura Alpha no tiene las limitantes del procesador INTEL y del BIOS de la PC, además de que la arquitectura Alpha requiere de cosas muy distintas a las que requiere la arquitectura PC, razón principal por la cual decidieron hacer un nuevo cargador y no portar Lilo a la Alpha.

3.2.2 Descripción

MILO es un cargador que entiende el sistema de archivos ext2, y por medio de una interface para el usuario se le puede decir el nombre del archivo que contiene la imagen del kernel a cargar. Así como en la PC la información vital para inicializar la máquina se encuentra en el BIOS, en la Digital el equivalente es la *consola* mejor conocida como **SRM console** (se describe ampliamente en el Manual de Referencia del Sistema). El cargador de cualquier sistema operativo en la Alpha necesita de este código, ya que al ser cargado, desde algún dispositivo o medio, el SRM console es quien provee la información necesaria y suficiente para accesar y manejar dichos dispositivos.

Para el manejo de hardware mas específico en la plataforma Alpha, se requiere del código especial hecho por Digital, *Digital UNIX PALcode* conocido simplemente como PALcode. Por todo esto, MILO consta fundamentalmente de las siguientes piezas funcionales [LnxJ]:

1. Manejadores de dispositivos (*Device Drivers*)
2. Kernel y pseudo-kernel de Linux
3. Código para la interface con el usuario
4. PALcode
5. Soporte para el sistema de archivos ext2

3.2.3 Elementos a favor y en contra

La gran cualidad que tiene MILO es la capacidad de buscar un archivo en el sistema de archivos ext2, que consiste (en esencia) de la siguiente sucesión de pasos:

1. Poder representar el nombre del archivo y obtener toda su información de la tabla de inodos.
2. Poder representar los dispositivos de la máquina (/dev) y determinar en qué dispositivo y partición se encuentra el archivo.
3. Recorrer la tabla de inodos para obtener cada bloque que forma el archivo.
4. Finalmente, cargar cada uno de estos bloques a memoria y ejecutarlo.

Uno de los principales problemas que tiene MILO es que resulta demasiado grande, ya que en realidad MILO es un kernel pequeño y recortado (pequeño para ser kernel, grande para ser un cargador). La idea general de MILO es que no se necesiten las tablas de LILO, y que MILO sea un cargador lo suficientemente inteligente como para entender el sistema de archivos nativo de Linux ext2. Sin embargo, el código que entiende este sistema de archivos fue extraído del kernel, al igual que las rutinas de SRM, para que MILO pueda entender y manejar los dispositivos.

Añadir todo este soporte hizo a MILO más grande, pero David Rusling pensó que de esta manera MILO sería más universal, ya que al soportar los mismos dispositivos que el kernel, y al ser éstos desarrollados tan rápido, MILO nunca perdería su vigencia.

Al principio, se vio que no era necesario el uso de verdaderas interrupciones para el manejo de los dispositivos, pues se había corregido el código que se había copiado del kernel y estas interrupciones eran simuladas (emuladas). Sin embargo el controlador para las tarjetas SCSI NCR 53C810 necesita hacer uso de verdaderas interrupciones, por lo que el equipo de MILO tuvo necesidad de agregar el código que habían quitado del kernel, y el código necesario para soportar verdaderas interrupciones, haciendo a MILO crecer

un poco más.

Cuando MILO fue desarrollado, era cargado en una sola etapa en una Alpha EB66 (basado en un procesador 21066), que era un modelo en evaluación. Sin embargo, al ser probado en una Alpha AxpPC133, se vio que MILO tenía que ser cargado de alguna otra manera, ya que se tuvieron problemas de espacio[6]. Como solución a este nuevo problema, se tuvo la necesidad de hacer MILO en dos etapas y por supuesto esto alteró la implementación de MILO e hizo que el código creciera aún más.

En resumen, MILO es un cargador demasiado grande y específico con las siguientes piezas funcionales:

1. Manejadores de dispositivos (*Device Drivers*)
2. Kernel y pseudo-kernel de Linux
3. Código para la interface con el usuario
4. PALcode
5. Soporte para el sistema de archivos ext2, extraídos del kernel
6. El soporte para el sistema de archivos ext2, que requiere entender el manejo de dispositivos ("/dev").
7. Código para el manejo de interrupciones, extraído del kernel.
8. Código para arrancar en dos etapas.

Y MILO, en lugar de ser un cargador pequeño en implementación y tamaño, resultó ser una especie de kernel "recortado" y adecuado solamente para arrancar y llamar a un verdadero kernel.

[6]Problemas de espacio en los sectores de arranque pues MILO ya no cabía en ellos.

3.3 SILO: Cargador para Linux/SPARC

3.3.1 Motivaciones

En un principio, se evaluó la posibilidad de portar los cargadores ya existentes para Linux a la arquitectura SPARC. Primero se estudió el código fuente de LILO, pero dado que no se tenían las mismas limitantes que en la arquitectura PC, ya que la SPARC cuanta con un ROM muy inteligente y completo (PROM), y agregando que la manera tradicional de arrancar una SUN es escribiendo el nombre del archivo (kernel) a arrancar, y no el nombre de una configuración, esta posibilidad fue desechada rápidamente. Por otro lado la presencia de MILO indicaba que había que hacer un cargador que soportase al sistema de archivos ext2 y no un cargador que leyese tablas.

Sin embargo, MILO resulta demasiado grande y específico. Se tenía que recortar todo el código nativo de la Alpha (SRM, PALcode, etc;) y "parchar" con el código respectivo para SUN. Además de que eso no resolvía el problema de tener un pseudo-kernel recortado y muy grande, que tenía que contemplar el soporte para distintos dispositivos, como sucedió con MILO.

La existencia de la biblioteca **ext2fs** abrió una puerta dándonos la posibilidad de hacer el cargador de una manera más limpia. Ligar contra esta biblioteca es no sólo más rápido y eficiente sino que más compatible. Cada vez que se haga alguna mejora a la biblioteca, sólo basta recompilar SILO y sin necesidad de "parchar" el código, SILO reflejará los cambios .

Fueron éstas las razones principales por las que SILO fue implementado desde cero: primero para hacer un cargador que aprovechase las ventajas de las arquitectura SPARC, y segundo, que fuese pequeño y eficiente.

SILO sólo cuenta con el soporte para leer los dispositivos del PROM (*sbus*[7]) así que no importa si el dispositivo es un disco SCSI o un disco flexible; el soporte para entrada y salida y las llamadas a la biblioteca. Por supuesto que SILO debe contemplar y dar solución a las llamadas que hace la biblioteca ext2fs al sistema. Sin embargo, SILO toma algunas ideas de los otros cargadores como son el poder cargar otro sistema operativo (SunOS,

[7]Bus con los dispositivos SCSI. El PROM tiene la habilidad para recorrer este bus.

SOLARIS), el uso de una tabla estática para determinar los bloques de la segunda etapa y el uso de argumentos por omisión al momento de arrancar.

3.3.2 Descripción

La arquitectura SPARC cuenta con un ROM (PROM) muy completo que permite recorrer en un árbol al bus que contiene a todos los dispositivos, determinar qué hay en los registros y en la memoria, y accesar algún dispositivo de red, entre otras cosas.

En esta arquitectura, los primeros 16 sectores de los discos duros están reservados: el primero (sector 0) contiene lo que se conoce como la etiqueta del disco (*Disk Label*), que es la información general del disco duro (particiones usadas, tamaños). Los otros 15 sectores están reservados y se conocen como bloques de arranque o *bootblocks*, y es en este espacio precisamente en donde debe ir el cargador del sistema operativo.

Sin duda alguna el cargador maneja los recursos de la máquina de manera muy distinta a cualquier otro programa, ya que los programas y procesos que estamos acostumbrados a hacer y usar, cuentan con el respaldo de un kernel detrás de ellos, permitiéndoles hacer uso de los recursos por medio de llamadas al sistema, mientras que el cargador no. Su tarea es, precisamente, *cargar* el kernel. Por ejemplo, para abrir un archivo, comúnmente se hace con una llamada al sistema (*fopen*), y el sistema operativo debe de encargarse de abrir el dispositivo correcto, interpretar la ruta dondein el archivo se encuentra (*path*) y una serie de operaciones diversas. Cuando el cargador trata de abrir el archivo donde está el kernel, el cargador no puede dejarle todas estas operaciones a un sistema operativo (kernel) que no existe. Por esta razón el cargador debe tener sus propias rutinas de entrada/salida, y en general, de todas las llamadas al sistema.

3.3.3 Dos Etapas

Los sectores físicos de los discos SCSI son de 512 bytes y esto nos deja un total de 7.5Kb de espacio para hacer un cargador. Esta limitante nos lleva a hacer un cargador en dos etapas, la primera consiste de un programa muy pequeño

(4.4 Kb) y sencillo, que basado en una tabla estática carga los bloques de un segundo programa. Este segundo programa o segunda etapa es bastante más complejo y grande (90Kb), capaz de manejar distintos discos, diversas particiones además de manejar y entender el sistema de archivos nativo de Linux, el *ext2fs*.

Ambos programas están hechos casi en su totalidad en C, salvo el inicio del programa que está en ensamblador para SPARC (83 líneas).

Un cargador para arrancar un sistema operativo es básico y fundamental en el desarrollo del mismo. El cargador no es una tarea prioritaria ya que hay otras alternativas para cargar al sistema operativo. Por ejemplo, durante el desarrollo de Linux/SPARC, los desarrolladores arrancaban sus máquinas vía red. Pero aunque no es prioritaria, sí es indispensable para que el sistema operativo pueda funcionar en los discos locales de la máquina.

3.3.4 Instalación

Originalmente se necesitaban los compiladores cruzados tanto para SunOS como para SOLARIS; ahora sólo es necesario el de SunOS. El compilador cruzado deberá generar código para Linux/SPARC. Dentro de la distribución existe un subdirectorio `bin` que contiene las imágenes de ambas etapas del cargador ya compiladas y listas para correr en Linux/SPARC. Estos programas NO corren en SunOS o SOLARIS. Los nombres de los archivos son **first.image** y **second.image** para la primera y segunda etapa respectivamente.

Hay que copiar estos dos archivos a una partición formateada con ext2, y luego hay que ejecutar el comando `instboot`, todo esto bajo la hipótesis de que se está corriendo Linux-SPARC, ya sea por red o instalación vía CDROM:

```
instboot /dev/sda4 /mnt/first.image /mnt/second.image
         /mnt/oldloader
```

Por el momento, las dos imágenes deben ir en la misma partición ext2. Este programa de instalación deberá realizar la siguientes tareas:

1. Leer las características físicas del disco (/dev/sda).
2. Salvar el cargador original en un archivo dentro de la partición ext2 (/mnt/oldloader).
3. Escribir en los primeros 12 bloques de arranque la imagen de la primera etapa del cargador (first.image).
4. Escribir en el bloque de arranque 13 el mapa de bloques del cargador original.
5. Escribir en los bloques 14 y 15 el mapa de bloques de la segunda etapa.
6. Finalmente pedir los parámetros por omisión (default) de arranque dejando instalado el cargador.

Al rearrancar la máquina, el primer cargador despliega sus mensajes y rápidamente carga a la segunda etapa. Si esto no sucede, se puede restaurar el cargador original con la herramienta del sistema dd[8]. La segunda etapa contestará con el prompt "`silo:` " que entiende la siguiente sintaxis:

[prompath;no_de_partición]Nombre_del_kernel [argumentos]

[prompath] es una ruta típica del PROM de la SPARC (**/sbus/esp@....**). Si se da un nuevo disco, se le debe de especificar la partición. Sin la partición es imposible que SILO adivine cuál partición está formateada con ext2. La partición debe separarse por el carácter ";". Ejemplo:

/sbus/esp@0,800000/sd@1,0;4

no_de_partición es el número que determina en qué partición se encuentra el kernel. De esta manera, SILO puede calcular el sector físico y real donde se encuentra la información.

Nombre_del_kernel es el nombre de un archivo que sea un kernel, ya sea en formato *AOut* o *ELF*[9]. Generalmente este archivo se llama "vmunix",

[8] dd if=/mnt/oldloader of=/dev/sda bs=512

[9] De estos formatos se hablará mas adelante (sección 4.1.5)

"vmlinux" o "penguinkernel". Algunas veces los distintos kernels son guardados en un directorio; esto también es válido mientras el directorio pertenezca a la misma partición ("/kernels/vmlinux").

[kernel-args] son los argumentos que se le pasan al kernel.

Como ejemplo, veamos la línea de comando que usamos para arrancar a platypus.nuclecu.unam.mx:

```
/sbus/esp@0,80000/sd@1,0;4/kernels/vmunix
                    nfsroot=132.248.29.9:/usr/nfs/root
```

Nótese el uso del caracter ';' y de que no hay espacios entre la ruta del **PROM**, la partición y el nombre del kernel.

Capítulo 4

SILO: Implementación

4.1 Primera Etapa

4.1.1 Código en Ensamblador

Dentro de la implementación del cargador, el primer archivo que debemos analizar es el `str0.S` (*startup-code*), que contiene la información pertinente para levantar al cargador y posteriormente al sistema. Este código es, en principio, el equivalente a la función distinguida **main** del lenguaje C, que es la función principal, la función que inicializa la memoria y variables, y la que invariablemente se ejecuta primero. En este caso, la función **main** no aparece en el código ya que, al ser el cargador llamado desde ROM, se necesitan otros mecanismos para inicializar los segmentos de datos, memoria y código. Este archivo `srt0.S` es el que nos va a permitir inicializar todos los espacios de memoria y sobre todo, los apuntadores a funciones del ROM.

La primera etapa del cargador es una imagen de un archivo ejecutable con formato *AOut* [1] por lo que este programa consta básicamente de dos partes, el texto (código) y los datos (variables).

```
#if 0
    .section ".bss"
#else
    .data
```

[1] La sección 4.1.5 explica este formato de archivo binario ejecutable.

```
#endif
    .align 4
    .global _bss
_bss:
#if 0
    .section ".text"
#else
    .text
#endif
```

La misión básica y vital de este código es inicializar la pila para las futuras llamadas, localizar la dirección del ROM de la SPARC (PROM) y llamar a la rutina **bootmain** quien hará el resto del trabajo. Por supuesto se debe contemplar los posibles errores en caso de que arrancar un kernel sea imposible.

```
start:
    !  Set up a stack
    sethi %hi(start),%l0
    or %l0,%lo(start),%l0
    save %l0,-120,%o6
    ....
!  Llama a bootmain (regresa al PROM si hay error)
    call _bootmain
    or %i0, %g0, %o0 !  Prom vector address
```

Nótese que primero se llama a la rutina **bootmain** y luego se obtiene la dirección del PROM. Esto se debe a la manera como funciona el PC y el NPC de la SPARC (sección 1.2.1), de tal forma que para cuando **bootmain** es realmente llamado, la dirección del PROM ya ha sido obtenida y es utilizada.

4.1.2 Interacción con el PROM

La interacción con el PROM de la SPARC se va a hacer por medio de un apuntador, que contiene la dirección de dicho PROM. El código en ensamblador se ha ocupado ya de proveerla a la rutina `bootmain` que la recibe

en un parámetro llamado *promvec. Éste es un apuntador a una estructura que contiene varios elementos y funciones, dentro de los cuales los más importantes son:

```
struct linux_romvec {
    unsigned int pv_romvers;  /* vers (0, 2, 3) */
    struct linux_mem_v0 pv_v0mem; /* Descriptor de la
                memoria */
    char **pv_bootst; /* Comando de arranque,
                sd(0,0,0)vmunix */
    struct linux_dev_v0_funcs pv_v0devops; /* Opciones
                para los dispositivos*/
    char *pv_stdin; /* stdin */
    char *pv_stdout; /* stdout */
#define PROMDEV_KBD 0 /* input (teclado)*/
#define PROMDEV_SCREEN 0 /* output (pantalla) */
#define PROMDEV_TTYA 1 /* in-out (serial 0) */
#define PROMDEV_TTYB 2 /* in-out (serial 1) */
    int (*pv_getchar)(void);
    void (*pv_putchar)(int ch);
    int (*pv_nbgetchar)(void);
    int (*pv_nbputchar)(int ch);
    void (*pv_putstr)(char *str, int len);
    void (*pv_reboot)(char *bootstr);
    void (*pv_printf)(const char *fmt, ...);
    void (*pv_abort)(void); /* BREAK key abort */
    struct linux_arguments_v0 **pv_v0bootargs;
            /* Opciones de arranque */
    unsigned int (*pv_enaddr)(int d, char *enaddr);
            /* Direccion de la tarjeta de red */
};
```

Con esta estuctura, el código en C usará y accesará toda la información del PROM de la SPARC.

4.1.3 Rutinas de E/S

Dado que no se tiene un sistema operativo que nos permita hacer entrada y salida de los dispositivos, es necesario implementar estas rutinas desde cero. Se implementaron las rutinas de abrir, cerrar y leer información del disco, `dkopen`, `dkclose` y `dkread` respectivamente, por medio de una estructura llamada `Disk`, en la que se guarda la información relevante sobre el disco al

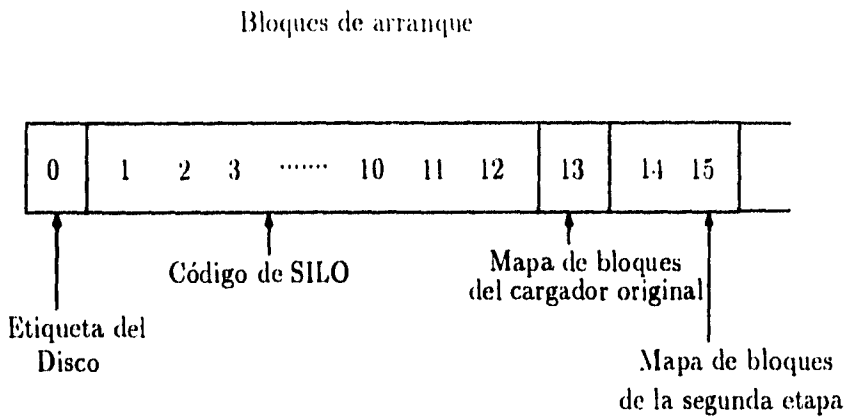


Figura 4.1: Organización de los bloques de arranque

que se desea accesar. La estructura de define a continuación:

```
typedef  struct  {
         int     fd_;
         int     seekp_;
         int     floppy_;
         Read0   read0_;
         Seek2   seek2_;
         Read2   read2_;
} Disk;
```

La parte más importante de esta estructura es el apuntador a una función de lectura del PROM (**read0_** o **read2_**); así, **dkread** va leyendo del disco en cuestión. La función **dkopen** entiende rutas del PROM[2], y esta rutina es la que proporciona el apuntador a la función de lectura dada por el PROM de la SPARC.

[2]Por ejemplo: `/sbus/esp@0,800000/sd@1,0`

4.1.4 Organización de los Bloques de Arranque

El cargador primario tiene un tamaño aproximado de 5 Kb, por lo que sólo utiliza hasta el bloque 12 de arranque. El cargador primario (primer paso) no necesita crecer más, y debe de leer la tabla de bloques del segundo paso desde alguno de los 15 sectores de arranque. Por esta razón, el bloque 13 se utiliza para guardar la información del cargador original (mapa de bloques) y los bloques 14 y 15 para guardar la información de la segunda etapa del cargador (figura 4.1).

Para almacenar la información que se desea guardar en los bloques 13, 14 y 15 se utiliza una estructura que se define de la siguiente manera:

```
struct  Block_Table{
        char                magic[10];     /* Firma */
        char                params[100];   /* Parámetros*/
        unsigned char       partno;        /* No. de Partición*/
        unsigned char       nsect;         /* Bloques/Sector*/
        unsigned short int  bs;            /* Tamaño del bloque*/
        unsigned long int   DiskOffset;    /* Desplazamiento*/
        unsigned long int   blocks[200];   /* Tabla de bloques*/
};
```

Basándose en la tabla de particiones (datos que están en la etiqueta del disco en el sector 0) y en el tamaño en kilobytes del bloque (*blocksize*), se calcula el verdadero lugar donde empieza la información de la partición (`DiskOffset`) y se le suma el número del sector que realmente se quiere leer. El tamaño del archivo que se puede guardar en la tabla de bloques (`blocks`) depende del tamaño en kilobytes de cada bloque; por ejemplo, si dicho tamaño es de 8Kb, se puede guardar un archivo de 8Mb aproximadamente.

```
DiskOffset = DiskLabel->ntrks * DiskLabel->nsect *
             DiskLabel->partitions[partno-1].start_cylinder;
```

El cargador original mide 7.5 Kb (o al menos ese es el tamaño total de todos los bloques de arranque), por lo que el bloque 13 se utiliza a menos de la mitad, y dado que la segunda etapa del cargador mide no más de 100Kb,

el espacio destinado es más que suficiente para albergar dichas tablas.

Una vez calculado el verdadero sector donde empieza la partición, éste se suma a la multiplicación de cada elemento en `Block_Table->blocks` con el tamaño del bloque (`Block_Table->bs`) y se cargan tantos bloques como sectores por bloque (`Block_Table->nsect`). De esta forma es como se determina en cuál bloque físico de 512b se encuentra cada pedazo de la segunda etapa.

```
Block_Table->nsect = Block_Table->bs / 512 ;
block_to_read = Block_Table->blocks * Block_Table->nesct
            + DiskOffset;
```

4.1.5 AOut/ELF

Linux maneja dos tipos de binarios, el primero es el *AOut*. En principio un binario AOut es una imagen de un fragmento de memoria. Es el formato del binario típico de los sistemas UNIX. Consta básicamente de una sección de texto (código) y una sección con datos (variables). Esta es la definición de un binario con este formato.

```
struct exec {
        int magic;
        int ltext;
        int ldata;
        int lbss;
        int lsym;
        int lentry;
        int x1;
        int x2;
};
```

El campo `magic` permite reconocer qué tipo de binario AOut es. A SILO le interesa que sea un binario estáticamente ligado, por lo que este campo deberá contener los siguientes valores en hexadecimal: `01 03 01 07`.

Al transferir el control al segundo cargador, es necesario eliminar el encabezado del binario AOut y para ello se calcula la longitud y el corrimiento para sólo tener código ejecutable, sin encabezados.

```
off = sizeof(struct exec);
len = hp.a->ltext + hp.a->ldata;
```

El segundo tipo de binario es el *ELF*. La diferencia básica entre el ELF y el AOut es que el ELF ya no es una imagen de memoria (que sólo contiene un segmento de datos y uno de texto), sino que se divide en secciones y, no importando el orden, cada una de estas secciones puede ser de texto o de datos. Aunque es más complicado el manejo de los binarios en formato ELF, a SILO sólo le interesa poder reconocer que se trate de un binario ELF ligado estáticamente y ejecutable. De manera análoga a los binarios AOut, los binarios ELF tienen su estructura definida así:

```
typedef         struct          elfhdr{
                unsigned char   e_ident[EI_NIDENT];
                Elf32_Half      e_type;
                Elf32_Half      e_machine;
                Elf32_Word      e_version;
                Elf32_Addr      e_entry;
                Elf32_Off       e_phoff;
}Elf32_Ehdr;
```

En este caso no se tiene una única firma (`magic`), sino un arreglo que identifican al binario ELF (`e_ident[EI_NIDENT]`). Cuatro de estas entradas permiten determinar qué tipo de binario ELF es; si alguna de estas firmas no es válida[3], entonces se aborta el cargado de bloques y se regresa al PROM de la SPARC. También hay que revisar que sea un ELF de 32 bits[4], y después verificar que exista un segmento de texto[5] (código ejecutable). Depués se cargan todas las secciones del binario ELF no importando si son texto o datos, ya que el binario automáticamente controla la ejecución del mismo.

[3]Válida para SILO, esto es, ligado estático.

[4]Linux trabaja en modo de 32 bits.

[5]Como es un binario ELF es válido que todas las secciones sean de datos, pero debemos revisar que por lo menos exista una de texo.

Qué es dato y qué es texto lo determina el mismo binario.

4.1.6 Llamando a la segunda etapa

Primero hay que detectar si los bloques 14 y 15 son bloques de arranque válidos para SILO. Para ello se usa una firma que en principio sólo cuenta con la palabra "SILO" y un espacio de 6 carácteres más[6] para incluir otro tipo de información, como es la versión de SILO, o en su defecto si se desea un prompt para arranque o que simplemente la máquina levante con los parámetros por omisión. Por ahora, SILO siempre pregunta con un *prompt* (`silo:`), pero esto no debe ser necesario, pues algunas veces la máquina debe levantarse por sí sola (cuando se va la luz por ejemplo). Esta firma `Block_Table->magic` sirve para reconocer los bloques de arranque y para determinar cómo se va a cargar la segunda etapa.

Una vez que se ha establecido que los bloques son válidos, se procede a cargar sólo el primer bloque del segundo cargador y a verificar si se trata de un archivo ejecutable y ligado estáticamente. Un cargador ligado dinámicamente no nos sirve ya que carecemos de un sistema operativo que nos respalde con el cargado de la bibliotecas en tiempo de ejecución.

Cuando se ha determinado que el programa a cargar (segundo paso) es un binario. ya sea en formato *AOut* o *ELF*, se carga el resto de los bloques y se transfiere el control a este nuevo programa. El primer cargador es totalmente remplazado y olvidado, y el segundo cargador es ahora el encargado de hacer la parte mas difícil del proceso: entender al sistema de archivos.

4.2 Segunda Etapa

4.2.1 Unix_IO Manager

Una de las razones principales para utilizar la biblioteca ext2fs es que ésta tiene la peculiaridad de que en su estructura se puede redefinir la interface para manejar la entrada y salida de los datos por medio de llamadas indirectas

[6] Esto hace un total de 10 carácteres, ver en la estructura (`Block_Table->magic`)

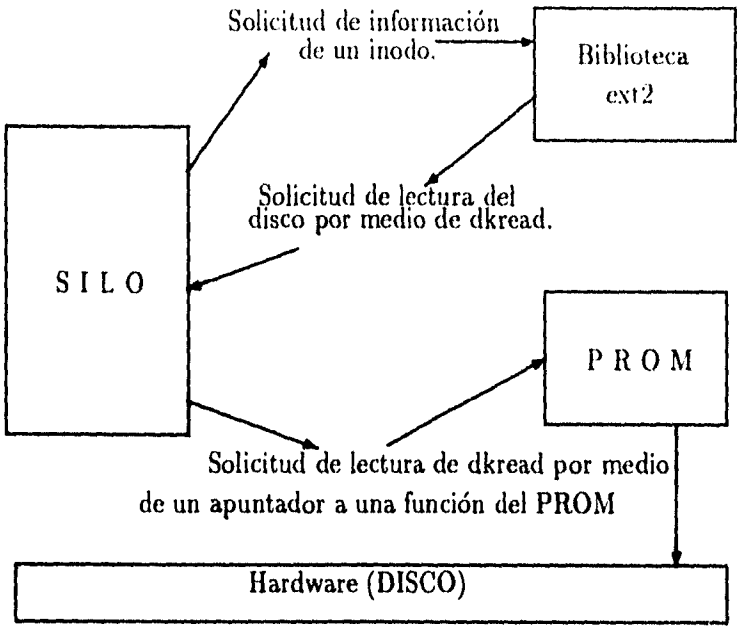


Figura 4.2: SILO interactuando con la biblioteca ext2 y el PROM

(*call-backs*). En esencia, estas llamadas se hacen por medio de un apuntador a una rutina; así, la biblioteca queda compilada con el apuntador (que hará la llamada a esta rutina). Cuando se compila el programa que hace uso de la biblioteca, este programa sólo tiene que definir hacia cuál función se debe apuntar, y la biblioteca, al usar el apuntador, llama a la rutina definida por el programa. haciendo uso de las variables y parámetros del programa, como si la biblioteca hubiese sido compilada con todo este código.

Por ejemplo (figura 4.2), si SILO quisiera conocer el primer inodo del archivo **/vmlinux**, SILO hace una llamada a la biblioteca ext2fs solicitando información referente al sistema de archivos; la biblioteca hace a su vez una llamada a SILO para poder leer la información; SILO hace una llamada a **dkread** que a su vez llama al PROM de la SUN, quien devuelve la información requerida, regresando la información a través de la mencionada cadena. La estructura que define al manejador de entrada y salida para UNIX se define a continuación.

```
static struct  struct_io_manager struct_unix_manager = {
               EXT2_ET_MAGIC_IO_MANAGER,
               "Unix I/O Manager",
               unix_open,
               unix_close,
               unix_set_blksize,
               unix_read_blk,
               unix_write_blk,
               unix_flush
};
```

De todas formas cada apuntador deberá tener un prototipo y cada redefinición de las mismas deberá respetar esos prototipos. Pero esto no se extiende a la implementación y ésta es la parte medular de usar las llamadas indirectas.

```
static errcode_t linux_read_blk(io_channel channel, unsigned
                        long block, int count, void *data);

static errcode_t unix_read_blk(io_channel channel, unsigned
                        long block, int count, void *data);
```

Mismo prototipo para las dos funciones

Originalmente estas rutinas usan llamadas al sistema, pero dado que nosotros carecemos del sistema operativo en si, estas llamadas deben de ser reemplazadas por llamadas propias, es decir, llamadas a **dkread**, **dkopen** y **dkclose**. Como parte del prototipo de las funciones, se requiere que usen un canal de entrada y salida; este canal es otra estructura que permite compartir datos. La idea fundamental es que, aunque todas las rutinas del manejador de entrada y salida (*unix_io_manager*) tienen el mismo prototipo, algunas veces se requieren nuevos parámetros o algunos dejan de ocuparse; estos canales dan flexibilidad para mandar parámetros a las funciones e intercambiar datos entre las funciones. Este canal deberá tener la información para saber cuál es el manejador de entrada y salida que usará, por ejemplo el **unix_io_manager** del sistema o el **linux_io_manager** de SILO.

```
struct  struct_io_channel  {
        int                magic;
        io_manager         manager;
        char               *name;
        int                block_size;
        errcode_t          (*read_error)(io_channel channel,
                           unsigned long block,
                           int count,
                           void *data,
                           size_t size,
                           int actual_bytes_read,
                           errcode_io_t error);
        errcode_t          (*write_error)(io_channel channel,
                           unsigned long block,
                           int count,
                           const void *data,
                           size_t size,
                           int actual_bytes_written,
                           errcode_t error);
        int                reserved[16];
        void               *private_data;
};
```

Estas estructuras determinan de manera fundamental tanto el dispositivo a leer y sus características, así como los métodos para hacer uso de la información. Algunas de las variables definidas en estas estructuras, no son usadas por las funciones **dk**. Sin embargo, en esta última estructura, son vitales los apuntadores a las funciones **write_error** y **read_error**, así como el manejador de entrada y salida (**manager**).

Parte de las llamadas al sistema, tienen que ser sustituidas por llamadas a la función **dkread** ya que es ahora ésta quien hará las lecturas sobre el disco. Sin embargo nos enfrentamos a tres problemas por resolver:

1. Cuando la biblioteca ext2 requiere leer la información de los nodos-i, se tiene que el tamaño del bloque (*blocksize*) siempre es 1024 y pide leer cierto número de bytes (no de bloques).

2. Cuando la biblioteca ext2 pide la información de un archivo, el tamaño del bloque es variable (depende de la densidad con la que se formateó esa partición, 1024, 2048, etc) y pide leer cierto número de bloques, no de bytes.

3. La rutina dkread sólo lee bloques físicos (no bloques del sistema de archivos) de 512 bytes.

Ante esta situación, se define que:

1. Si el argumento de la función es negativo, entonces se desea leer ese número de bytes.

2. Si el argumento es positivo entonces se desea leer ese número de bloques (del sistema de archivos).

Y en ambos casos se hace la conversión necesaria para que dkread lea correctamente la información.

```
size = (count < 0) ?  -count :  count*bs;
nsect = bs / 512;
if (size<512){
    int i;
    char buff[512];
    xprintf ((int) &cons0, ``DEBUG: Read size smaller than
                            512 (%d)'',size);
    dkread (&dk0, buff, 1, block*nsect+DiskOffset);
    memcpy (data, buff, size);
} else {
    dkread (&dk0, data, size/512, block*nsect+DiskOffset);
}
return 0;
```

4.2.2 Malloc

Parte del problema de ligar contra la biblioteca ext2fs, era llenar todos los huecos de las llamadas a funciones y rutinas que hace la biblioteca misma, desde llamadas al sistema (unix_io_manager) hasta simples llamadas a funciones tan cotidianas en C como *malloc*.

Después de hacer un estudio del código de la biblioteca, se determinó que el uso de *malloc* no era muy complejo. Generalmente después de hacer un requerimiento de memoria, el espacio es liberado antes de hacer otro pedido, así que se hizo un *malloc* lo más primitivo y sencillo posible, para así enfocar todos los esfuerzos en el cargador y no en la implementación de un malloc complicado mas allá de lo necesario.

Se supone que el código del segundo cargador no debe pasar del lugar 0x680000 de memoria. Así que todo el código del *malloc* se limita a dar memoria después de este lugar alineándola correctamente.

```
/* Memory allocation */
void *malloc (int size)
{
    char *caddr;
    caddr = malloc_ptr;
    malloc_ptr += size;
    last_alloc = caddr;
    malloc_ptr = (char *) ((((unsigned int)malloc_ptr) + 3)
                          & (3));
    return caddr;
}
```

Dado que SILO hace un requerimiento de memoria y lo libera antes de solicitar otro, la rutina **free** fue implementada de esta manera. Libera la memoria sólo si fue pedida recientemente y no ha habido otro pedido. Estas funciones fueron hechas como un módulo aparte por si en algún momento se desea reescribir el código por uno más eficiente. Para los requerimientos de SILO esta implementación es más que suficiente.

```
/* Free memory */
void free (void *m)
{
    if (m == last_alloc)
        malloc_ptr = last_alloc;
}
```

4.2.3 Búsqueda de Archivos

Una vez que han quedado determinados los parámetros de arranque, y se tiene el nombre del kernel a arrancar, SILO debe realizar dos tareas claves: la primera es buscar el archivo (por nombre) dentro del sistema de archivos y determinar sus inodos; la segunda es cargar a la memoria cada uno de los bloques que forman el archivo. Aquí es donde entra la interacción con la biblioteca ext2.

La figura 4.3 muestra esquemáticamente esta interacción. Algorítmicamente se debe realizar los siguientes pasos:

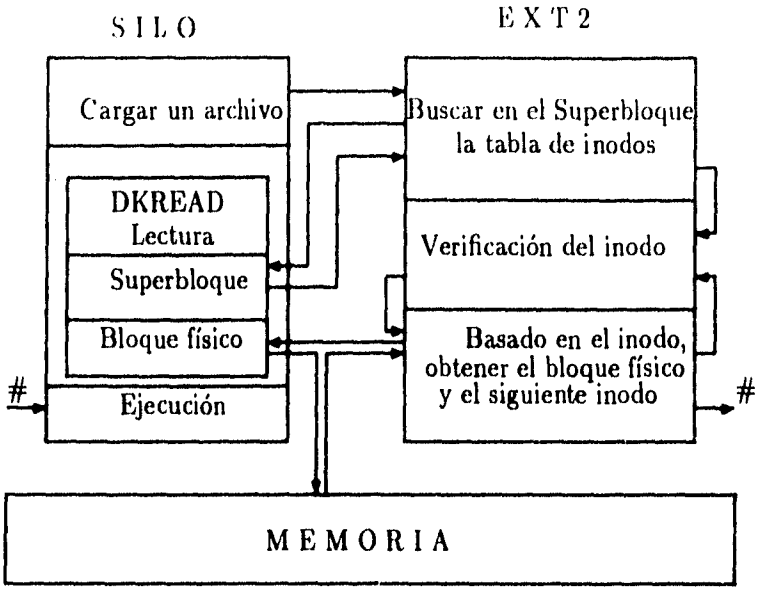


Figura 4.3: Búsqueda de un archivo por medio de la biblioteca ext2

1. Obtener el Superbloque.
2. Del superbloque, obtener el primer inodo del archivo.
3. Verificamos que sea un inodo válido.
4. Con el inodo, sabemos en qué bloque físico del disco está la información y la cargamos a memoria.
5. Obtenemos el siguiente inodo.
6. Repetimos desde el punto 3 hasta que se acaben los inodos del archivo.

SILO invoca a funciones de la biblioteca y a través del **linux_io_manager** la biblioteca invoca a SILO.

```
int silo_open_ext2 (char *device)
{
    int retval;
```

```
    retval = ext2fs_open (device, EXT2_FLAG_RW, 0, 0,
                linux_io_manager, &fs);
    retval = ext2fs_read_inode_bitmap(fs);
    retval = ext2fs_read_block_bitmap(fs);
    root = cwd = EXT2_ROOT_INO;
    return 1;
}
```

4.2.4 Varios discos/particiones

Es muy común que las máquinas tengan varios discos duros, y que tengan la información en distintas particiones. Ante esta situación, SILO debe de implementar el manejo correcto de ellas, ya que la primera etapa siempre estará en los bloques de arranque, pero la segunda etapa puede estar en una partición, mientras el kernel a cargar en otra. O mas aún, la segunda etapa puede estar en un disco mientras que el kernel está en otro. Lo que siempre debe de suceder es que la primera etapa y la segunda etapa estén en el mismo disco, ya que la primera etapa no entiende particiones, sistemas de archivos o discos y tan sólo lee a memoria un mapa de bloques (segunda etapa) y los ejecuta. Por ejemplo:

```
# df
Filesystem  1024-blocks  Used    Available  Capacity  Mounted o
/dev/hda1   240023       224569  2645       99%       /usr
/dev/hda2   125901       109737  9445       92%       /home
/dev/sda1   240023       224569  2645       99%       /usr/loca
/dev/sdb5   240023       224569  2645       99%       /kernels
#
```

La segunda etapa puede cargar el kernel desde la raíz (**/vmlinux**), o desde cualquier otro punto (**/kernels/vmlinux-1.3.19**).

4.2.5 Cargando otros sistemas operativos

Parte del éxito de Linux en la PC consistió en que su cargador (LILO) permite arrancar la máquina con otro sistema operativo distinto a Linux (DOS, OS/2). De tal forma que la gente reticente a usar Linux, de alguna manera no siente que pierde lo que ya se tiene por usar este nuevo sistema operativo, exceptuando un poco de espacio en disco duro.

Por esta razón, SILO debe contemplar el poder arrancar la máquina con otro sistema operativo, como lo es SunOS o SOLARIS. Aunque Linux en la SPARC corre binarios de SunOS, y en un futuro próximo binarios de SOLARIS, hay muchos otros sistemas operativos para SUN como NextStep y Mach. SILO podría implementar el sistema de archivos de cada uno de estos sistemas operativos, pero se tienen varios problemas:

1. Problemas con las licencias: en algunos casos el código no está liberado y está protegido con Derechos Reservados.
2. SILO crecería muchísimo y sería más difícil mantenerlo y entenderlo.
3. Si se implementase un nuevo sistema de archivos o algún nuevo sistema operativo para SUN con su propio sistema de archivos, SILO tendría que ser modificado para poder cargar este nuevo sistema.

Dado que queremos que la gente no pierda lo que ya está instalado (SunOS, SOLARIS), partimos de la hipótesis que su máquina ya tiene un sistema operativo instalado[7] y en pleno funcionamiento, esto es, que ya tiene su cargador instalado en los bloques de arranque; así que la solución a los problemas anteriores consiste en:

1. Salvar los bloques de arranque (*bootblocks*) en un archivo (*old_loader*) en la partición formateada con el sistema de archivos ext2.
2. Agregar en el sector 13 de los bloques de arranque el mapa de los sectores de este archivo (*old_loader*).

Estos sectores son sectores del sistema de archivos, no son sectores físicos del disco. De tal forma que si en el prompt de SILO (`silo:`) se teclea la palabra reservada **sunos**, SILO cargará a memoria estos bloques y ejecutará el cargador original de la máquina como si hubiese sido llamado por el PROM y SILO nunca hubiese sido ejecutado.

```
/* Loading OLD loader */
int load_sunos(){
```

[7]Esto no es necesariamente cierto, así que SILO implementa todo esto como una opción.

```
        char buffer[1024];
        struct Block_Table *sbt;
        int i;
        char *b=(char *) 0x4000;
        sbt = (struct Block_Table *) &buffer;
        if (dkread (&dk0, buffer, 2, OLD_LOADER_SEC)!=2) {
            xprintf ((int) &cons0, ``Can not read old loader
                                blocks....aborting´´);
            return 0;
        }
        xprintf ((int) &cons0, ``Magic = %s, Partition = %d, Offset
            sbt->magic, (int)sbt->partno, sbt->DiskOffset);
        xprintf ((int) &cons0, ``Block Size = %d, NSect = %d´´,sbt-
                    (int)sbt->nsect);
        xprintf ((int) &cons0, ``Loading old lodaer (probably SunOS
        for (i=0; sbt->blocks[i]!=0; i++){
            dkread (&dk0, b, sbt->nsect,
                    sbt->blocks[i]*sbt->nsect+sbt->DiskOffset);
            b+=512*sbt->nsect;
            xprintf ((int) &cons0,".");
        }
        return 1;
}
```

Esta manera resultó más eficiente de lo esperado, ya que en realidad no importa si el sistema operativo original de la máquina es SunOS, SOLARIS o NextStep. Teóricamente no hay diferencia entre ellos y no hay necesidad de agrandar SILO con el código para los sistemas de archivos de los distintos sistemas operativos. Esto es, en teoría SILO puede cargar a cualquier otro sistema operativo para SPARC.

4.3 Piedras Angulares (*Milestones*)

4.3.1 Cargar de manera cruda un kernel

El cargador desarrollado por Peter Zaitcev, consistía sólo en el código suficiente para que cargara un kernel desde disco flexible. En un disco flexible de 3.5 alta densidad sin sistema de archivos, se escribe el kernel a partir del sector 18 de manera cruda. Así, este primitivo cargador que va en los primeros 18 sectores, sólo debe de cargar todo lo que haya en el disco a partir

de dicho sector; la raíz del sistema de archivos es montada vía NFS. El primer problema a resolver era ver si parte del código para la entrada y salida podía funcionar con un disco SCSI en lugar de un disco flexible.

Para conseguir lo anterior, se usa un disco fijo SCSI sin particionar y sin sistema de archivos; con la herramienta **dd** ponemos un kernel de manera cruda (*raw*) en el disco[8], se modifica el código para leer del disco SCSI y se arranca el kernel. De esta manera se empieza a ver qué necesidades nuevas surgen al hacer el cambio a un disco y cómo se va a manejar el SCSI. El kernel escrito de manera cruda sobre el disco SCSI arranca y monta la raíz del sistema de archivos por NFS, como era de esperarse. Una vez que se ha determinado que el kernel de Linux puede ser arrancado de un disco de manera cruda, se investiga cómo funcionan los bloques de arranque de la SPARC y con cuantos bloques contamos realmente.

4.3.2 Cargar al cargador

Una vez que se vio que se podía cargar un kernel de manera cruda, la siguiente piedra angular a completar era cargar al cargador mismo en lugar del kernel. Este fue un momento crítico, ya que si no era posible realizar tal tarea, entonces **SILO** no hubiera podido hacerse en dos etapas, no al menos con el código que ya se tenía. El cargador fue puesto de manera cruda en el disco de igual forma que el kernel y se arrancó.

El cargador se autocargó satisfactoriamente y se determinó que es posible hacerlo en dos etapas. De la misma forma se llegó a las siguientes conclusiones:

1. La segunda etapa deberá ser cargada por medio de una tabla que contenga cada uno de los bloques físicos que conforma el archivo, ya que la primera etapa no entiende de sistemas de archivos y de alguna manera se debe saber qué bloques pertenecen a la segunda etapa, para llamarlos a memoria y ejecutarlos.

[8]El disco no está montado, no tiene sistema de archivos: se hace directamente sobre el dispositivo `/dev/sda`.

2. El mapa de bloques de la segunda etapa debe estar en algún espacio sobrante en los bloques de arranque, ya que no hay otro espacio reservado para éstos.

3. La primera etapa no debe contener más código y cualquier otra operación deberá dejarse a la segunda etapa. Se debe medir el tamaño de la primera etapa y planear estructuras que satisfagan la necesidades antes mencionadas (mapeo de bloques, espacio en bloques de arranque).

4.3.3 Ligado con el ext2fs

La estructura clave de SILO estaba ya formada: se tenían las rutinas básicas para manipular los sectores físicos del disco haciendo uso de las rutinas de entrada y salida de dispositivos a través del PROM. La primera etapa estaba casi concluída y la segunda etapa, al ser un programa que estaría en un sistema de archivos (ext2), podía ser casi tan grande y complicada como fuera necesario. La única limitante que se tenía era el espacio sobrante en los bloques de arranque, pero era un dificultad fácil de solucionar.

La siguiente piedra angular a completar era escribir todo el código para soportar el sistema de archivos ext2 de Linux. Sin embargo, escribir el código desde cero era peligroso por varias razones, pero sin duda la más importante es que se podían cometer los mismos errores que con MILO, ya que las ideas básicas para soportar el sistema de archivos ext2 serían tomadas del kernel. Esto traería problemas de compatibilidad, además de ineficiencia en el código.

La existencia de la biblioteca GNU ext2fs abrió la posibilidad de intentar ligarla con el programa SILO. Ligar una aplicación a nivel de usuario o programa (*user-level*) con esta biblioteca es relativamente sencillo. Sin embargo, ligar un cargador con esta biblioteca no fue tarea fácil ya que no se cuenta con el sistema operativo para que realice las tareas propias del sistema como son: ligado dinámico, rutinas de entrada/salida, y acceso a dispositivos. Por lo tanto, al ligar contra esta biblioteca, no había garantía alguna de que el código, aunque compilase, tuviese sentido, es decir funcionase correctamente.

Se hizo un estudio del funcionamiento de la biblioteca y se determinaron las llamadas que hace al sistema. Se evaluó la posibilidad de sustituir estas llamadas con código propio y se detectaron las llamadas a la biblioteca

estándard de C. Se escribió el código de lo que es nuestro manejador de entrada/salida que haga uso de las funciones `dk` (`linux_io_manager`) y se sustituyó el manejador del sistema (`unix_io_manager`) por este nuevo manejador.

4.3.4 Ligado contra la biblioteca estándard de *C*

En realidad el cargador casi no puede hacer uso de las rutinas estándard de entrada/salida (`stdio.h`, `stdlib.h`) ya que la mayor parte de este código hace uso de llamadas al sistema. Por ejemplo, no se puede hacer uso de **`printf`** o **`malloc`**. SILO en sí no usa ninguna de estas funciones, y si acaso alguna era requerida, se tenía que escribir de nuevo.

Sin embargo, la biblioteca ext2 hace referencia a muchas de las rutinas de la biblioteca estándard de C, por lo que se tuvieron que crear sólo los prototipos de las funciones requeridas, para después ligar contra esta biblioteca de manera estática. Así, el compilador es engañado e incluye el código objeto de las rutinas antes declaradas, pareciendo que el código fue escrito en el código fuente. Las rutinas que hacen uso de llamadas al sistema forzosamente tienen que ser totalmente reescritas como **`malloc`**, **`sprintf`** y **`printf`** (que en realidad se llama **`xprintf`**).

Apéndice A

El Código Fuente

En el presente apéndice se incluye sólo el código fuente que se consideró más importante, ya que el total de la distribución de SILO es demasiado grande para incluirla completa. La distribución completa de este trabajo se anexa en un disco flexible de alta densidad formateado con el sistema de archivos ext2 y en formato `tar.gz`. Para ver la distribución se debe montar la undad de dsico flexible:

```
# mount -t ext2 /dev/fd0 /mnt
```

También se puede obtener la distribución via FTP en `vger.rutgers.edu` que es el sitio oficial de la distribución Linux/SPARC.

A.1 Primera Etapa

A.1.1 ./silo-0.5.5/first/main.c

```
/*
 * Linux Loader for SPARC
 *
 * Linux Sparc Loader First Stage
 * Mauricio Plaza
 * Based on Pete A. Zaitcev's floppy lodaer
 *
 *
 */
#include <linux/elf.h>
#define ELFDATA2MSB   2
#include <asm/openprom.h>
```

```
/* We carry own a.out header for ease of cross-compilation.  */
struct exec {
        int magic;
        int ltext;
        int ldata;
        int lbss;
        int lsym;
        int lentry;
        int x1;
        int x2;
    };
#ifndef CONSOLE_H
#include "console.h"
#endif
#ifndef CMDLINE_H
#include "cmdline.h"
#endif
#include <storage.h>
extern xprintf(int filog, char *fmt, ...);
/** extern unsigned csum(const unsigned *p, unsigned l); **/
typedef int (*Read0)(int dev, int num_blks, int blk_st,
                     char* buf);
typedef int (*Seek2)(int, int, int);
typedef int (*Read2)(int, char *, int);
typedef struct {
        int fd_;
        int seekp_;
        int floppy_;
        Read0 read0_;
        Seek2 seek2_;
        Read2 read2_;
} Disk;
extern int dkopen(Disk *, const struct linux_romvec *,
                  Console *);
extern int dkread(Disk *, char *, int nblk, int blk);
extern void dkclose(Disk *, const struct linux_romvec *);
int mult(int a, int b){
    int i, res=0;
    for (i=0; i<b; i++) res+=a;
    return res;
}
/* Here we are launched */
int bootmain(struct linux_romvec *promvec){
        static Console cons0;
        static Disk dk0;
        static Cmdline cmd0;
        char *buff;
        unsigned off;
        int len;
        union {
```

```
        char *b;
        struct exec *a;
        Elf32_Ehdr *e;
} hp;
int rc;
char rotator[5] ={".oOo."};
char block_buff [1024];
        /* Reading from sectors 14-15 */
struct Block_Table *bt;
/*
* Create fake auto objects.
*/
dpset(&cons0, promvec);
cmdcons(&cmd0);
if (dkopen(&dk0, promvec, &cons0) == -1) {
        xprintf((int) &cons0, "ERROR: Cannot open disk/n");
        return 0;
}
buff = (char *) 0x4000;
bt  = (struct Block_Table *) &block_buff;
/* Support for reading OLD_LOADER_SEC Info
            instead of my loader */
xprintf ((int) &cons0, "/nSILO  First  Stage  Boot  Loader/n");
rc = dkread(&dk0, block_buff, 2, SECOND_STAGE_SEC);
if (rc!=2) goto eread;
rc = dkread (&dk0, buff, bt->nsect, mult(bt->blocks[0],
            bt->nsect)+bt->DiskOffset);
if (rc != bt->nsect) goto eread;
hp.b = buff;
if (hp.a->magic == 0x01030107) {
    off = sizeof(struct exec);
    len = hp.a->ltext + hp.a->ldata;
    xprintf((int) &cons0, "%d/0x%x bytes a.out/n",
            len, len);
} else {
    if (hp.e->e_ident[EI_MAG0] == ELFMAG0 &&
        hp.e->e_ident[EI_MAG1] == ELFMAG1 &&
        hp.e->e_ident[EI_MAG2] == ELFMAG2 &&
        hp.e->e_ident[EI_MAG3] == ELFMAG3) {
          if (hp.e->e_ident[EI_CLASS] != ELFCLASS32) {
              xprintf((int) &cons0, "Not a 32bits ELF/n");
              return 0;
          }
          if (hp.e->e_ident[EI_DATA] != ELFDATA2MSB) {
              xprintf((int) &cons0, "Not an MSB ELF/n");
              return 0;
          }
```

```
                {
                    Elf32_Phdr *p = (Elf32_Phdr *) (hp.b + hp.e->e_phoff);
                    if (p->p_type != PT_LOAD) {
                          xprintf((int) &cons0, "Cannot find
                              loadable segment/n");
                          return 0;
                    }
                    off = p->p_offset;
                    len = p->p_filesz;
                }
                xprintf((int) &cons0, "%d/0x%x bytes ELF/n",
                        len, len);
            } else {
                xprintf((int) &cons0, "Unknown image format/n");
                return 0;
            } /* Not ELF....No aout */
        }
        buff+=512*bt->nsect;
    {
        int i;
        xprintf ((int) &cons0, "Loading Second Stage Blocks:  ");
        for (i=1; bt->blocks[i]!=0; i++){
            if (!  (i % 5)) xprintf ((int) &cons0, "%c/b",
                        rotator[(i/5) % 5]);
            rc = dkread (&dk0, buff, bt->nsect, mult(bt->blocks[i],
                        bt->nsect)+bt->DiskOffset);
            buff +=512*bt->nsect;
        }
    }
        xprintf((int) &cons0, "/n%d/0x%x bytes to move/n",
                    len, len);
        {
                unsigned *f, *t;
                int cn;
                t = (unsigned *) 0x4000;
                f = t + off/sizeof(int);
                cn = len;
                while (cn > 0) {
                        *t++ = *f++;
                        *t++ = *f++;
                        *t++ = *f++;
                        *t++ = *f++;
                        cn -= 16;
                }
        }
        dkclose(&dk0, promvec);
```

```
        /* sendparam(promvec, &cmd0, &cons0);  */
        xprintf ((int) &cons0, "/nLoading Second Stage..../n");
        return 0x4000;
eread:
        xprintf((int) &cons0, "ERROR: Read error/n");
        return 0;
}
int dkopen(Disk *t, const struct linux_romvec *promvec,
            Console *con){
        static char name0[11];
        char *name;
        t->fd_ = 0;
        t->seekp_ = -1;
        t->floppy_ = 0;
        t->read0_ = 0;
        t->seek2_ = 0;
        t->read2_ = 0;
        if (promvec->pv_romvers == 0) {
                struct linux_arguments_v0 *ap = *promvec->pv_v0bootargs;
                char *s = name0;
                int unit;
                *s++ = ap->boot_dev[0];
                *s++ = ap->boot_dev[1];
                *s++ = '(';
                *s++ = (ap->boot_dev_ctrl & 07) + '0';
                *s++ = ',';
                if ((*s = ap->boot_dev_unit/10 + '0') != '0')
                                s++;
                *s++ = ap->boot_dev_unit%10 + '0';
                *s++ = ',';
                *s++ = (ap->dev_partition & 07) + '0';
                *s++ = ')';
                *s = 0;
                xprintf((int) con, "%s/n", name0);
                t->fd_ = (*promvec->pv_v0devops.v0_devopen)(name0);
                if (t->fd_ == 0 || t->fd_ == -1) return -1;
                t->read0_ = promvec->pv_v0devops.v0_rdblkdev;
                if (name0[0] = 'f' && name0[1] == 'd')
                                t->floppy_ = 1;
        } else {
                int rc;
                xprintf((int) con, "%s/n", *promvec->pv_v2bootargs.bootpath);
                t->fd_ = (*promvec->pv_v2devops.v2_dev_open)
                        (*promvec->pv_v2bootargs.bootpath);
                if (t->fd_ == 0 || t->fd_ == -1) return -1;
                t->seek2_ = promvec->pv_v2devops.v2_dev_seek;
                t->read2_ = promvec->pv_v2devops.v2_dev_read;
                /** rc = (*t->seek2_)(t->fd_, 0, 18*512); **/
                /** xprintf((int) con, "%x/n", rc); **/
```

```
        }
        return 0;
}
int dkread(Disk *t, char *buff, int size, int blk){
        if (t->read0_ != 0) {
                t->seekp_ = blk;
                return (*t->read0_)(t->fd_, size, blk, buff);
        }
        if (t->read2_ != 0) {
                int rc;
                if (t->seekp_ != blk) {
                        if ((*t->seek2_)(t->fd_,0,blk*512)==-1)
                                return -1;
                        t->seekp_ = blk;
                }
                rc = (*t->read2_)(t->fd_, buff, size*512);
                if (rc == size*512) return size;
        }
        return -1;
}
void dkclose(Disk *t, const struct linux_romvec *promvec){
        if (promvec->pv_romvers == 0) {
                /* Free drive for a root disk.  */
                (*promvec->pv_v0devops.v0_devclose)(t->fd_);
        } else {
                (*promvec->pv_v2devops.v2_dev_close)(t->fd_);
        }
}
```

A.1.2 ./silo-0.5.5/first/printf.c

```
/**
   ** Standalone printf.
   **/
   /*
   * Main purpose of this program is to run on v3 PROMs which have
   * no printf included.
   */
   #include <stdarg.h>
   static void prf(int, char*, va_list);
   extern void dps(int,char*,int);
   static void printn(int,long,int);
   static void uputo(char,int);
   #define putchar(c,file)  uputo((char)(c),file)
   /*
   * Scaled down version of C Library printf.
```

```
* Only %c %s %u %d (==%u) %o %x %D %O are recognized.
*/
xprintf(int filog, char *fmt, ...)
{
        va_list x1;
        va_start(x1,fmt);
        prf(filog,fmt,x1);
        va_end(x1);
}
static void
prf(int filog, char *fmt, va_list adx)
{
        register c;
        char *s;
        for(;;) {
                while((c = *fmt++) != '%') {
                        if(c == '/0') {
                                putchar(0,filog);
                                return;
                        }
                        putchar(c,filog);
                }
                c = *fmt++;
                if(c == 'd' || c == 'o' || c == 'x' || c == 'X'){
                        printn(filog, (long)va_arg(adx,unsigned),
                                c=='o'? 8:  (c=='d'?  10:16));
                } else if(c == 'c') {
                        putchar(va_arg(adx,unsigned), filog);
                } else if(c == 's') {
                        s = va_arg(adx,char*);
                              while(c = *s++)
                                putchar(c,filog);
                      } else if (c == 'l' || c == 'O') {
                        printn(filog, (long)va_arg(adx,long),
                                c=='l'?10:8);
                }
        }
}
/*
* Print an unsigned integer in base b, avoiding recursion.
*/
static void
printn(int filog, long n, int b)
{
        char prbuf[24];
        register char *cp;
```

```
        if (b == 10 && n < 0) {
                putchar('-',filog);
                n = -n;
        }
        cp = prbuf;
        do
                *cp++ = "0123456789ABCDEF"[(int)(n%b)];
        while(n = n/b & 0x0FFFFFFF);
        do
                putchar(*--cp, filog);
        while(cp > prbuf);
}
/*
* This part is rewritten by Igor Timkin <ivt@msu.su>.  Than I
* rewritten it again but kept the MISS style, originated from
* Wladimir Butenko.  --P3
*/
#define OBSIZE      200
static void
uputo1(char c,int filog)
{
/* P3:  This buffer can be static while
xprintf flushes it on exit.  */
static char       buff[OBSIZE+1];
static int        ox;
if ((buff[ox]=c) == 0) {
    dps(filog,buff,ox);  ox=0;
} else {
    if (++ox>=OBSIZE) {
      buff[ox]=0;
      dps(filog,buff,ox);  ox=0;
    } /*if*/
}
} /*uputo1*/
static void
uputo(char c,int filog)
{
        if (c == '/n')
                uputo1('/r', filog);
        uputo1(c, filog);
}
}
```

A.2 Segunda Etapa

A.2.1 ./silo-0.5.5/second/main.c

```
/*
   1996 Second Stage Linux/SPARC boot loader
   Mauricio Plaza Villegas
   Under GPL
   Based on Peter Zaitcev's floppy loader.
   */
   #include <asm/openprom.h>
   #include <linux/elf.h>
   #define ELFDATA2MSB   2
   /* We carry own a.out header for ease of cross-compilation.  */
   struct exec {
       int magic;
       int ltext;
       int ldata;
       int lbss;
       int lsym;
       int lentry;
       int x1;
       int x2;
   };
   #ifndef CONSOLE_H
   #include "console.h"
   #endif
   #ifndef CMDLINE_H
   #include "cmdline.h"
   #endif
   #include <storage.h>
   extern xprintf(int filog, char *fmt, ...);
   extern int load_kernel (char *S, int N);
   /** extern unsigned csum(const unsigned *p, unsigned l); **/
   typedef int (*Read0)(int dev, int num_blks, int blk_st,
           char* buf);
   typedef int (*Seek2)(int, int, int);
   typedef int (*Read2)(int, char *, int);
   typedef struct {
       int fd_;
       int seekp_;
       int floppy_;
       Read0 read0_;
       Seek2 seek2_;
       Read2 read2_;
   } Disk;
   extern int dkopen(Disk *, const struct linux_romvec *,
           Console *);
   extern int dkread(Disk *, char *, int nblk, int blk);
   extern void dkclose(Disk *, const struct linux_romvec *);
```

```
Console cons0;
Disk dk0;
int mult(int a, int b){
    int i, res=0;
    for (i=0; i<b; i++) res+=a;
    return res;
}
void bootparam(Cmdline *cmdp, Console *conp, Disk *dkp);
void sendparam(struct linux_romvec *promvec, Cmdline *cmdp,
        Console *conp);
int load_sunos(){
    char buffer[1024];
    struct Block_Table *sbt;
    int i;
    char *b=(char *) 0x4000;
    sbt = (struct Block_Table *) &buffer;
    if (dkread (&dk0, buffer, 2, OLD_LOADER_SEC)!=2) {
          xprintf ((int) &cons0, "Can not read old loader blocks....
                    aborting/n");
          return 0;
    }
    xprintf ((int) &cons0, "Magic = %s, Partition = %d,
            Offset = %d/n", sbt->magic, (int)sbt->partno,
            sbt->DiskOffset);
    xprintf ((int) &cons0, "Block Size = %d, NSect = %d/n",
            sbt->bs, (int)sbt->nsect);
    xprintf ((int) &cons0, "/n/nLoading old lodaer (probably
            SunOS)/n");
    for (i=0; sbt->blocks[i]!=0; i++){
          dkread (&dk0, b, sbt->nsect,
              sbt->blocks[i]*sbt->nsect+sbt->DiskOffset);
          b+=512*sbt->nsect;
          xprintf ((int) &cons0,".");
    }
    return 1;
}
/* Here we are launched */
int bootmain(struct linux_romvec *promvec){
    static Cmdline cmd0;
    unsigned off;
    int len;
    union {
          char *b;
          struct exec *a;
          Elf32_Ehdr *e;
    } hp;
    int isfile, fileok=0;
```

```
        dpset(&cons0, promvec);
        cmdcons(&cmd0);
        if (dkopen(&dk0, promvec, &cons0) == -1) {
              xprintf((int) &cons0, "Cannot open disk/n");
              return 0;
        }
        /* Ask for the kernel, and loadit into memory */
        isfile = 0;/* RC=0 invalid file or not an executable */
        while (!isfile){
              char kname[CMD_LENG];
              int partno,i;
              xprintf ((int) &cons0,"/nType [device]kernel_name
                        [parameters]/n");
              for (i=0; i<CMD_LENG; i++) /* Some Cleaning */
                  cmd0.cbuff_[i]=kname[i]=0;
              bootparam (&cmd0, &cons0, &dk0);
              if (strcmp (cmd0.cbuff_, "halt") == 0)
                  return 0;
              if (!select_parameters (cmd0.cbuff_, kname, &partno,
                        promvec))
                  continue;
              if (strcmp (kname, "sunos") == 0){
                  fileok = load_sunos();
              } else {
                  fileok = load_kernel(kname, partno);
              }
              if (fileok){
    /* At this point the first sector is loaded (and the rest of */
    /* the kernel) so we check if it is an executable file */
                    hp.b = (char *) 0x4000;
                    if (hp.a->magic == 0x01030107) {
                          off = sizeof(struct exec);
                          len = hp.a->ltext + hp.a->ldata;
                          xprintf((int) &cons0, "%d/0x%x bytes a.out/n",
                                len, len);
                          isfile = 1;
                    } else {
                          if (hp.e->e_ident[EI_MAG0] == ELFMAG0 &&
                              hp.e->e_ident[EI_MAG1] == ELFMAG1 &&
                              hp.e->e_ident[EI_MAG2] == ELFMAG2 &&
                              hp.e->e_ident[EI_MAG3] == ELFMAG3) {
                              if (hp.e->e_ident[EI_CLASS] != ELFCLASS32)
{
                                    xprintf((int) &cons0, "Not a 32bits
                                       ELF/n");
```

```
                    return 0;
                }
                if (hp.e->e_ident[EI_DATA] != ELFDATA2MSB)

                    xprintf((int) &cons0, "Not an MSB
                        ELF/n");
                    return 0;
                }
                {
                    Elf32_Phdr *p = (Elf32_Phdr *)
                        (hp.b + hp.e->e_phoff);
                    if (p->p_type != PT_LOAD) {
                        xprintf((int) &cons0, "Cannot
                          find a loadable segment/n");
                        return 0;
                    }
                    off = p->p_offset;
                    len = p->p_filesz;
                }
                xprintf((int) &cons0, "%d/0x%x bytes ELF/n",
                        len, len);
                isfile=1;
            } else {
                xprintf((int) &cons0, "Unknown image format/n");
                /* Do not abort.....ask for another file
                    return 0; */
            } /* If ELF  */

        } else {
            xprintf ((int) &cons0,"File NOT found....try again/n");
        } /* If file Ok */
    } /* While is file */
    /* Now we have a real binary file, let's move it to the */
    /* correct memaddress */
    xprintf((int) &cons0, "/n%d/0x%x bytes to move/n", len, len);
    {
            unsigned *f, *t;
            int cn;
            t = (unsigned *) 0x4000;
            f = t + off/sizeof(int);
            cn = len;
            while (cn > 0) {
                    *t++ = *f++;
                    *t++ = *f++;
                    *t++ = *f++;
                    *t++ = *f++;
```

```
                        cn -= 16;
                }
        }
        dkclose(&dk0, promvec);
        sendparam(promvec, &cmd0, &cons0);
        xprintf ((int) &cons0, "Penguin boot..../n");
        return 0x4000;
eread:
        xprintf((int) &cons0, "Read error/n");
        return 0;
}
/*
* Get command line from disk.
*/
static int readhint(Cmdline *cmdp)
{
        static char cmdblock[1024];
        struct Block_Table *sbt;
        sbt = (struct Block_Table *) cmdblock;
        if (dkread (&dk0, cmdblock, 2, OLD_LOADER_SEC)!=2){
                xprintf ((int) &cons0, "Error reading block table/n");
                return -1;
        }
        if (strcmp(&sbt->magic[1], SILO_MAGIC)){
                xprintf ((int) &cons0, "This is not a SILO block/n");
                return -1;
        }
        cmdfill(cmdp, sbt->params);
        return 0;
}
/*
* Obtain a command line from disk or console.
*/
void bootparam(Cmdline *cmdp, Console *conp, Disk *dkp)
{
        if (readhint(cmdp) != 1) {
/*
* Cases:  0 - edit with hint, -1 - edit without hint.
*/
                xprintf((int) conp, "silo:  ");
                cmdedit(cmdp, conp);
                xprintf((int) conp, "/n");
        }
        xprintf((int) conp, "silo:  /"%s/"/n", cmdp->cbuff_);
}
/*
* Supply a command line to the kernel.
```

```
*/
void sendparam(struct linux_romvec *promvec, Cmdline *cmdp,
        Console *conp)
{
        /* These are for return.  */
        static char arg0[CMD_LENG];
        static char *argv[2] = { arg0, 0 };
        /*
        * We expect a kernel to extract a command line
        * from our dead body promptly.  If somebody wants
        * to boot something other than Linux kernel than
        * he/she should just use a command line from PROM.
        */
        if (cmdp != 0 && cmdp->cbuff_[0] != 0){
                /* Be robust.  */
            /** strcpy(arg0, cmdp->cbuff_); **/
            {
                register char *f = cmdp->cbuff_, *t =
                        arg0;
                while(*t++ = *f++) {}
            }
            switch (promvec->pv_romvers) {
            case 0:
            case 1:
                {
                    char **p = (*(promvec->pv_v0bootargs))->argv;
                    p[0] = arg0;
                    p[1] = 0;
                }
                break;
            case 2:
            case 3:
                xprintf((int) conp, "silo:  %x %x/n",
                        argv, arg0);
                xprintf((int) conp, "silo:  V2 %x/n",
                        promvec->pv_v2bootargs.bootargs);
                *promvec->pv_v2bootargs.bootargs = arg0;
                break;
            default:
            }
        }
}
int dkopen(Disk *t, const struct linux_romvec *promvec,
        Console *con){
        static char name0[11];
        char *name;
        /* xprintf ((int) &cons0, "DEBUG: Disk Open/n");*/
```

```
        t->fd_ = 0;
        t->seekp_ = -1;
        t->floppy_ = 0;
        t->read0_ = 0;
        t->seek2_ = 0;
        t->read2_ = 0;
        if (promvec->pv_romvers == 0) {
                struct linux_arguments_v0 *ap =
                    *promvec->pv_v0bootargs;
                char *s = name0;
                int unit;
                *s++ = ap->boot_dev[0];
                *s++ = ap->boot_dev[1];
                *s++ = '(';
                *s++ = (ap->boot_dev_ctrl & 07) + '0';
                *s++ = ',';
                if ((*s = ap->boot_dev_unit/10 + '0') != '0')
                        s++;
                *s++ = ap->boot_dev_unit%10 + '0';
                *s++ = ',';
                *s++ = (ap->dev_partition & 07) + '0';
                *s++ = ')';
                *s = 0;
                xprintf ((int) &cons0, ":  name0 = %s/n", s);
                xprintf((int) con, "%s/n", name0);
                t->fd_ = (*promvec->pv_v0devops.v0_devopen)(name0);
                if (t->fd_ == 0 || t->fd_ == -1) return -1;
                t->read0_ = promvec->pv_v0devops.v0_rdblkdev;
                if (name0[0] == 'f' && name0[1] == 'd')
                        t->floppy_ = 1;
        } else {
                int rc;
                xprintf((int) con, "Bootpath is %s/n",
                        *promvec->pv_v2bootargs.bootpath);
                t->fd_ = (*promvec->pv_v2devops.v2_dev_open)
                        (*promvec->pv_v2bootargs.bootpath);
                if (t->fd_ == 0 || t->fd_ == -1) return -1;
                t->seek2_ = promvec->pv_v2devops.v2_dev_seek;
                t->read2_ = promvec->pv_v2devops.v2_dev_read;
                /** rc = (*t->seek2_)(t->fd_, 0, 18*512); **/
                /** xprintf((int) con, "%x/n", rc); **/
        }
        return 0;

dkread(Disk *t, char *buff, int size, int blk){
        if (t->read0_ != 0) {
                t->seekp_ = blk;
                return (*t->read0_)(t->fd_, size, blk, buff);
```

```
        }
        if (t->read2_ != 0) {
                int rc;
                if (t->seekp_ != blk) {
                        if ((*t->seek2_)(t->fd_,0,blk*512)==-1)
                                return -1;
                        t->seekp_ = blk;
                }
                rc = (*t->read2_)(t->fd_, buff, size*512);
                if (rc == size*512) return size;
        }
        return -1;
}
void dkclose(Disk *t, const struct linux_romvec *promvec){
        if (promvec->pv_romvers == 0) {
                /* Free drive for a root disk.  */
                (*promvec->pv_v0devops.v0_devclose)(t->fd_);
        } else {
                (*promvec->pv_v2devops.v2_dev_close)(t->fd_);
        }
}
```

A.2.2 ./silo-0.5.5/second/prom_io.c

```
/**/
  /*
  1996 IO_Manager (SPARC Linux Loader)
  Mauricio Plaza
  Under GPL
  Unix_IO Manager as PROM_IO Manager (for ext2fs)
  */
  #ifdef UTEST
  #include <stdio.h>
  #include <unistd.h>
  #include <stdlib.h>
  #include <time.h>
  #include <fcntl.h>
  #include <string.h>
  #endif
  #include <ctype.h>
  #include <sys/types.h>
  #include <errno.h>
  #ifndef UTEST
  typedef int FILE;
  #endif
  #include "prom_io.h"
  #include <storage.h>
  typedef int (*Read0)(int dev, int num_blks, int blk_st,
          char* buf);
```

```
typedef int (*Seek2)(int, int, int);
typedef int (*Read2)(int, char *, int);
typedef struct {
        int fd_;
        int seekp_;
        int floppy_;
        Read0 read0_;
        Seek2 seek2_;
        Read2 read2_;
} Disk;
extern void xprintf(int filog, const char *fmt, ...);
extern int  dkread(Disk *, char *, int nblk, int blk);
extern Disk dk0;
extern Console cons0;
ino_t                root,cwd;       /* Directories */
io_manager           linux_io_manager = &struct_linux_manager;
static unsigned char nsect;          /* Sectors per Block */
static unsigned int  bs;             /* Block Size */
static unsigned long int DiskOffset;
                        /* Block where partition starts */
char                     *buffer;    /* 0x4000 */
int select_parameters (char source[], char kname[], int *partno,
            struct linux_romvec *promvec){
    char *params;
    char buff[1024];
    struct Block_Table *stb;
    params = strchr(source, ' ');
    strncpy (kname, source, strlen(source) - strlen(params));
    if (params) {
          strcpy (source, params);
    } else {
          source[0]=0;
    }
    stb = (struct Block_Table *) &buff;
    dkread (&dk0, buff, 2, SECOND_STAGE_SEC);
    bs = stb->bs;
    if (params = strstr(kname,";")) {
          char odisk[80]={0};
          xprintf ((int) &cons0,"Loading kernel from other disk,
                performing io-changes/n");
          strncpy (odisk, kname, (int)(params - kname));
          params++;
          dkclose (&dk0, promvec);
          strcpy (*promvec->pv_v2bootargs.bootpath, odisk);
          if (dkopen (&dk0, promvec, &cons0)==-1) {
              xprintf (&cons0, "ERROR: Can not open %s/n", odisk);
              return 0;
```

```
            }
            *partno = (int) (*params - '0');
            params ++;
            if (*partno>0 && *partno<9 && *params=='/') {
                strcpy (kname, params);
            } else {
                xprintf ((int) &cons0, "ERROR: Invalid partition
                      (%d) or not a '/'./n", *partno);
                return 0;
            }
        } else {
            if ((int) stb->partno != 0) {
                *partno = (int) stb->partno;
                xprintf ((int) &cons0, "Default partition is %d./n",
                     *partno);
            }else{
                xprintf ((int) &cons0, "Invalid partition %d on
                          block %d", (int) stb->partno,
                          OLD_LOADER_SEC);
                xprintf ((int) &cons0, " defaulting to 4/n");
                *partno = 4;
            }
        }
        return 1;
    }
    int read_sun_part  (int partno)
    {
        int rc;
        struct sun_disklabel *DiskLabel;
        char sdlbuffer[512];  /* DiskLabel as seen on disk */
        DiskLabel = (struct sun_disklabel *) &sdlbuffer;
        rc = dkread (&dk0, sdlbuffer, 1, 0);
        if (rc != 1){
            xprintf ((int) &cons0, "ERROR: Can not read partition
                   ...aborting!/n");
            return 0;
        }
        DiskOffset = DiskLabel->ntrks * DiskLabel->nsect *
              DiskLabel->partitions[partno-1].start_cylinder;
        nsect = bs / 512;
        return 1;
    }
    #ifdef CHEAT_C_LIB
    char *strcpy (char *d, char *s)
    {
    }
```

```
char *memcpy (char *d, char *s)
{
}
int strlen (char *s)
{
}
int memset (char *buf, int value, int count)
{
}
void strncmp (char *a, char *b)
{
}
char *strchr (char *s, int c)
{
}
#endif
int sprintf (char *buf, char *fmt, ...)
{
    strcpy (buf, fmt);
}
errcode_t linux_open(const char *name, int flags, io_channel *channel)
{
    int   partno;
    io_channel io = 0;
    if (name == 0)
        return EXT2_ET_BAD_DEVICE_NAME;
    io = (io_channel) malloc(sizeof(struct struct_io_channel));
    if (!io)
        return EXT2_ET_BAD_DEVICE_NAME;
    memset(io, 0, sizeof(struct struct_io_channel));
    io->magic = EXT2_ET_MAGIC_IO_CHANNEL;
    io->manager = linux_io_manager;
    io->name = (char *) malloc (strlen (name)+1);
    strcpy (io->name, name);
    io->block_size = bs;
    io->read_error = 0;
    io->write_error = 0;
    partno = *(name + strlen(name) - 1) - '0';
    if (!read_sun_part (partno)){
        xprintf ((int) &cons0," ERROR: On linux_open, the SUN
            partition of %s (part= %d)is invalid./n", name,partno);
        return EXT2_ET_BAD_DEVICE_NAME;
    }
    *channel=io;
    return 0;
}
errcode_t linux_close(io_channel channel){
```

```
xprintf ((int) &cons0,"DEBUG: linux close no existe!/n");
}
errcode_t linux_set_blksize(io_channel channel, int blksize){
channel->block_size = bs = blksize;
}
errcode_t linux_read_blk(io_channel channel, unsigned long block,
                int count, void *data)
{
int size;
size = (count < 0) ?  -count :  count*bs;
nsect = bs / 512;
if (size<512){
    int i;
    char buff[512];
    xprintf ((int) &cons0, "DEBUG: Read size smaller than 512
            (%d)/n",size);
    dkread (&dk0, buff, 1, block*nsect+DiskOffset);
    memcpy (data, buff, size);
} else {
    dkread (&dk0, data, size/512, block*nsect+DiskOffset);
}
return 0;
}
errcode_t linux_write_blk(io_channel channel, unsigned long block,
                    int count, const void *data)
{
xprintf ((int) &cons0,"DEBUG: write_blk llamado!/n");
}
errcode_t linux_flush(io_channel channel)
{
xprintf ((int) &cons0,"DEBUG: flush llamado!/n");
}
ext2_filsys fs = 0;
int silo_open_ext2 (char *device)
{
    int retval;
    retval = ext2fs_open (device, EXT2_FLAG_RW, 0, 0,
            linux_io_manager, &fs);
    if (retval){
        xprintf ((int) &cons0,"ERROR: Unable to open the filesystem
                %d(0x%x)/n", retval, retval);
        return 0;
    }
    retval = ext2fs_read_inode_bitmap(fs);
    if (retval) {
        xprintf ((int) &cons0,"ERROR: Unable to read inode bitmap
```

```
                %d(0x%x)/n",retval, retval);
        return 0;
    }
    retval = ext2fs_read_block_bitmap(fs);
    if (retval) {
        xprintf ((int) &cons0, "ERROR: Unable to read block bitmap
                %d(0x%x)/n", retval, retval);
        return 0;
    }
    root = cwd = EXT2_ROOT_INO;
    return 1;
}
static int dump_block(ext2_filsys fs, blk_t *blocknr, int blockcnt,
                      void *private)
{
        size_t nbytes;
        int errcode;
        static int i=0;
        static char rot [] = "///|/-//";
        if (blockcnt < 0)
                return 0;
        if (*blocknr) {
                errcode = io_channel_read_blk(fs->io, *blocknr,
                                                1, buffer);
                buffer += bs;
                if (i % 5)
                  xprintf ((int) &cons0,"%c/b",rot[(i/5)%5]);
                i++; /*  = (i + 1) % sizeof (rot); */
                if (errcode)
                    return BLOCK_ABORT;
        }
        return 0;
}
int dump_file(ino_t inode, int fd, char *outname)
{
        errcode_t retval;
        retval = ext2fs_block_iterate(fs, inode, 0, 0,
                                      dump_block, 0);
        if (retval) {
                com_err("xx", retval, "while iterating over blocks
                        in %s",
                        outname);
                return 0;
        }
        return 1;
}
int load_file(char *argv)
```

```
{
        ino_t inode;
        int retval;
        char arr [1024];
        if (retval = ext2fs_namei(fs, root, cwd, argv, &inode)){
          xprintf ((int) &cons0,"ERROR: On load_file, Inode error
                 #%d./n", retval);
          return 0;
        }
        if (!inode)
              return 0;
        if (dump_file(inode, 0, "dumpmok"))
           return 1;    /* In fact on 0x4000 */
        return 0;
}
void com_err (const char *a, long i, const char *fmt,...)
{
    xprintf ((int) &cons0,fmt);
}
load_kernel(char *filename, int N)
{
    char device[]="/dev/sdaX";  /* CHEAT. */
    device[8]=(char)N + '0';
    xprintf ((int) &cons0,"Loading Kernel %s form device %s
             (%d)/n", filename, device, N);
    if (!silo_open_ext2(device)){
          xprintf ((int) &cons0," ERROR: On load_kernel.  Can
                not open partition %s /n", device);
          return 0;
    }
    xprintf ((int) &cons0,"Loading file %s./n", filename);
    buffer = (char *)0x4000;
    if (!load_file (filename)){
          xprintf ((int) &cons0,"ERROR: On load_kernel.
                Failed to load file/n");
          return 0;
    }
    return 1;
}
```

A.2.3 ./silo-0.5.5/second/prom_io.h

```
#include <linux/ext2_fs.h>
  #include "ext2fs/ext2fs.h"
  #include <asm/openprom.h>
  #ifndef CONSOLE_H
  #include "console.h"
```

```
#endif
#ifndef CMDLINE_H
#include "cmdline.h"
#endif
#ifdef __STDC__
#define NOARGS void
#else
#define NOARGS
#define const
#endif
struct sun_disklabel {
        unsigned char info[128]; /* Informative text string*/
        unsigned char spare[292];/* Boot information etc.*/
        unsigned short rspeed;   /* Disk rotational speed*/
        unsigned short pcylcount;/* Physical cylinder count*/
        unsigned short sparecyl; /* extra sects per cylinder*/
        unsigned char spare2[4]; /* More magic...*/
        unsigned short ilfact;   /* Interleave factor*/
        unsigned short ncyl;     /* Data cylinder count*/
        unsigned short nacyl;    /* Alt.  cylinder count*/
        unsigned short ntrks;    /* Tracks per cylinder*/
        unsigned short nsect;    /* Sectors per track*/
        unsigned char spare3[4]; /* Even more magic...*/
        struct sun_partition {
                unsigned long start_cylinder;
                unsigned long num_sectors;
        } partitions[8];
        unsigned short magic;    /* Magic number */
        unsigned short csum;     /* Label xor'd checksum*/
};
struct dump_block_struct {
        int fd;
        char *buf;
        errcode_t errcode;
};
static  errcode_t linux_open(const char *name, int flags,
        io_channel *channel);
static  errcode_t linux_close(io_channel channel);
static  errcode_t linux_set_blksize(io_channel channel,
        int blksize);
static  errcode_t linux_read_blk(io_channel channel,
        unsigned long block, int count, void *data);
static  errcode_t linux_write_blk(io_channel channel,
        unsigned long block, int count, const void *data);
static  errcode_t linux_flush(io_channel channel);
static struct struct_io_manager struct_linux_manager = {
        EXT2_ET_MAGIC_IO_MANAGER,
        "linux I/O Manager",
        linux_open,
```

```
        linux_close,
        linux_set_blksize,
        linux_read_blk,
        linux_write_blk,
        linux_flush
};
extern ext2_filsys fs;
extern ino_t    root, cwd;
extern ino_t    root, cwd;
```

A.2.4 ./silo-0.5.5/second/malloc.c

```
#define MALLOC_BASE 0x680000
   static char *malloc_ptr = (char *) MALLOC_BASE;
   static char *last_alloc = 0;
   void *malloc (int size)
   {
       char *caddr;
       caddr = malloc_ptr;
       malloc_ptr += size;
       last_alloc = caddr;
       malloc_ptr = (char *) ((((unsigned int)malloc_ptr)
                  + 3) & ( 3));
       return caddr;
   }
   void free (void *m)
   {
       if (m == last_alloc)
           malloc_ptr = last_alloc;
   }
```

A.3 Utilerías

A.3.1 ./silo-0.5.5/instboot/instboot.c

```
   /*
   1996 SPARC Linux Loader Installation Utility
   Mauricio Plaza
   Under GPL
   */
   /* This program must generate a lists of block where the second
   loader will be stored.  By doing this:
     1- Get the block in a partition
     2- Get the real block
     3- Write the information on blocks 14-15.  The BOOTBLOCK
     are from 1-15 and the silo fits on 11.  So we have enogh
     room for the block table.
     4- Old SunOS loader will be stored into a file and its
```

```
  blocks will be saved on block 13.  So we may be able to
  boot SunOS from time to time.
  */
#include <stdio.h>
#include <linux/fs.h>
#include <linux/ext2_fs.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <storage.h>
struct sun_disklabel {
    unsigned char info[128];   /* Informative text string */
    unsigned char spare[292];  /* Boot information etc.  */
    unsigned short rspeed;     /* Disk rotational speed */
    unsigned short pcylcount;  /* Physical cylinder count */
    unsigned short sparecyl;   /* extra sects per cylinder */
    unsigned char spare2[4];   /* More magic...  */
    unsigned short ilfact;     /* Interleave factor */
    unsigned short ncyl;       /* Data cylinder count */
    unsigned short nacyl;      /* Alt.  cylinder count */
    unsigned short ntrks;      /* Tracks per cylinder */
    unsigned short nsect;      /* Sectors per track */
    unsigned char spare3[4];   /* Even more magic...  */
    struct sun_partition {
          unsigned long start_cylinder;
          unsigned long num_sectors;
    } partitions[8];
    unsigned short magic;      /* Magic number */
    unsigned short csum;       /* Label xor'd checksum */
};
#ifdef SUN
__u32 swab32(__u32 value)
{
    return((value>>24) | ((value>>8)&0xff00) |
           ((value<<8)&0xff0000) | (value<<24));
}
__u16 swab16(__u16 value)
{
    return (value >> 8) | (value << 8);
}
# else
#   define swab32(x) (x)
#   define swab16(x) (x)
#endif
#define OLD_LOADER_NAME "/boot.loader"
struct ext2_super_block sb;      /* Super Block Info */
struct Block_Table *bt;          /* Block Table structure */
char   block_buff[1024]={0};     /* Block Table seen on disk */
int    force=0;                  /* Forcing installation */
```

```
char   *first, *second, *old;   /* File names */
#ifdef DEBUG
int fdb;
void do_cat(char *device){
    int fd,i,j;
    char buff[1024];
    if ((fd = open (device, O_RDONLY)) == -1){
         printf ("Cannot Open %s/n", device);
         exit (1);
    }
    printf (" Absoluto desde %s /n/n",device);
    for (i=0; bt->blocks[i]!=0; i++)
         for (j=0; j<bt->nsect; j++){
             lseek (fd, (bt->blocks[i]*bt->nsect+j+bt->DiskOffset)
                        *512, 0);
             read (fd, buff, sizeof(buff));
             printf ("%s",buff);
         }
}
#endif
void read_sb(char *device){
    int fd, partno;
    char buff[512];
    struct sun_disklabel *sdl;
    if ((fd = open (device, O_RDONLY)) == -1){
         printf ("Super Block:  Cannot find %s/n",device);
         exit (1);
    }
    lseek (fd, 1024, 0);
    if (read(fd, &sb, sizeof (sb)) != sizeof (sb)) {
         puts (" Can not read Super Block!");
         exit(1);
    }
    if (swab16(sb.s_magic)!=EXT2_SUPER_MAGIC){
         printf (" %s is not an Ext2 File System", device);
         exit (2);
    }
#ifdef DEBUG
    fdb = sb.s_first_data_block;
    puts (" Super Block Partition Information");
    printf (" Inodes Count = %d  Block Count = %d/n",
          sb.s_inodes_count, sb.s_blocks_count);
    printf (" First Data Block = %d /n", fdb);
#endif
    close (fd);
    partno = (int)(*(device+strlen(device)-1) - '0') - 1;
    *(device+strlen(device)-1)=0; /* Truncate X on /dev/sdaX*/
```

```
    sdl = (struct sun_disklabel *) &buff;
    if ((fd = open (device, O_RDONLY)) == -1){
        printf ("Error reading %s ...",device);
        exit (1);
    }
    read (fd, buff, sizeof(buff));
    printf ("Tracks = %d, Sectors = %d, Cylinders = %d/n",
            sdl->ntrks, sdl->nsect, sdl->ncyl);
    printf ("Starting Cylinder = %d/n", sdl->
            partitions[partno].start_cylinder);
    bt->DiskOffset = sdl->ntrks * sdl->nsect *
            sdl->partitions[partno].start_cylinder;
    bt->partno = partno + 1;
    printf ("DiskOffset = %d /n", bt->DiskOffset);
}
/* Return the block list on  bt->blocks */
int get_partition_blocks(char *filename) {
    int fd;
    int r;
    int block, i, j;
    if ((fd = open (filename, O_RDONLY)) == -1){
          printf ("Cannot find %s/n",filename);
          exit (1);
    }
    printf ("/n/nInformation about the %s file ..../n",
            filename);
    printf ("Partition Number  = %d/n",(int)bt->partno);
    printf ("Block Size        = %d/n",bt->bs);
    printf ("Sectors per Block = %d/n",(int)bt->nsect);
    printf ("File Blocks       = {");
    for (i=0; ; i++){
          bt->blocks[i]=0;
          block = i;
          r = ioctl (fd, FIBMAP, &block);
          if (r == -1 || block == 0){
              break;
          }
          bt->blocks[i]=block;
          printf ("%d,",block);
    }
    puts ("0}");
    close (fd);
    return 0;
}
void write_block_table(char *device, int boot_block_sector){
    int fd,rc;
```

```
    printf ("/n/nOpening raw device %s for writing .../n",
            device);
    if ((fd = open (device, O_RDWR)) == -1){
          printf ("Cannot open %s/n",device);
          exit (1);
    }
    bt->magic[0]='0';
    strcpy (&bt->magic[1], SILO_MAGIC);
    lseek (fd, boot_block_sector*512, 0);
    rc=write (fd, block_buff, sizeof(block_buff));
    printf ("/n %d Bytes written..../n",rc);
    close (fd);
    puts ("");
}
usage(char *s){
    printf ("Usage:  %s [-f] booting_device first.loader
        second.loader [oldloader]",s);
    puts ("/n    -f forces to install both, first stage and
        second stage.  By defaul SILO will install the firts stage
        loader only if the SILO signature is not found.");
    puts ("    booting_device is the device where SILO will
        be installed");
    puts ("    oldloader is the file name where the original
        boot loader will be saved into.  See OLD_FILE_LOADER
        in sources for default file.  Watch out for -f option,
        the original loader may be overwritten by a SILO-loader.");
}
void say_partition(char *filename, char part[], char *bootdev){
          puts (" Old Loader and second loader must be
              on the booting disk ");
          exit (1);
    }
}
int copy_old_loader(char *device, char *filename){
    char buffer[512*15];
    struct Block_Table *sbt;
    int fd, rc;
    FILE *fp;
    printf ("/n/nSearching for an old loader.../nOpening
            raw device %s for reading .../n", device);
    if ((fd = open (device, O_RDONLY)) == -1){
          printf ("Cannot open %s/n",device);
          exit (1);
    }
    lseek (fd, 512, 0);
    read (fd, buffer, sizeof(buffer));
```

```
        close (fd);
        sbt = (struct Block_Table *) &buffer[512*13];
        printf ("Tabla nsect = %d, bs = %d, magic = %s/n",
                sbt->nsect, sbt->bs, sbt->magic);
        if (!strcmp(&sbt->magic[1], SILO_MAGIC) ){
              puts ("SILO Found.....skiping backup");
              if (force) puts ("Force requested....");
              else return 0;
        }
        if ((fp=fopen(filename, "w"))==NULL){
              puts ("Can not save old loader....aborting!");
              exit (1);
        }
        rc = fwrite (buffer, 1, sizeof (buffer), fp);
        printf (" %d Bytes of oldloader saved on %s./n", rc,
                filename);
        fclose (fp);
        return rc;
}
void install_first_stage(char *device, char *filename){
        char buff[512];
        int i,rc;
        int fd;      /* Device */
        FILE *fp;    /* File */
        printf ("Opening device %s for writing..../n", device);
        if ((fd=open(device,O_WRONLY)) == -1){
              puts ("Opening device error....aborting!");
              exit (1);
        }
        printf ("Opening file %s for reading..../n", filename);
        if ((fp=fopen(filename,"r"))==NULL){
              puts ("Opening file error....aborting!");
              exit (1);
        }
        for (i=512; !feof(fp); i+=512){
              rc = fread (buff, 1, sizeof(buff), fp);
              lseek (fd, i, 0);
              printf ("Reading %d bytes from %s...  /n", rc, filename);
              rc = write (fd, buff,  rc);
              printf ("Writing %d bytes to %d on %s...  /n",
                  rc, i, device);
        }
        close (fd);
        fclose (fp);
}
set_booting_params(char *device){
```

```
        int fd;
        char buffer[1024];
        struct Block_Table *sbt;
        char aux[100];
        sbt=(struct Block_Table *) &buffer;
        printf ("Setting Boot Parameters./nOpening device %s
                for writing..../n",
                device);
        if ((fd=open(device,O_RDWR)) == -1){
              puts ("Opening device error....aborting!");
              exit (1);
        }
        lseek (fd, 512*OLD_LOADER_SEC, 0);
        read (fd, buffer, sizeof(buffer));
        printf ("/n/nType booting parameters (%s):",
                sbt->params);
        gets(aux);
        if (strcmp(aux, "")){
              printf ("Parameters are :  %s /n", aux);
              strcpy (sbt->params, aux);
              lseek (fd, 512*OLD_LOADER_SEC, 0);
              write (fd, buffer, sizeof(buffer));
        }
        close(fd);
}
main (int argc, char *argv[]){
    char bootdev[10], fpart[10], spart[10], lpart[10];
    bt  = (struct Block_Table *) &block_buff;
    if (argc<4) {
          usage (argv[0]);
          return -1;
    }
    if (argv[1][0]=='-'){
          if (argv[1][1]=='f') force = 1;
          else printf ("Invalid option %s...ignoring./n", argv[1]);
          strcpy (bootdev, argv[2]);
          first = strdup (argv[3]);
          second = strdup (argv[4]);
          if (argc==6) old = strdup (argv[5]);
          else old = strdup (OLD_LOADER_NAME);
    } else {
          strcpy (bootdev,argv[1]);
          first = strdup (argv[2]);
          second = strdup (argv[3]);
          if (argc==5) old = strdup (argv[4]);
          else old = strdup (OLD_LOADER_NAME);
```

```
    }
    printf ("/t Installing on device %s, getting first
        loader from %s /nand second loader from %s, saving
        old loader on %s.  %s/n", bootdev, first, second,
        old, force?"Forcing to install":"!");
    say_partition (old, fpart, bootdev);
    read_sb(fpart);
    bt->bs = 1024 << swab32(sb.s_log_block_size);
    bt->nsect = (bt->bs / 512);
    if (copy_old_loader (bootdev, old)){
          get_partition_blocks (old);
          write_block_table (bootdev, OLD_LOADER_SEC);
          install_first_stage(bootdev, first);
    }
    set_booting_params(bootdev);
    say_partition (second, spart, bootdev);
    read_sb(spart);
    bt->bs = 1024 << swab32(sb.s_log_block_size);
    bt->nsect = (bt->bs / 512);
    get_partition_blocks(second);
#ifdef DEBUG
    do_cat(argv[3]);
#endif
    write_block_table (bootdev, SECOND_STAGE_SEC);
    return 0;
}
```

A.3.2 ./silo-0.5.5/include/storage.h

```
/*
1996 Include file for bootblocks
Mauricio Plaza
Under GPL
*/
    /* The firts loader must read the parameters and the
        blocks from the bootbloks 14-15.  In order to load the
        secondary loader which understand ext2fs */
#define SECOND_STAGE_SEC 14
#define OLD_LOADER_SEC   13
#define SILO_MAGIC  "SILO"
/* WARNING: This structute must be no longer than 1024 bytes */
struct Block_Table{
  char               magic[10];   /* Signature */
  char               params[100]; /* Booting Parameters*/
  unsigned char      partno;      /* Default Partition No.  */
  unsigned char      nsect;       /* Sectors per Block */
  unsigned short int bs;          /* Block Size */
```

```
  unsigned long  int DiskOffset;  /* Real Cylinder */
  unsigned long  int blocks[200]; /* File Block Table */
};
```

A.3.3 ./silo-0.5.5/include/asm/openprom.h

```
#ifndef __SPARC_OPENPROM_H
#define __SPARC_OPENPROM_H
/* openprom.h:    Prom structures and defines for access to the
*             OPENBOOT prom routines and data areas.
*
* Copyright (C) 1995 David S. Miller (davem@caip.rutgers.edu)
*/
/* In the v0 interface of the openboot prom we could traverse
* a nice little list structure to figure out where in vm-space
* the prom had mapped itself and how much space it was taking
* up.  In the v2 prom interface we have to rely on 'magic' values.
* :-( Most of the machinesI have checked on have the prom mapped
* here all the time though.
*/
#define KADB_DEBUGGER_BEGVM     0xffc00000
                /* Where kern debugger is in virt-mem */
#define LINUX_OPPROM_BEGVM    0xffd00000
#define LINUX_OPPROM_ENDVM    0xfff00000
#define LINUX_OPPROM_MAGIC      0x10010407
#ifndef __ASSEMBLY__
/* The device functions structure for the v0 prom.  Nice and neat,
* open,close, read & write divvied up between net + block + char
* devices.  We also have a seek routine only usable for block
* devices.  The divide and conquer strategy of this struct becomes
* unnecessary for v2.
*
* V0 device names are limited to two characters, 'sd' for
* scsi-disk, 'le' for local-ethernet, etc.  Note that it is
* technically possible to boot a kernel off of a tape drive and
* use the tape as the root partition!  In order to do this you
* have to have 'magic' formatted tapes from Sun supposedly :-)
*/
struct linux_dev_v0_funcs {
        int (*v0_devopen)(char *device_str);
        int (*v0_devclose)(int dev_desc);
        int (*v0_rdblkdev)(int dev_desc, int num_blks,
                int blk_st, char*  buf);
        int (*v0_wrblkdev)(int dev_desc, int num_blks,
                int blk_st, char*  buf);
        int (*v0_wrnetdev)(int dev_desc, int num_bytes,
                char*  buf);
        int (*v0_rdnetdev)(int dev_desc, int num_bytes,
                char*  buf);
        int (*v0_rdchardev)(int dev_desc, int num_bytes,
```

```
        int dummy, char*  buf);
        int (*v0_wrchardev)(int dev_desc, int num_bytes,
                int dummy, char*  buf);
        int (*v0_seekdev)(int dev_desc, long logical_offst,
                int from);
};
/* The OpenBoot Prom device operations for version-2 interfaces
* are both good and bad.  They now allow you to address ANY device
* whatsoever that is in the machine via these funny "device paths".
* They look like this:
*
*   "/sbus/esp@0,0xf004002c/sd@3,1"
*
* You can basically reference any device on the machine this way,
* and you pass this string to the v2 dev_ops.  Producing these
* strings all the time can be a pain in the rear after a while.
* Why v2 has memory allocations in here are beyond me.  Perhaps
* they figure that if you are going to use only the prom's device
* drivers then your memory management is either non-existent or
* pretty sad.  :-)
*/
struct linux_dev_v2_funcs {
        int (*v2_inst2pkg)(int d);
        char* (*v2_dumb_mem_alloc)(char*  va, unsigned sz);
        void (*v2_dumb_mem_free)(char*  va, unsigned sz);
        char* (*v2_dumb_mmap)(char*  virta, int which_io,
                unsigned paddr, unsigned sz);
        void (*v2_dumb_munmap)(char*  virta, unsigned size);
        /* Basic Operations, self-explanatory */
        int (*v2_dev_open)(char *devpath);
        void (*v2_dev_close)(int d);
        int (*v2_dev_read)(int d, char*  buf, int nbytes);
        int (*v2_dev_write)(int d, char*  buf, int nbytes);
        void (*v2_dev_seek)(int d, int hi, int lo);
        /* Never issued (multistage load support) */
        void (*v2_wheee2)(void);
        void (*v2_wheee3)(void);
};
struct linux_mlist_v0 {
        struct linux_mlist_v0 *theres_more;
        char*  start_adr;
        unsigned num_bytes;
};
struct linux_mem_v0 {
        struct linux_mlist_v0 **v0_totphys; /* physical */
        struct linux_mlist_v0 **v0_prommap;
                    /* addresses map'd by prom */
```

```
        struct  linux_mlist_v0 **v0_available;
                    /* what phys.  is left over */
};
/* Arguments sent to the kernel from the boot prompt.  */
struct linux_arguments_v0 {
        char *argv[8];          /* argv format for boot string */
        char args[100];         /* string space */
        char boot_dev[2];  /* e.g., "sd" for `b sd(...'  */
        int boot_dev_ctrl; /* controller # */
        int boot_dev_unit; /* unit # */
        int dev_partition;  /* partition # */
        char *kernel_file_name; /* kernel to boot */
        void *aieee1; /* give me some time  :> */
};
/* Prom version-2 gives us the raw strings for boot arguments
* and boot device path.  We also get the stdin and stdout file
* pseudo descriptors for use with the mungy v2 device functions.
*/
struct linux_bootargs_v2 {
        char **bootpath; /* V2:  Path to boot device */
        char **bootargs; /* V2:  Boot args */
        int *fd_stdin; /* V2:  Stdin descriptor */
        int *fd_stdout; /* V2:  Stdout descriptor */
};
struct linux_romvec {
        /* Version numbers.  */
        unsigned int pv_magic_cookie;
        unsigned int pv_romvers;
        unsigned int pv_plugin_revision;
        unsigned int pv_printrev;
        struct linux_mem_v0 pv_v0mem;
        /* Node operations (see below).  */
        struct linux_nodeops *pv_nodeops;
        char **pv_bootstr;
                    /* Boot command, eg sd(0,0,0)vmunix */
        struct linux_dev_v0_funcs pv_v0devops;
        char *pv_stdin; /* stdin cookie */
        char *pv_stdout; /* stdout cookie */
#define PROMDEV_KBD 0 /* input from keyboard */
#define PROMDEV_SCREEN 0 /* output to screen */
#define PROMDEV_TTYA 1 /* in/out to ttya */
#define PROMDEV_TTYB 2 /* in/out to ttyb */
        /* Blocking getchar/putchar.    NOT REENTRANT! (grr) */
        int (*pv_getchar)(void);
        void (*pv_putchar)(int ch);
        /* Non-blocking variants that return -1 on error.  */
```

```
        int (*pv_nbgetchar)(void);
        int (*pv_nbputchar)(int ch);
        /* Put counted string (can be very slow).  */
        void (*pv_putstr)(char *str, int len);
        /* Miscellany.  */
        void (*pv_reboot)(char *bootstr);
        void (*pv_printf)(const char *fmt, ...);
        void (*pv_abort)(void);
        int *pv_ticks;
        void    (*pv_halt)(void);
        void (**pv_synchook)(void);
        union {
                void (*v0_eval)(int len, char *str);
                void (*v2_eval)(char *str);
        } pv_fortheval;
        struct linux_arguments_v0 **pv_v0bootargs;
        /* Extract Ethernet address from network device.  */
        unsigned int (*pv_enaddr)(int d, char *enaddr);
        struct linux_bootargs_v2 pv_v2bootargs;
        struct linux_dev_v2_funcs pv_v2devops;
        int filler[15];
        void (*pv_setctxt)(int ctxt, char*  va, int pmeg);
        int (*v3_cpustart)(unsigned int whichcpu, int ctxtbl_ptr,
                           int thiscontext, char* prog_counter);
        int (*v3_cpustop)(unsigned int whichcpu);
        int (*v3_cpuidle)(unsigned int whichcpu);
        int (*v3_cpuresume)(unsigned int whichcpu);
};
struct linux_nodeops {
        /*
        * Tree traversal.
        */
        int (*no_nextnode)(int node); /* next(node) */
        int (*no_child)(int node); /* first child */
        int (*no_proplen)(int node, char*  name);
        int (*no_getprop)(int node, char*  name,
                char*  val);
        int (*no_setprop)(int node, char*  name,
                char*  val, int len);
        char* (*no_nextprop)(int node, char*  name);
};
/* More fun PROM structures for device probing.  */
#define PROMREG_MAX     16
#define PROMVADDR_MAX   16
#define PROMINTR_MAX    15
struct linux_prom_registers {
int which_io;           /* is this in OBIO space?  */
```

```
    char *phys_addr;
    int reg_size;
    };
    struct linux_prom_irqs {
    int pri;    /* IRQ priority */
    int vector; /* This is foobar, what does it do?  */
    };
    /* Element of the "ranges" vector */
    struct linux_prom_ranges {
            unsigned int ot_child_space;
            unsigned int ot_child_base;
            unsigned int ot_parent_space;
            unsigned int ot_parent_base;
            unsigned int or_size;
    };
    #endif /* !(__ASSEMBLY__) */
    #endif /* !(__SPARC_OPENPROM_H) */
    #endif /* !(__SPARC_OPENPROM_H) */
```

A.3.4 ./silo-0.5.5/include/ext2fs/io.h

```
/*
   * io.h --- the I/O manager abstraction
   *
   * Copyright (C) 1993 Theodore Ts'o.    This file may be redistributed
   * under the terms of the GNU Public License.
   */
   /*
   * ext2_loff_t is defined here since unix_io.c needs it.
   */
   #if defined(__GNUC__) || defined(HAS_LONG_LONG)
   typedef long long ext2_loff_t;
   #else
   typedef long ext2_loff_t;
   #endif
   /* llseek.c */
   ext2_loff_t ext2_llseek (unsigned int, ext2_loff_t,
                unsigned int);
   typedef struct struct_io_manager *io_manager;
   typedef struct struct_io_channel *io_channel;
   struct struct_io_channel {
           int magic;
           io_manager manager;
           char *name;
           int block_size;
           errcode_t (*read_error)(io_channel channel,
                                   unsigned long block,
                                   int count,
                                   void *data,
```

```
                                        size_t size,
                                        int actual_bytes_read,
                                        errcode_t error);
        errcode_t (*write_error)(io_channel channel,
                                        unsigned long block,
                                        int count,
                                        const void *data,
                                        size_t size,
                                        int actual_bytes_written,
                                        errcode_t error);
        int reserved[16];
        void *private_data;
};
struct struct_io_manager {
        int magic;
        const char *name;
        errcode_t (*open)(const char *name, int flags,
                io_channel *channel);
        errcode_t (*close)(io_channel channel);
        errcode_t (*set_blksize)(io_channel channel,
                int blksize);
        errcode_t (*read_blk)(io_channel channel, unsigned
                long block, int count, void *data);
        errcode_t (*write_blk)(io_channel channel, unsigned
                long block, int count, const void *data);
        errcode_t (*flush)(io_channel channel);
        int reserved[16];
};
#define IO_FLAG_RW 1
/*
* Convenience functions....
*/
#define io_channel_close(c)
                ((c)->manager->close((c)))
#define io_channel_set_blksize(c,s)
                ((c)->manager->set_blksize((c),s))
#define io_channel_read_blk(c,b,n,d)
                ((c)->manager->read_blk((c),b,n,d))
#define io_channel_write_blk(c,b,n,d)
                ((c)->manager->write_blk((c),b,n,d))
#define io_channel_flush(c)
                ((c)->manager->flush((c)))
extern io_manager unix_io_manager;
```

A.3.5 ./silo-0.5.5/README

This is the first attempt of an complete boot loader for Linux on the Sparc. Because of the lack of space on the bootblocks. we have to do it in two steps, the first step is a very simple loader based on Peter Zaitcev's silo (we will call this the first stage loader) which should fit in less than 8Kb and it's sole purpose is to load a more complete bootstrap loader, herein refered as the second stage boot loader. The cool thing about the second stage loader I implemented is that it makes use of the ext2 library (provided with the ext2fs tools). and thus allows the loader to access any file on a ext2 file system. This is different from Linux/i386 lilo that needs a map for each kernel, in silo, we just keep one map file for the second stage loader, we don't expect you to be changing the second stage loader on your daily routine (you can do so, you will just need to use a tool to reinstall the maps).

During the installation, SILO saves the original boot loader under an user especified file name, so, when silo prompts for a kernel name it is possible to load the original boot loader by typing the reseved word "sunos". There is another magic word (halt) that gets you back to the prom.

If you wan to uninstall SILO, tun the simple script uninst, which restores the original bootloader.

1. Installation.
You need all the propper compilers and linkers for SunOS.

Under the bin subdir you will find all the programs already compiled and ready to run on Linux/SPARC, this won't work on SunOS or Solaris. Copy the files "first.image" and "second.image" to the same partition (it should be an ext2 filesystem), then run the command like this:

instboot /dev/sda4 /mnt/first.image /mnt/second.image /mnt/oldloader

Currently, the two images and the oldloader must be saved on the same partition (working on fixing that). This should save your original bootblocks (those from SunOS for example) on a file called "oldloader", this serves two purposes: in the future we will be able to boot SunOS and Solaris with silo and as a back up you can restore with the friendly dd tool. Finally the in-

stboor program will install the both loaders properly in your machine. Then cross your fingers and reboot. The firts loader pumps you into the second one very quickly then you will see a prompt that understands strings of the form:

[prompath;partition_number]kernelname [kernel-args]

[prompath] is the typical path ath the prom (/sbus/esp@....) If a different disk is given the I shloud have the new partition too. Example: /sbus/esp@0,800000/sd@1,0;4

[kernelname] is the kernel file name, this usually is something like "vmlinux", "penguinkernel" or "turkey-kernel", some people like to have their kernels on a directory, this is just fine, as far as the directory does not span multiple devices (ie, you can use as a pathname something like: "/kernels/mylinux").

[kernel-args] are the arguments that are going to be passed to the kernel

For example, this is the command line I use to boot platypus.nuclecu.unam.mx:

/sbus/esp@0,80000/sd@1,0;4/kernels/vmunix
nfsroot=132.248.29.9:/usr/nfs/root

2. Caveats

There are some features I'm working on (like a config file for password protecting the kernels (just like lilo) and adding some extra checks to the code).

3. License

All the code is under GPL.

4. TODO

1. Check the kernel size
2. NetBoot
3. /etc/silo.conf

5. Credits.

1. Mauricio Plaza
2. Peter Zaitcev
3. David Miller
4. Miguel de Icaza

A.3.6 ./silo-0.5.5/ChangeLog

```
Fri Mar 22 19:05:26 1996  Mauricio Plaza Villegas
   <mok@sphinx.nuclecu.unam.mx>
          * Ext2fs and libs are now on the distribution.
          * second/prom_io.c:  Allowing prom paths.
   Thu Mar 21 13:59:41 1996  Mauricio Plaza Villegas
   <mok@sphinx.nuclecu.unam.mx>
          * second/main.c:  Support for booting from other disks.
   Tue Mar 19 07:23:28 1996  David Miller
   <davem@caip.rutgers.edu>
          * The first stage loader can be compiled under SunOS.
   Fri Mar 15 18:03:38 1996  Mauricio Plaza Villegas
   <mok@nuclecu.unam.mx>
          * second/prom_io.c:  BUG FIXED: Block Size was hardcoded
          as 1024, now it should work for other block sizes.
   Thu Mar 14 19:35:51 1996  Mauricio Plaza Villegas
   <mok@sphinx.nuclecu.unam.mx>
          * bin/uninstall:  Uninstall Utility (a pretty simple one).
          * second/main.c:  CHAIN LOADING, Now we are able to load
          another Operating System by saving the old loader and
          re-running it instead of the kernel (tree steps).  There
          are olny two reserved words when second stage loader
          prompts, "halt" and "sunos".  Halt will interrupt the
          whole process backing you to the "ok" prompt.  Sunos will
          load the original loader of the original Operating System
          like SunOS.
          * instboot/instboot.c:  BUG FIXED: Default parameters
          are keeped.
          * second/prom_io.c (load_kernel):  BUG FIXED: Other
          partition loading now works.
   Wed Mar 13 09:22:32 1996  Mauricio Plaza Villegas
   <mok@sphinx.nuclecu.unam.mx>
          * second/main.c:  Adding support for default partition
          (same partition where the first stage is).
          (readhint):  Parameters from the blocktable.
          * instboot/instboot.c:  Adding the params support.  Read
          old parameters and give the chance to change them.
          * storage.h (struct Block_Table):  Adding a field called
          params so we can have default booting parameters.  Also
```

SILO_MAGIC has changed so we can boot without prompting,
if light runs out the machine will be able to reboot.

Apéndice B

Licencia GPL

Todo el software que se incluye con la distribución de `silo-0.5.5` está liberado con la licencia GPL (*General Public License*), que es la licencia con la que está liberado todo el código de Linux. Esta licencia, diseñada por la *Free Software Foundation*, protege al programador contra un posible plagio de su trabajo y además garantiza la libre distribución del software.

La licencia está en inglés, y no puede ser traducida pues la licencia se protege a si misma contra alguna modificación, ya sea en concepto o en lenguaje. Existe un proyecto para traducir oficialmente la licencia GPL y los manuales de referencia del software liberado con esta licencia, sin embargo, este proyecto aún no se ha terminado y no existe una traducción oficial de esta licencia en español. Por ejemplo, el término *free software* se puede traducir como software gratuito o software libre, que no es lo mismo.

Aunque no existe traducción oficial para la licencia, esto no impide dar una interpretación de la misma y en esencia esta licencia garantiza que:

1. El código está registrado a nombre del autor, pero no le pertence al autor, le pertence a todo el mundo.
2. No debe de haber restricción alguna para distribuir el software con el código fuente.
3. Si alguna persona modificara el código, no debe de haber restricción alguna para distribuir los cambios.

1. Si alguna persona modificara el código, se deberá añadir su nombre a la lista de contribuyentes al software.

A continuación se presenta la versión oficial completa de la licencia GPL en inglés.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is

addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License.

(Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to dis-

tribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and

conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such

case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

¡one line to give the program's name and a brief idea of what it does.¿
Copyright (C) 19yy ¡name of author¿

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave. Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

¡signature of Ty Coon¿, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Apéndice C

Benchmarks

Las siguientes pruebas comparativas de eficiencia (*Benchmarks*) fueron hechas para reafirmar el por qué Linux/SPARC es un proyecto importante y por qué la migración debía de ser realizada. Es un hecho que existen muchos sistemas operativos que funcionan en la plataforma SPARC, además de los creados por SUN Microsystems (SunOS y SOLARIS), como NextStep. sin embargo los sistemas desarrollados por SUN Microsystems habían sido los más eficientes en esta arquitectura.

Linux. además de ser gratuito, distribuirse con el código fuente del kernel, majenar consolas virtuales y correr binarios de SunOS y SOLARIS, resulta ser más eficiente que los sistemas operativos nativos de la SPARC.

Los algoritmos de estas pruebas comparativas de eficiencia (lmbench), fueron diseñadas por Larry McVoy de Silicon Graphics. donde hace un exhaustivo análisis entre distintos sistemas operativos en distintas plataformas [Usnx]. Ese trabajo fue presentado en Enero de 1996 en las conferencias técnicas mundiales de UNIX (USENIX) en San Diego California. En esas fechas la versión de Linux/SPARC aún no había sido liberada y sin embargo un procesador INTEL Pentuim a 100Mhz funcionaba casi con la misma eficiencia que un procesador RISC SPARC 20 a 173Mhz. Una vez liberada la versión Linux/SPARC, las pruebas de eficiencia han sido ejecutadas en el mismo procesador SPARC con los tres sistemas operativos mas eficientes, SunOS. SOLARIS y Linux.

L M B E N C H 1 . 0

Versión 1.99.X de Linux/SPARC

Procesamiento. Tiempos en micro segundos

Máquina	S. O.	Mhz	Sys Call	Proc Null	Proc Senc	/bin /sh	Mmap lat	2proc ctxsw	8proc ctxsw
trombetas	Lnx 1.99.5	50	11	8.7K	40.9K	75K	346	84	102
geneva.ru	SOL 2.5	50	31	33.7K	148.2K	274K	596	174	205
negro.rut	Sun 4.1.3_U	49	124	18.3K	63.9K	110K	470	152	262

Latencia de Comunicación en microsegundos

Máquina	S. O.	Pipe	UDP	RPC/ UDP	TCP	RPC/ TCP
trombetas	Linux 1.99.5	270	966	1686	1326	2500
geneva.ru	SOLARIS 2.5	530	1563	2080	1354	2398
negro.rut	SunOS 4.1.3_U	890	1375	2287	1573	2804

Ancho de banda en comunicaciones Mb/s

Máquina	S.O.	Pipe	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
trombetas	Lnx 1.99.5	8	4.4	25.0	17.4	18	25	41	36
geneva.ru	SOLARIS 2.5	8	7.0	12.6	19.5	18	18	40	36
negro.rut	Sun 4.1.3_U	4	2.0	19.5	8.2	18	24	41	36

Latencia de la memoria en nanosegundos

Máquina	OS	Mhz	L1 $	L2 $	Main mem	TLB	Guesses
trombetas	Linux 1.99.5	50	20	170	180	600	No L2 cache
geneva.ru	SOLARIS 2.5	49	-	-	-	-	Bad mhz
negro.rut	SunOS 4.1.3_U	49	20	175	183	659	No L2 cache

Versión 2.0 de Linux/SPARC

Procesamiento. Tiempos en micro segundos

Máquina	S. O.	Mhz	Sys Call	Proc Null	Proc Senc	/bin /sh	Mmap lat.	2proc ctxsw	8proc ctxsw
ariane	Lnx 2.0	50	6	3K	42K	107K	84	17	33
ariane	Sun 4.1.4	50	24	6K	42K	57K	183	30	68
ariane	SOLARIS 2.5	50	9	12K	75K	117K	141	46	77

Latencia de Comunicación en microsegundos

Máquina	S. O.	Pipe	UDP	RPC/ UDP	TCP	RPC/ TCP
ariane	Linux 2.0	70	332	670	452	1006
ariane	Sun 4.1.4	174	381	668	427	923
ariane	SOLARIS 5.4	211	519	799	423	910

Ancho de banda en comunicaciones Mb/s

Máquina	S.O.	Pipe	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
ariane	Linux 2.0	29	10	17	31	19	21	43	32
ariane	SunOS 4.1.4	16	5	31	21	18	22	45	32
ariane	SOLARIS 5.4	13	12	26	24	19	19	45	32

Latencia de la memoria en nanosegundos

Máquina	OS	Mhz	L1 $	L2 $	Main mem
ariane	Linux 2.0.0	50	20	250	1030
ariane	SunOS 4.1.4	49	20	145	1053
ariane	SOLARIS 2.4	49	20	141	1043

Conclusiones

El desarrollo de software gratuito es de suma importancia para el avance y la divulgación del cómputo, evitando que programas y sistemas sean reservados sólo para aquellos que pueden pagar costosas licencia. Además es indispensable para la investigación y para el desarrollo de nuevas tecnologías. Y más aún, es importante la creación de programas bajo la licencia GPL, ya que esta licencia no sólo esta diseñada para prevenir que otras personas se adueñen del trabajo del programador, sino que además garantiza que no debe de haber restricción al distribuir el programa con su código fuente, y si este código sufriese alguna modificación, estos cambios deben de ser regresados al código original con su respectivo crédito al autor. Esto en escencia se traduce en hacer del computo un modo de vida y no un modo de ganarse la vida.

El colaborar con este tipo de proyectos a nivel internaccional, enriquece la experiencia como programador en la construcción de sistemas de software de alta calidad y portabilidad. Si bien SILO es tan sólo un ladrillo en la inmensa muralla que forma el sistema operativo Linux, los conocimientos adqueridos para hacer posible su elaboración abren muchas puertas en esta dirección. Las metas que se fijaron al planear SILO fueron cubiertas satisfactoriamente, las cuales son:

1. Entiende el sistema de archivos ext2.
2. Tiene funcionalidad completa con distintos discos y/o particiones.
3. Es pequeño.
4. Es modular.
5. Tiene la capacidad para cargar otros sistemas operativos.

SILO es un trabajo aún en desarrollo (WIP), y aún hay muchas cosas por hacer, entre las más importantes están:

1. Construir un archivo de configuración `/etc/silo.conf`. Similar a `/etc/lilo.conf`.

2. Agregar la capacidad de cargarse desde unidad flexible. Esto es, tener una etapa 0 que quepa en un sector de 512 bytes que llame a la primera etapa de SILO descrita en este trabajo.

3. Entender el sistema de archivos para CDROM. Esto facilita las distribuciones de instalación del sistema operativo.

4. Entender el sistema de archivos *UFS*, nativo de SUN.

5. Constuir un menú que permita seleccionar de una manera más sencilla el kernel con el que se va a arrancar.

La versión que se presenta en este trabajo es `silo-0.5.5`, que fue liberado a finales del mes de Abril de 1996. Actualmente, un checoeslovaco llamado Jakub Jelinek, ha liberado la versión 0.6 de SILO que ya contiene los puntos 1 y 2 que le faltan a `silo-0.5.5` (el archivo de configuración `silo.conf` y el cargado en tres etapas). Esta es la ventaja de ser un trabajo gratuito y liberado con la licencia GPL, cualquier persona puede hecerle modificaciones al código que benefician única y exclusivamente al desarrollo del sistema y a una multitud de usuarios en todo el mundo. Dado que no existe un capital detrás de éste y todos los programas liberados con la licencia GPL, el único y verdadero interés es el desarrollo óptimo del sistema.

Bibliografía

[Stal] *William Stallings*,
Operating Systems,
1992 Maxwell MacMillan International Editions,
ISBN_0-00-946491-9.

[LnxJ] *David A. Rusling*,
Linux Journal: Kernel Korner,
"Linux on Alpha AXP: MILO the miniloader",
Enero 1996, número 21.

[Unix] *Brian Kernighan y Rob Pike*,
El entrono de programación UNIX,
1987. Prentice Hall, ISBN_968-880-067-8.

[Usnx] *Larry McVoy, Silicon Graphics, Inc.*
USENIX 1996 Annual Technical Conference,
"Imbench: Portable tools performance analysis",
Enereo 1996, USENIX, ISBN_1-880446-76-6.

[Sprc] *SUN Microsystems, Inc.*,
Manual de referencia del procesador SPARC,

[ExtD] *Louis-Dominique Dubeau*,
Especifications of an ext2 file system,
1994, Linux Documentation Project.

[Torv] **Newsgroup: comp.os.minix**,
Subject: What would you like to see most in minix?,
From: torvalds@klaava.Helsinki.FI*(Linus Benedict Torvalds)*,
Date: 25 Aug 91,
Summary: small poll for my new operating system,
http://babar.mit.edu/LPP/misc/Linux-History-2.html.

Agradecimientos

Son demasiadas las personas que de alguna u otra manera han influido y participado en mi formación, no sólo acedémica sino tambien personal. No podría hacer justicia mencionando a todas esas personas, son demasiadas y probablemente olvidare a algunas. Por supuesto que la gran parte de lo que soy se lo debo a mi padres, María Clemecia Villegas y Ramón Plaza Mancera. Este trabajo va dedicado a ellos, quienes siempre han estado a mi lado apoyandome, ayudandome y guiandome, además de haberme transmitido y enseñado el orgullo de ser Universitario y ese amor por nuestra máxima casa de estudios. Este trabajo representa la coronación de toda una vida de esfuerzo, dedicación y paciencia, mucha paciencia, pero no por parte mía, sino parte de ellos, mis padres. Las bases las fundaron ellos y ahora es mi misión continuarlas y engrandecerlas.

Le debo una agradecimiento muy especial a mi hermano, quien siempre ha estado a mi lado compartiendo los buenos y los malos momentos, compañero de camino y sobre todo el mejor amigo que he tenido.

Agradezco a todos mis profesores, en especial a la Maestra Elisa Viso, quien me ha apoyado, cuidado y orientado a lo largo de mi carrera, pero sobre todo por ser una gran amiga quien siempre ha estado pendiente de mis proyectos y problemas. Al Dr. Mario Magidin, al Dr. Jorge Ize y al Dr. Angel Carrillo a quienes debo gran parte mi formación como profesionista.

La lista de amigos es interminable, Miguel de Icaza, quien ha compartido conmigo sus proyectos, su amistad y una visión muy especial del cómputo. A Sara Luz Patiño, que más que una amiga, mucho más, ha sido mi cómplice y mi confidente. A Beatriz Vargas, una mujer muy especial que me ha enseñado una manera muy distinta y maravillosa de ver el mundo. A Ramses Perez,

Agradecimientos

Roberto Jimeno, Pablo Rosell, y a toda la banda de la Facultad de Ciencias, por su amistad y por los grandes momentos que pasamos juntos.

Agradezcole a todas estas personas su franca amistad y el aguantar mis necedades. Esta tesis va dedicada a todas estas personas.