CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
SECCIÓN DE COMPUTACIÓN

Núcleo Educativo Basado en Linux

Tesis que presenta

Domínguez Hernández Rogelio

Para obtener el grado de

Maestro en Ciencias

En la especialidad de

Ingeniería Eléctrica

Opción Computación

Director de la Tesis: **Dr. Jorge Buenabad Chávez**

México, D.F., Enero de 2006

CINVESTAV-IPN

Esta tesis fue realizada con ayuda de KOMA-Script, LaTeX y GNU/Linux

Resumen

En la actualidad existen varios sistemas operativos que pueden ser usados en cursos prácticos de Sistemas Operativos. Sin embargo, creemos que Linux es el sistema operativo ideal para este propósito debido a su robustez, confiabilidad, estabilidad, disponibilidad del código fuente y su amplio uso en los ámbitos académico e industrial. En particular, su amplio uso en distintos ámbitos, implica que nuevas ideas implementadas en Linux serán evaluadas y, posiblemente, usadas y mejoradas por mucha gente. Desafortunadamente, su soporte para distintas arquitecturas de procesador y una gran cantidad de dispositivos han hecho que Linux creciera tanto en tamaño como en complejidad. Por lo tanto, un curso de Sistemas Operativos basado en Linux podría ser muy demandante para los estudiantes.

En esta tesis presentamos eLinux, un núcleo basado en Linux que ha sido reducido tanto en código como en complejidad de tal manera que es conveniente de usar en un curso completo y práctico de Sistemas Operativos. Dicho curso permitiría a los estudiantes determinar si el área de Sistemas Operativos es uno de sus intereses de investigación. Para los estudiantes que así lo decidieran, eLinux proveería los conocimientos y habilidades básicas para que inicien sus desarrollos.

Abstract

Nowadays there are several operating systems that can be used to teach a practical course on Operating Systems. However, we think Linux is the most suitable for this purpose because of its robustness, wide-spread use in different contexts, and its support for various and varied hardware platforms. Its wide-spread use, particularly, means that new ideas developed in Linux will be evaluated and possibly used and enhanced by many people. Unfortunately, its support to many hardware platforms has meant a considerable increase in size and complexity. Hence a course on Operating Systems based on Linux is now a rather demanding task on students.

This thesis presents a Linux kernel reduced both in code and complexity that is suitable to teach a practical and comprehensive course on Operating Systems. Such course should allow students to realise whether operating systems is one of their research interests. And for those that is, it would have provided the base knowledge and skills to move further.

Agradecimientos

Quiero agradecer a mis padres, Clara y Raúl, por todo el apoyo que me han brindado durante toda mi vida. Quiero agradecerles no sólo por ser mis mejores amigos y mis mejores maestros sino, además, por concederme el honor de ser su hijo.

A mis hermanos Raúl, Roberto y Ricardo por todo lo que me han enseñado y por todos los invaluables momentos que hemos compartido juntos. También a mis cuñadas Miriam Iliana, Miriam y Yaneth por hacer de mis hermanos mejores personas.

A Karen, quien más que mi sobrina la considero como mi hermanita, por ser la alegría de toda la familia.

Al Dr. Jorge Buenabad por su amistad y asesoría en el desarrollo de la presente tesis.

A todos mis amigos que, con su ejemplo, su apoyo, sus consejos y su compañía, siempre son un aliciente para seguir adelante. Evitaré hacer una lista incómoda, ustedes saben quiénes son.

A todos los profesores que he tenido, pues considero que el hecho de compartir conocimientos es uno de los más grandes actos de humildad que un ser humano puede realizar.

A todos ustedes les prometo seguir siempre adelante.

Al CONACyT por la beca otorgada para realizar mis estudios de maestría.

Al CINVESTAV por la beca otorgada para finalizar la presente tesis.

Índice general

1. Introducción **1**

1.1. Terminología 5
1.2. Organización General de Linux 5
1.3. Panorama General de la Simplificación de Linux 13

2. Administrador de Procesos **17**

2.1. Administración de Procesos en Linux 17
2.1.1. Diagrama de Estados de Procesos 17
2.1.2. El Algoritmo de Planificación 19
2.1.3. El Descriptor de Proceso 24
2.1.4. Inicialización 28
2.2. Simplificación de la Administración de Procesos en eLinux 31
2.3. Tareas y Proyectos 32

3. Administrador de Memoria **35**

3.1. Organización de la Administración de Memoria en Linux 36
3.1.1. Organización de la Memoria Física 36
3.1.2. Tablas de Páginas 36
3.1.3. Inicialización 37
3.1.4. Boot Memory Allocator 37
3.1.5. Tablas Maestras de Páginas del Núcleo 40
3.1.6. Binary Buddy Allocator 41
3.1.7. Slab Allocator 43
3.1.8. Espacio de Direcciones de Proceso(Process Address Space) . . . 46
3.1.9. Administración del Espacio de Direcciones de Proceso en Linux 46
3.2. Simplificación de la Administración de Memoria en eLinux 49
3.3. Tareas y Proyectos 51

4. Administración de Entrada/Salida **53**

4.1. Administración de la Entrada/Salida en Linux 53
4.1.1. Representación de dispositivos 53

4.1.2. Tipos de dispositivos 54
4.1.3. Capa de Entrada/Salida para Dispositivos de Bloque 55
4.1.4. Inicialización 58
4.2. Simplificación de la Administración de Entrada/Salida de Linux 59
4.3. Tareas y Proyectos 62

5. Administración de Archivos 67
5.1. Sistemas de Archivos en Linux 67
5.1.1. Los sistemas de archivos tipo Unix 68
5.1.2. Sistemas de Archivos Soportados por Linux 69
5.2. La capa VFS 72
5.2.1. Inicialización 74
5.3. Simplificación de la Administración de Archivos de Linux 76
5.4. El Sistema de Archivos Xinu sobre eLinux 78
5.4.1. Estructura 78
5.4.2. Implementación 81
5.4.3. Registro 81
5.4.4. Manejo de archivos (inodes) 86
5.4.5. Lectura y Escritura de Directorios 91
5.4.6. Lectura y Escritura de Archivos 94
5.5. Tareas y Proyectos 99

6. Conclusiones 101
6.1. Verificación del Núcleo 102

A. Entorno de Desarrollo del Núcleo de eLinux 107
A.1. Entorno de Desarrollo Normal 107
A.1.1. Compilando el código de eLinux 107
A.1.2. Configurando el Cargador de Arranque 110
A.1.3. Desventajas 111
A.2. Entorno de Desarrollo con QEMU 112
A.2.1. Instalación y Configuración de QEMU 112
A.2.2. Instalación de Linux dentro de QEMU 113
A.2.3. Ejecutando QEMU 114
A.2.4. Ejecutando eLinux en QEMU 114
A.2.5. Copiando archivos al disco duro virtual de QEMU 115

B. Estructuras del Administrador de Procesos **117**
B.1. prio_array 117
B.2. runqueue 117
B.3. task_struct 119

C. Estructuras del Administrador de Memoria **125**
C.1. vm_area_struct 125
C.2. mm_struct 126

D. Estructuras de la Capa de Entrada/Salida para Dispositivos de Bloque **129**
D.1. buffer_head 129
D.2. Estructura bio 131

E. Estructuras de la capa VFS **133**
E.1. file_system_type 133
E.2. super_block 134
E.3. super_operations 136
E.4. inode 137
E.5. inode_operations 141
E.6. dentry 142
E.7. dentry_operations 144
E.8. file 144
E.9. file_operations 146
E.10. namespace 148
E.11. vfsmount 149

Bibliografía **151**

Índice alfabético **155**

Índice de cuadros

1.1. Arquitecturas soportadas por Linux 2.6.10. 7
1.2. Tipos de puertos y buses soportados por Linux 2.6.10. 7
1.3. Tipos de dispositivos soportados por Linux 2.6.10. 8
1.4. Algunas estadísticas de Linux 2.6.10. 9

2.1. Estadísticas sobre la administración de procesos en Linux 2.6.10 32
2.2. Estadísticas sobre la administración de procesos en eLinux 33

3.1. Estadísticas sobre la administración de memoria en Linux 2.6.10 49
3.2. Estadísticas sobre la administración de memoria en eLinux 51

4.1. Estadísticas sobre los controladores de dispositivos en Linux 2.6.10 . . . 62
4.2. Dispositivos soportados por eLinux 64
4.3. Estadísticas sobre los controladores de dispositivos en eLinux 65

5.1. Estadísticas sobre los sistemas de archivos en Linux 2.6.10 76
5.2. Estadísticas sobre los sistemas de archivos en eLinux 78

6.1. Algunas estadísticas de Linux 2.6.10. 101
6.2. Algunas estadísticas de eLinux 2.6.10. 102

Índice de figuras

1.1. Estructura general de Linux. 6
1.2. Dispositivos de hardware de una computadora. 9
1.3. Estructura general de eLinux. 15

2.1. Diagrama de estados en Linux 18
2.2. Los arreglos de prioridad (prio_array_t) 20
2.3. Secuencia de Inicialización del Administrador de Procesos 29
2.4. Organización de pid_hash 30
2.5. La interrupción de reloj 33

3.1. Relaciones entre las estructuras principales en Linux 37
3.2. Manejo de las tablas de páginas en eLinux 38
3.3. Secuencia de inicialización del administrador de memoria 39
3.4. Mapa de memoria en eLinux 40
3.5. Binary Buddy Allocator 42
3.6. Relación entre las estructuras del Slab Allocator. 44
3.7. Ejemplo de una página utilizada por el Slab Allocator. 45
3.8. Representación del espacio de direcciones de proceso 48
3.9. Relaciones entre las estructuras principales en eLinux 50
3.10. Mapa de memoria en eLinux 50

4.1. Organización de la estructura buffer_head. 57
4.2. Relaciones entre las estructuras bio, bio_vec y page. 59
4.3. Secuencia de inicialización en la administración de entrada/salida . . . 60
4.4. Dispositivos de hardware de una computadora. 61
4.5. Dispositivos de hardware soportados en eLinux. 63

5.1. Sistema de archivos tipo Unix 68
5.2. La capa VFS. 73
5.3. Secuencia de Inicialización del Administración de Archivos 75
5.4. Sistema de Archivos Xinu 79
5.5. Sistema de Archivos Xinu 79

5.6. Sistema de Archivos Xinu 80

A.1. Instalando Debian en QEMU 113

Capítulo 1.

Introducción

Los Sistemas Operativos son un componente esencial de las computadoras. Su propósito "es proveer un entorno para que el usuario pueda ejecutar programas. Por lo tanto, su propósito principal es hacer que una computadora sea conveniente de usar. Una meta secundaria es usar el hardware de una manera eficiente." [34] Dada su importancia, actualmente todo plan de estudios de una carrera de Ciencias Computacionales incluye al menos un curso de Sistemas Operativos [30]. El objetivo de dicho curso es que los estudiantes aprendan sobre el diseño e implementación de cada una de las partes que conforman un sistema operativo. Este curso es de importancia crítica para que los estudiantes comprendan los principios y el funcionamiento de los sistemas de cómputo modernos [39].

Existen distintas formas de impartir un curso de Sistemas Operativos. Comúnmente, los cursos introductorios (en nivel de licenciatura) son teóricos. Están orientados hacia el estudio de los conceptos y los distintos problemas que surgen en el diseño de sistemas operativos, así como las soluciones que se han propuesto, pero sólo se estudian de manera teórica. Los cursos prácticos, impartidos comúnmente en niveles más avanzados, requieren de dos componentes esenciales: máquinas para experimentación y el código fuente de un sistema operativo. Es indispensable contar con el código fuente del sistema operativo para poder analizarlo y modificarlo, y así obtener el mayor aprovechamiento del curso.

Antes de la década de los 80's, algunas instituciones educativas ofrecían cursos prácticos de sistemas operativos. Muchas de estas instituciones contaban con el código fuente de UNIX [33], el cual fue desarrollado en ensamblador por Dennis M. Ritchie y Ken Thompson en los laboratorios Bell de AT&T en 1969; y posteriormente reescrito en lenguaje C, también desarrollado en los laboratorios Bell por Brian W. Kernighan y Dennis M. Ritchie [32, 19]. La licencia para obtener el código fuente de UNIX podía ser adquirida de los laboratorios Bell a un costo muy bajo. En 1977, John Lions publicó sus notas tituladas "Source Code and Commentary on Unix level 6", la cual era la única documentación detallada disponible sobre el funcionamiento interno de UNIX. Estas

notas no estaban disponibles comercialmente, sólo las instituciones que adquirían la licencia de UNIX podían obtener una copia de ellas. A pesar de esto, estas notas fueron muy populares y se distribuyeron muchas copias ilegalmente. Recientemente, estas notas fueron reeditadas y publicadas como un clásico [23]. Con estas notas, durante la década de los 70's, UNIX fue muy utilizado dentro de las instituciones educativas. Pero en esa época el problema de dichas instituciones era la escasez de recursos de hardware, pues carecían de máquinas que pudieran ofrecer a los estudiantes para su experimentación con ellas.

Con la llegada de las computadoras personales en los inicios de la década de los 80's, comenzó a ser posible el ofrecer más y mejores cursos prácticos de Sistemas Operativos. Pero surgió otro problema: el código fuente de UNIX dejó de estar disponible. Algunas instituciones educativas y empresas adquirieron licencias del código fuente y realizaron sus propias variaciones a UNIX. Pero AT&T prohibía su distribución sin cubrir el costo de una licencia, el cual era muy alto. Y UNIX dejó de ser utilizado por instituciones educativas de bajos recursos.

A principios de los 80's ya no había sistemas operativos cuyo código fuente estuviese disponible para su estudio. Desde entonces, varios investigadores han desarrollado sistemas operativos con propósito educativo para ser utilizados en cursos tanto básicos como avanzados. Estos sistemas operativos están escritos en lenguaje C, primordialmente.

En 1984, Douglas Comer desarrolló XINU [10], un sistema operativo pequeño, elegante y con una estructura monolítica bien definida. Así mismo, Comer hizo disponible a XINU y publicó un libro [10] que explicaba detalladamente el funcionamiento de XINU, y los principios básicos de diseño e implementación de sistemas operativos.

Andrew Tanenbaum, en 1987, desarrolló el sistema operativo Minix [37], cuya estructura se basa en el paradigma cliente-servidor. Hasta el momento, MINIX ha evolucionado hasta la versión de MINIX 2.0, la cual puede ejecutarse en procesadores 8088, 286, 386, 486, y Pentium, en modos de 16 ó 32 bits. Igualmente, Tanenbaum publicó un libro sobre los aspectos teóricos de sistemas operativos y sobre su diseño e implementación en base a MINIX.

XINIX fue desarrollado en 1989, por Jorge Buenabad [9], con el propósito de tener una versión de XINU en computadoras PC-XT, adicionando características de MINIX, para ofrecer un sistemas operativo más completo.

Los anteriores sistemas operativos han sido aceptados por el campo académico de distintas maneras, siendo Minix el más utilizado en los cursos. El problema con dichos sistemas operativos es que no han traspasado las fronteras del ámbito académico, y que su desarrollo se ha detenido desde hace más de 10 años. Más recientemente, han surgido otros sistemas operativos con propósitos educativos, como NachOS [8], OS/161 [14], PortOS [4] y GeekOS [15].

NachOS provee una máquina virtual simulando un procesador MIPS R2000/3000.

Es decir, se ejecuta un programa (existen versiones para Ultrix, SunOS, HP-UX ,BSD UNIX y Linux) el cual emula un procesador MIPS y los programas de usuario son interpretados instrucción por instrucción.

OS/161 también corre sobre una máquina virtual que puede emular procesadores MIPS r2000 y r3000. Está escrito en C y su diseño está basado en NetBSD. Los creadores de OS/161 utilizaron NachOS durante un tiempo, pero consideraron que su diseño no era bueno; escondía demasiados detalles sobre el manejo del hardware. Así que decidieron crear su propio sistema operativo.

Igualmente, PortOS se ejecuta sobre una máquina virtual y sobre la familia de sistemas operativos Windows, en sus versiones de escritorio y móviles. PortOS emula un procesador basado en las arquitecturas x86 y StrongARM.

El motivo de que NachOS, OS/161 y PortOS corran sobre máquinas virtuales es ocultar a los estudiantes los detalles técnicos requeridos para manejar el hardware directamente. De esta manera, los estudiantes se concentran en el diseño de los sistemas operativos y no en los detalles de la arquitectura del hardware.

GeekOS es muy reciente y se encuentra en sus primeras versiones. Es un núcleo muy pequeño para procesadores x86 que ofrece las funcionalidades básicas (manejo de interrupciones, planificador de procesos, administrador de memoria básico y manejadores de dispositivos para el controlador de interrupciones, teclado, monitor VGA, discos duros IDE y floppy). Está orientado para ser utilizado en un primer curso de sistemas operativos. GeekOS ofrece funcionalidades básicas ideales para el desarrollo de núcleos desde etapas muy básicas, y también es ideal para un primer curso de introducción a los Sistemas Operativos.

Todos los sistemas operativos antes mencionados son buenas opciones para impartir un curso práctico de sistemas operativos. Los estudiantes aprenden a analizar, modificar y comprender un sistema operativo. Sin embargo, ¿Qué sucede si un estudiante, después de haber tomado un curso basado en uno de estos sistemas operativos, desea realizar investigación en el área de Sistemas Operativos? Por ejemplo, su tesis de maestría o de doctorado podría ser en el área de sistemas de archivos distribuidos, sistemas de tiempo real, migración de procesos, etc.

Idealmente, el estudiante implementaría sus nuevas ideas en un sistema operativo ampliamente utilizado, de tal manera que su desarrollo sea probado y utilizado fácilmente por otras personas interesadas en sus ideas. No utilizaría uno de los sistemas operativos mencionados anteriormente, aún cuando hubiese sido utilizado en el curso de sistemas operativos tomado por el estudiante, ya que, en la actualidad, dichos sistemas operativos no son utilizados fuera de la academia y, por lo tanto, el impacto sería mínimo.

Idealmente, el estudiante implementaría sus ideas en el sistema operativo Linux. Linux es un sistema operativo de alta calidad, con características que lo han llevado

a ser uno de los sistemas operativos más actualizados en cuanto al estado del arte y, por lo tanto, es ampliamente utilizado en los ámbitos académico e industrial. Su desarrollo se realiza por muchas personas alrededor del mundo bajo la coordinación de Linus Torvalds, quien lo introdujo en 1991. Linux está escrito en lenguaje C y basado en UNIX [33]. El código fuente está disponible al público en general, liberado bajo la licencia GPL (General Public License) [13], por lo que puede ser modificado y distribuido libremente.

Por lo anterior, Linux es una excelente opción para realizar investigación en el área de Sistemas Operativos y, para reducir la curva de aprendizaje, Linux debería ser la base de cursos prácticos de Sistemas Operativos. De hecho, el uso de Linux dentro de un curso de Sistemas Operativos ya ha sido propuesto por Bovet y Cesati [7], quienes ofrecen una organización del curso muy bien detallada, presentando todas las dependencias entre los temas del curso. Recientemente, Nieh y Vaill publicaron un artículo [29] donde comentan sus experiencias ofreciendo un curso de sistemas operativos basado en Linux. En este artículo mencionan el uso de un emulador (VMWare) para facilitar el desarrollo con el núcleo de Linux. También mencionan que el curso ha tenido gran éxito en su universidad (Columbia University) porque los estudiantes esperan obtener mucho provecho de él. Finalmente, el artículo describe brevemente los proyectos que se realizan durante el curso. Sin embargo, estos cursos están basados en el núcleo oficial de Linux, el cual, actualmente soporta 19 arquitecturas de hardware diferentes, así como una cantidad muy grande de dispositivos ([18, 31]). Un curso basado en el mismo no es viable de ser aprovechado por los estudiantes, algunos de los cuales podrían estar tomando un curso de sistemas operativos por primera vez. Es demasiado complejo.

Esta tesis presenta el diseño e implementación de Linux Educativo, o eLinux, una versión simplificada del núcleo oficial de Linux versión 2.6.10. En eLinux, se han eliminado características innecesarias para un curso de sistemas operativos impartido en un tiempo razonable (4 a 6 meses), tales como el soporte de distintas arquitecturas de hardware y de una gran cantidad de dispositivos. eLinux mantiene la estructura básica de Linux, pero solo soporta la arquitectura i386 y los dispositivos básicos para que los estudiantes ganen un entendimiento del funcionamiento global de un sistema operativo.

Estamos convencidos que los estudiantes, al finalizar un curso basado en eLinux, tendrán los conocimientos suficientes de su funcionamiento y sólo requerirán analizar los distintos subsistemas modificados o eliminados del núcleo oficial de Linux para implementar sus ideas sobre el mismo.

El beneficio obtenido por los estudiantes con un curso basado en eLinux es muy grande, pues el tiempo de desarrollo de sus proyectos se verá reducido de manera sustancial y tendrán un campo de aplicación mucho mayor y, por lo tanto, un impacto más grande en el área. Creemos que eLinux fomentará el interés en el área.

1.1. Terminología

En la actualidad existe confusión sobre el término Sistema Operativo. Para muchos usuarios un sistema operativo es el núcleo (kernel) más un conjunto de aplicaciones que ofrecen los servicios básicos del sistema (estas aplicaciones pueden ser desde el shell hasta la interfaz gráfica). Pero estrictamente, el sistema operativo sólo es el núcleo. La definición de sistema operativo de Douglas Comer es la siguiente:

> "Un sistema operativo es el software responsable del control directo y administración del hardware y las operaciones básicas del sistema. Adicionalmente, provee la base sobre la cual se pueden ejecutar aplicaciones".

Apegándonos a esta definición, en esta tesis nos referiremos indistintamente al término sistema operativo, ya sea como sistema operativo o como núcleo.

Otra confusión surge al referirnos a Linux. Richard Stallman lo aclara [36] de la siguiente manera:

> "Muchos usuarios no son plenamente conscientes de la diferencia entre el núcleo, que es Linux, y el sistema completo, al que también llaman «Linux»... Al día de hoy usamos sistemas GNU basados en Linux para la mayor parte de nuestro trabajo, y esperamos que usted también lo haga. Pero por favor, no confunda a la gente usando el nombre «Linux» de manera ambigua. Linux es el núcleo, uno de los principales componentes del sistema. El sistema en su conjunto es más o menos el sistema GNU, con Linux añadido. Cuando hable de esta combinación, por favor, llámela «GNU/Linux»".

Nosotros nos referiremos, correctamente, solamente al núcleo como Linux. El sistema GNU es irrelevante para la presente tesis (pero indispensable para el desarrollo del núcleo, pues se utilizaron herramientas del sistema GNU como el compilador `gcc` y el editor `emacs`, entre muchas otras).

También, para diferenciar entre Linux y el núcleo simplificado desarrollado en esta tesis, usaremos el término eLinux para referirnos a nuestro núcleo.

1.2. Organización General de Linux

La estructura general de Linux consiste de los siguientes subsistemas principales:

- Administrador de Procesos.
- Administrador de Memoria.

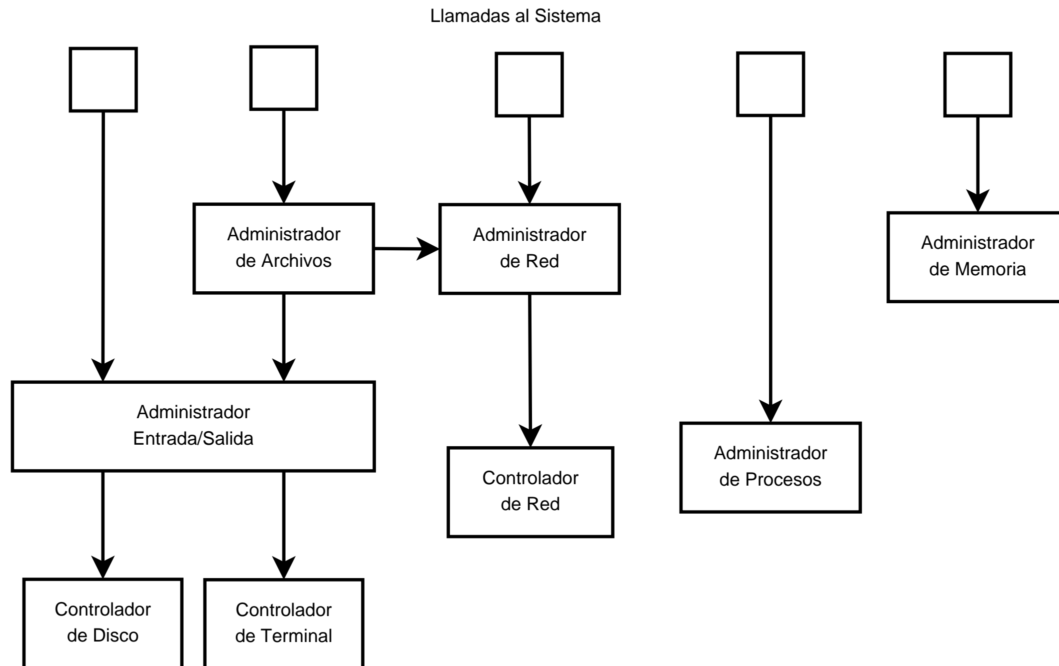

Figura 1.1.: Estructura general de Linux.

- Administrador de Red.
- Administrador de Entrada/Salida.
- Administrador de Archivos.

La relación entre estos subsistemas puede verse, de una manera muy simplificada, en la Figura 1.1. A pesar de que en dicha figura los subsistemas aparecen como si fueran independientes, en realidad todos están estrechamente ligados.

La estructura mostrada en la Figura 1.1 se ha mantenido constante desde la primeras versiones de Linux. Pero, internamente, los subsistemas han crecido mucho. Linux comenzó enfocado en la arquitectura i386. Posteriormente se realizaron esfuerzos para poder ejecutar Linux sobre otras arquitecturas. En la actualidad, Linux soporta 19 arquitecturas distintas de hardware que van desde sistemas embebidos hasta computadoras de alto rendimiento, tanto de 32 como de 64 bits. En el Cuadro 1.1 se enlistan las arquitecturas soportadas por Linux hasta la versión utilizada en esta tesis (2.6.10). Y esta lista sigue creciendo. Actualmente se está agregando soporte para las arquitecturas Cell (basado en PowerPC) y Xtensa.

i386	x86_64
ARM	ARM26
CRIS	H8300
IA-64	M68k
MIPS (32/64, LE/BE)	PA-RISC (32/64)
PowerPC	PowerPC 64
S/390 (32/64)	SuperH
SPARC	UltraSPARC
VAX	v850
Alpha	

Cuadro 1.1.: Arquitecturas soportadas por Linux 2.6.10.

IDE	SCSI
ATA	Serial ATA
RAID	USB
PS/2	Puerto Serial
Puerto Paralelo	AGP
PCI	PCI Express

Cuadro 1.2.: Tipos de puertos y buses soportados por Linux 2.6.10.

Además de soportar distintos procesadores, Linux también soporta arquitecturas de cómputo diferentes, tales como Symmetric Multi-Processing (SMP) y Simultaneous Multi-Threading (SMT), en las cuales se tienen más de una unidad de proceso. Para cada arquitectura de procesador que soporta SMP y/o SMT, Linux soporta SMP y/o SMT.

Linux también soporta arquitectutas NUMA (Non-Uniform Memory Access), la cual es una arquitectura de memoria compartida distribuida donde el tiempo de acceso a un dato depende de su localización en la memoria. Con NUMA, un procesador puede acceder a su propia memoria local más rápido que a otra memoria remota. Igualmente que con SMP y SMT, Linux soporta NUMA en todas las arquitecturas de procesador que lo soportan.

Linux también soporta una cantidad muy grande de dispositivos de todo tipo. El Cuadro 1.2 lista algunos de los distintos tipos de puertos y buses para dispositivos soportados por Linux. El Cuadro 1.3 enlista algunos de los tipos de dispositivos soportados por Linux.

Discos duros	Discos flexibles
Unidades ópticas (CD, DVD, etc.)	Unidades removibles USB
Tarjetas aceleradoras de video	Tarjetas de sonido
Tarjetas de red	Dispositivos de captura de video
Teclado	Ratones

Cuadro 1.3.: Tipos de dispositivos soportados por Linux 2.6.10.

Bovet y Cesati [7] identifican los dispositivos de hardware de un sistema de cómputo y los organizan en capas según su importancia, como se muestra en la Figura 1.2.

Obviamente, el dispositivo más importante es el CPU. En la capa alrededor del CPU se encuentran los dispositivos cuya existencia es esencial para el funcionamiento de un sistema de cómputo moderno y que deben ser administrados por el sistema operativo. Estos dispositivos son:

PIC, Programmable Interrupt Controller El controlador de interrupciones programable, encargado de administrar las interrupciones generadas por los dispositivos.

PIT, Programmable Interrupt Timer El temporizador de interrupción programable, encargado de enviar una interrupción periódica.

RTC, Real Time Clock El reloj de tiempo real, encargado de mantener la hora y fecha del sistema.

RAM (Random Access Memory) y Cache La memoria principal del sistema.

En la última capa encontramos a los dispositivos que no son requeridos por el sistema de cómputo para su correcto funcionamiento y su utilidad varía dependiendo de la aplicación.

Todos los controladores (drivers) de los dispositivos soportados por Linux se distribuyen junto con el código fuente de Linux. Linux se distribuye como un archivo `tar.bz2`, el cual es un archivo comprimido que contiene el código fuente para todas las arquitecturas y dispositivos soportados por Linux. El núcleo más reciente puede descargarse libremente de la página `www.kernel.org`. La cantidad de código en el archivo `linux-2.6.10.tar.bz2` es muy grande. Algunas estadísticas pueden verse en el Cuadro 1.4.

Para compilar el código de Linux y generar un núcleo, se deben configurar una gran cantidad de opciones. Las opciones determinan la arquitectura para la cual se compilará el código, algunas opciones específicas de la arquitectura, si se compilará el soporte

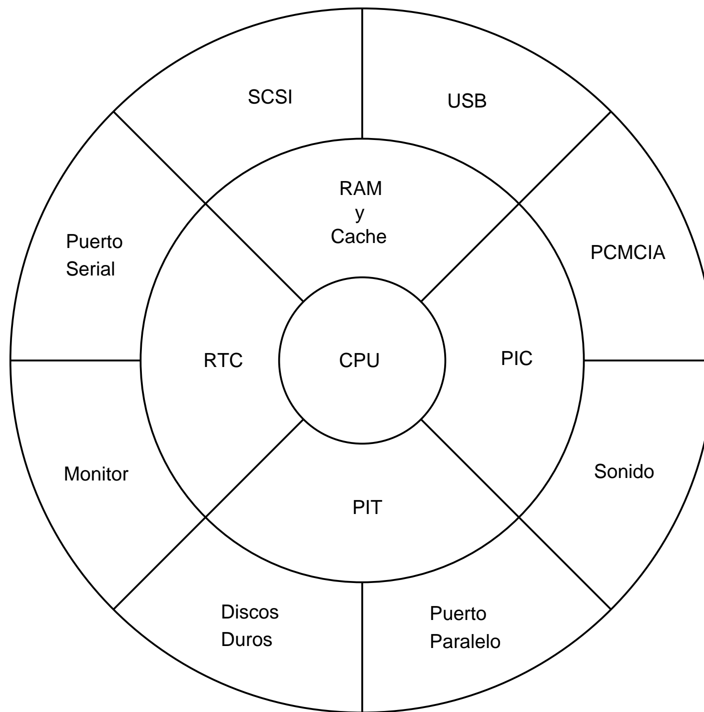

Figura 1.2.: Dispositivos de hardware de una computadora.

Tamaño descomprimido	227 MB
Archivos .c	6,906
Archivos .S (ensamblador)	6,773
Archivos .h	721
Total de archivos de código	14,400
Líneas de código en archivos .c	4,589,722
Líneas de código en archivos .S (ensamblador)	1,119,080
Líneas de código en archivos .h	256,488
Total de líneas de código	5,965,290
Cantidad de opciones de configuración	**5063**

Cuadro 1.4.: Algunas estadísticas de Linux 2.6.10.

para SMP, SMT y NUMA (si están disponibles para la arquitectura seleccionada), los tipos de dispositivos soportados y los controladores de dispositivos a compilar. El último renglón del Cuadro 1.4 se refiere a estas opciones de configuración.

Para administrar estas opciones de configuración, Linux utiliza un sistema de configuración y compilación llamado *kbuild*, el cual fue desarrollado específicamente para Linux. `kbuild` consiste de un conjunto de scripts y programas que se encargan de configurar el núcleo con ayuda del usuario. `kbuild` permite la configuración de Linux a través de un menú que muestra las opciones disponibles de la arquitectura para la cual se está compilando el núcleo.

Una vez realizada la configuración del núcleo, `kbuild` permite compilar el núcleo con las opciones de configuración seleccionadas. El trabajo de `kbuild` en esta etapa es definir las opciones seleccionadas como macros (es decir, declarar un conjunto de `#define`'s), las cuales permiten la compilación condicional del núcleo. El otro trabajo de `kbuild` en esta etapa es decidir que archivos son compilados y cuáles no.

Un ejemplo de código condicional puede verse en el siguiente fragmento de código, en el cual una función es definida de diferente manera dependiendo si está definida la macro `CONFIG_SMP`.

```
/*
 * resched_task - mark a task 'to be rescheduled now'.
 *
 * On UP this means the setting of the need_resched flag,
 * on SMP it might also involve a cross-CPU call to trigger
 * the scheduler on the target CPU.
 */
#ifdef CONFIG_SMP
static void resched_task(task_t *p)
{
	int need_resched, nrpolling;

	BUG_ON(!spin_is_locked(&task_rq(p)->lock));

	nrpolling = test_tsk_thread_flag(p, TIF_POLLING_NRFLAG);
	need_resched = test_and_set_tsk_thread_flag(p,
						TIF_NEED_RESCHED);
	nrpolling |= test_tsk_thread_flag(p,TIF_POLLING_NRFLAG);

	if (!need_resched && !nrpolling && (task_cpu(p) !=
						smp_processor_id()))
		smp_send_reschedule(task_cpu(p));
}
```

```
#else
static inline void resched_task(task_t *p)
{
    set_tsk_need_resched(p);
}
#endif
```

Otro ejemplo del uso de compilación condicional es agregar más campos a una estructura como en el siguiente fragmento de código.

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 */
struct runqueue {
        spinlock_t lock;

    /*
     * nr_running and cpu_load should be in the same cacheline
     * because remote CPUs use both these fields when doing
     * load calculation.
     */
    unsigned long nr_running;
#ifdef CONFIG_SMP
    unsigned long cpu_load;
#endif
    unsigned long long nr_switches;

    ...
}
```

Otro uso de compilación condicional se muestra en el siguiente fragmento de código, en el cual una función requiere realizar más operaciones de acuerdo con el valor de la macro `CONFIG_SMP`.

```
void register_irq_proc(unsigned int irq)
{
    char name [MAX_NAMELEN];

    if (!root_irq_dir ||
        (irq_desc[irq].handler == &no_irq_type) ||
        irq_dir[irq])
        return;
```

```
    memset(name, 0, MAX_NAMELEN);
    sprintf(name, "%d", irq);

    /* create /proc/irq/1234 */
    irq_dir[irq] = proc_mkdir(name, root_irq_dir);

#ifdef CONFIG_SMP
    {
        struct proc_dir_entry *entry;

        /* create /proc/irq/<irq>/smp_affinity */
        entry = create_proc_entry("smp_affinity",
                                  0600, irq_dir[irq]);

        if (entry) {
            entry->nlink = 1;
            entry->data = (void *)(long)irq;
            entry->read_proc = irq_affinity_read_proc;
            entry->write_proc = irq_affinity_write_proc;
        }
        smp_affinity_entry[irq] = entry;
    }
#endif
}
```

También existe código condicional dependiente de la arquitectura para la cual se está compilando el núcleo. El siguiente fragmento de código muestra un ejemplo.

```
/**
 * panic - halt the system
 * @fmt: The text string to print
 *
 * Display a message, then perform cleanups.
 * Functions in the panic notifier list are called after the
 * filesystem cache is flushed (when possible).
 *
 * This function never returns.
 */

NORET_TYPE void panic(const char * fmt, ...)
{
```

```
    long i;
    static char buf[1024];
    va_list args;
#if defined(CONFIG_ARCH_S390)
    unsigned long caller = (unsigned long)
                        __builtin_return_address(0);
#endif
    va_start(args, fmt);
    vsnprintf(buf, sizeof(buf), fmt, args);
    va_end(args);
    printk(KERN_EMERG "Kernel panic - not syncing: %s\n",buf);

    ...

#if defined(CONFIG_ARCH_S390)
    disabled_wait(caller);
#endif
    local_irq_enable();
    for (i = 0;;) {
        i += panic_blink(i);
        mdelay(1);
        i++;
    }
}
```

Así, podemos clasificar los distintos usos de la compilación condicional dentro del código de Linux de la siguiente manera:

- Definición de funciones.
- Definición de estructuras.
- Modificación de funciones.
- Introducción de código dependiente de la arquitectura en funciones genéricas.

1.3. Panorama General de la Simplificación de Linux

La gran cantidad de características de Linux, mencionadas en la sección 1.2, pueden dar una idea de la complejidad de su código.

La compilación condicional es sencilla de comprender pero dificulta el seguimiento del código para el caso más sencillo. Este tipo de situaciones dificulta a un estudiante

el comprender el sistema operativo, pues debe estar consciente de las características avanzadas de Linux y tomarlas en cuenta al momento de estudiar el código e, incluso, al momento de escribir nuevo código.

Mucho del código de Linux se debe a la gran cantidad de arquitecturas que soporta. $1,424,198$ (23,87 % del total) líneas de Linux son de código dependiente de la arquitectura.

El primer paso para reducir el código de Linux fue eliminar el soporte para todas las arquitecturas excepto para la arquitectura `i386`. La arquitectura `i386` es la más común y a la que los estudiantes están más familiarizados.

Igualmente, se eliminó el soporte de características avanzadas de Linux como SMP y NUMA.

También se eliminó el soporte de una gran cantidad de dispositivos de hardware, dejando sólo los indispensables (discos duros IDE, discos flexibles, teclado y monitor).

El eliminar dispositivos de red también permite eliminar el subsistema encargado de la administración de red, además de que lo que se busca es obtener un núcleo mínimo con las funcionalidades básicas de un sistema operativo.

De esta manera, el núcleo quedó reducido a los 4 subsistemas básicos mostrados en la Figura 1.3.

En los siguientes capítulos se analizan las características de Linux en cada uno de los subsistemas mostrados en la Figura 1.3 y se explican los cambios realizados a cada uno de ellos.

El Apéndice A describe cómo utilizar eLinux.

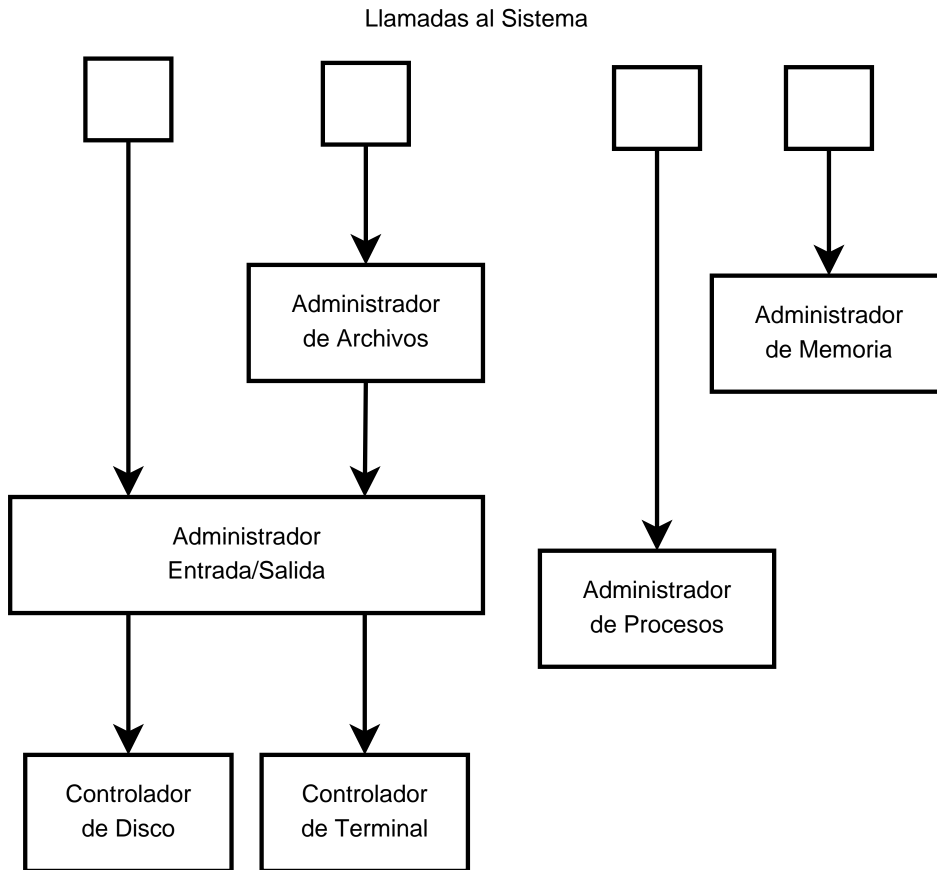

Figura 1.3.: Estructura general de eLinux.

Capítulo 2.

Administrador de Procesos

Un proceso es un programa en ejecución. Esto implica que un proceso es el conjunto de instrucciones contenidas en un programa y el conjunto de datos que el programa utiliza. Es decir, un proceso es el conjunto de las secciones de código, de datos, de memoria dinámica (heap), su pila, sus tablas de páginas y el contenido de los registros del procesador. Toda esta información es administrada por el sistema operativo para ofrecer un sistema multitarea.

La administración de procesos es un componente esencial de todo sistema operativo. El administrador de procesos debe mantener control de todos los procesos creados por el usuario, distribuir los recursos del sistema entre ellos, permitir la comunicación entre los procesos, y planificarlos.

En este capítulo, analizamos la administración de procesos en Linux, y cómo se simplificó para eLinux (en la Sección 2.2).

2.1. Administración de Procesos en Linux

2.1.1. Diagrama de Estados de Procesos

Durante su existencia, un proceso puede estar en diferentes estados. La Figura 2.1 muestra los estados utilizados en Linux y los eventos que generan las transiciones entre ellos. Los distintos estados en que un proceso puede estar son los siguientes:

`TASK_RUNNING`. Cuando un proceso es creado, este es su estado inmediato. Un proceso se encuentra en éste estado cuando está listo para ejecución o cuando se está ejecutando. Por razones de eficiencia, Linux no hace distinción, de manera explícita, entre los procesos listos y los procesos en ejecución. Un proceso listo es aquél que está en estado `TASK_RUNNING` y no está utilizando el procesador. Esto se hace para evitar cambiar el estado del proceso en cada ocasión que a un proceso listo se le asigna el procesador.

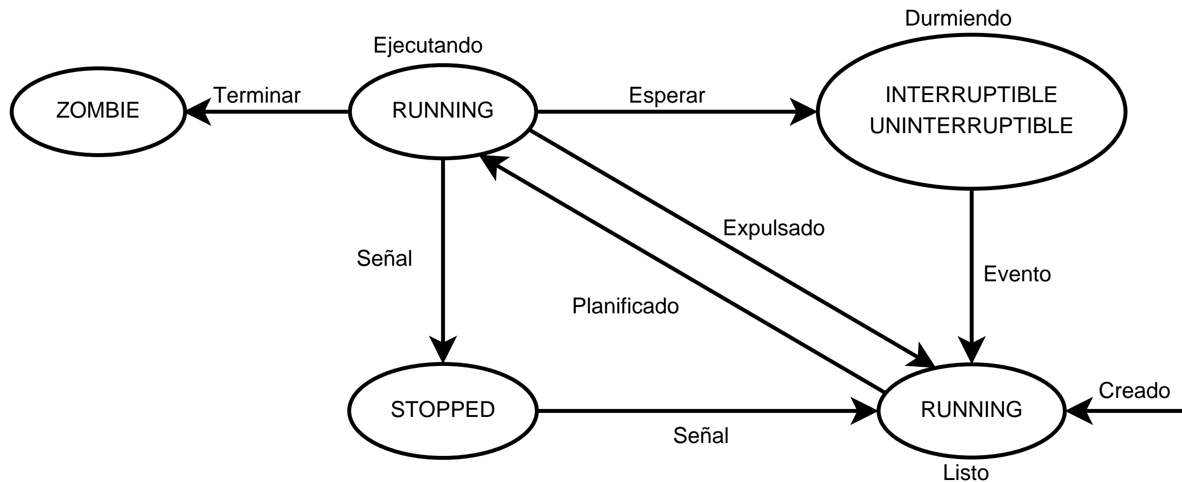

Figura 2.1.: Diagrama de estados en Linux

`TASK_INTERRUPTIBLE` y `TASK_UNINTERRUPTIBLE`. Cuando un proceso requiere esperar a que suceda un evento, cambia su estado a alguno de estos estados. La diferencia es que en `TASK_INTERRUPTIBLE` el proceso despierta ya sea cuando se cumple el evento por el cual estaba esperando o cuando recibe una señal. En `TASK_UNINTERRUPTIBLE` el proceso sólo despierta cuando el evento se cumple, ignorando las señales recibidas. `TASK_UNINTERRUPTIBLE` se utiliza en pocas ocasiones, generalmente en situaciones en que la espera va a ser muy corta; es decir, el evento por el que se espera es muy probable que ocurra y que ocurra pronto.

`TASK_STOPPED`. Un proceso se encuentra en este estado cuando su ejecución se ha detenido y no es elegible para ser ejecutado. Un proceso pasa a este estado a través de una de las siguientes señales: `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIFTTOU`. Los intérpretes de comandos (como `zsh` y `bash`) y los depuradores (como `gdb`) utilizan estas señales para controlar la ejecución de los procesos. Un proceso sale de este estado al recibir una señal `SIGCONT`, pasando al estado `TASK_RUNNING`.

`TASK_ZOMBIE`. Cuando un proceso finaliza su ejecución no es removido completamente del sistema, se coloca en este estado para que el proceso padre pueda recuperar información sobre el estado de finalización del proceso a través de la llamada a sistema `wait4()`.

2.1.2. El Algoritmo de Planificación

Algoritmo

La principal estructura utilizada por el algoritmo de planificación es la cola de procesos `runqueue_t`, la cual se define de la siguiente manera:

```
#define MAX_PRIO      140
#define BITMAP_SIZE  ((((MAX_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long))

struct prio_array {
    unsigned int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};

typedef struct prio_array prio_array_t;

struct runqueue {
    spinlock_t lock;
    unsigned long nr_running;
    unsigned long long nr_switches;
    unsigned long nr_uninterruptible;
    unsigned long expired_timestamp;
    unsigned long long timestamp_last_tick;
    task_t *curr, *idle;
    struct mm_struct *prev_mm;
    prio_array_t *active, *expired, arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;
};

typedef struct runqueue runqueue_t;
```

Aquí solo describiremos los campos más importantes de las estructuras, la descripción completa puede verse en el apéndice B.

Los elementos más importantes de `runqueue_t` son los 2 arreglos de prioridad `active` y `expired`, los cuales están definidos como apuntadores a estructuras `prio_array_t`. Estos 2 arreglos se utilizan de la siguiente manera. El arreglo de procesos activos `runqueue_t`→`active` contiene colas (una cola por cada prioridad que maneja Linux) con los procesos listos para ejecución (`TASK_RUNNING`) que aún tienen tiempo de ejecución asignado (timeslice o quantum). El arreglo de procesos expirados `expired` contiene

a los procesos listos para ejecución que han agotado su tiempo de ejecución asignado.

Cuando un proceso pasa al estado `TASK_RUNNING` es colocado en la cola correspondiente a su prioridad en el arreglo de procesos activos `runqueue_t→active` y el bit correspondiente en el mapa de bits `prio_array_t→bitmap` es puesto a `1`. De ésta manera, para determinar cuál es el proceso siguiente a ejecutar, sólo es cuestión de encontrar el primer bit con el valor `1` en el mapa de bits, y tomar el primer elemento en la cola de procesos correspondiente. Linux siempre ejecuta el proceso con mayor prioridad (siendo `0` la mayor prioridad y `139` la menor). La Figura 2.2 muestra la relación de los elementos de la estructura `prio_array_t`.

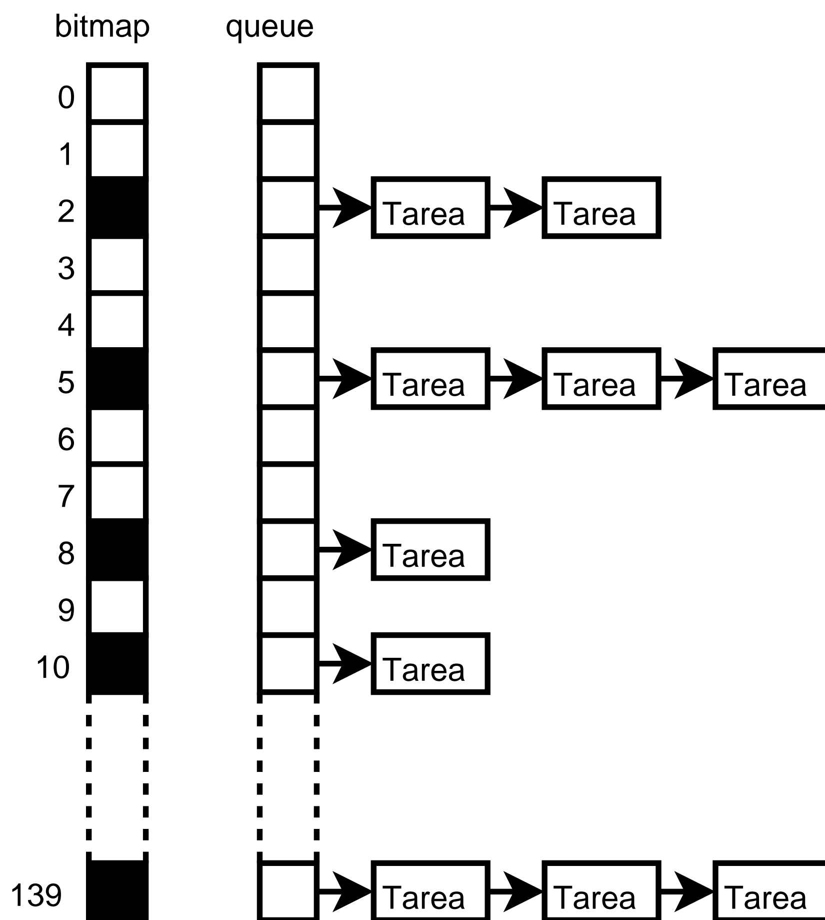

Figura 2.2.: Los arreglos de prioridad (`prio_array_t`)

Cuando un proceso agota su tiempo de ejecución asignado es removido del arreglo de prioridad `runqueue_t→active`, se le asigna su siguiente tiempo de ejecución y se

coloca en el arreglo de prioridad `runqueue_t→expired`.

Cuando todos los procesos han agotado su tiempo de ejecución asignado, el arreglo de prioridad `runqueue_t→active` queda vacío. En este momento, Linux intercambia los 2 arreglos de la siguiente manera:

```
array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
}
```

Así, como este intercambio y el encontrar el primer bit con valor `1` son operaciones que se realizan con un tiempo acotado superiormente, sin importar la cantidad de procesos en el sistema, el planificador de Linux es de complejidad $O(1)$.

Asignación Dinámica de Prioridades

El comportamiento de los procesos se puede clasificar en 2 tipos: los procesos interactivos (orientados a Entrada/Salida, en inglés I/O-Bound) y los procesos no interactivos (orientados al uso de procesador, en inglés CPU-Bound).

Los procesos interactivos son aquellos que se bloquean continuamente, ya sea esperando entrada del teclado, lectura y/o escritura de datos en disco, datos de la red, etc. En esta categoría se encuentran los editores de texto, hojas de cálculo, etc. Es conveniente que los procesos que esperan entrada del usuario respondan rápidamente. También es conveniente que los procesos que leen continuamente del disco sean planificados tan pronto como se tengan los datos listos, de esta manera, el proceso podrá solicitar más datos y el procesador se liberará rápidamente para atender otros procesos.

Los procesos no interactivos son los que hacen uso intensivo del procesador, sin requerir una cantidad grande de entrada/salida de datos. Ejemplos de procesos no interactivos son los compiladores, codificadores de audio/video, etc. Los procesos no interactivos no requieren ofrecer una respuesta rápida al usuario. El usuario no notará la diferencia si el proceso tarda unos segundos más en ofrecer la respuesta.

Es por esto que Linux intenta determinar si un proceso es interactivo o no interactivo para poder asignarle dinámicamente una prioridad, de tal manera que se mejore el rendimiento del sistema. Cada proceso, al crearse, obtiene una prioridad estática determinada por el usuario. Esta prioridad es conocida como el valor `nice` del proceso y sus valores están en el rango de -20 a 19, siendo -20 la mayor prioridad. Este valor se almacena en la variable `static_prio` del descriptor de proceso (`task_struct`, ver Sección 2.1.3) y no cambia en toda la vida del proceso.

La prioridad `static_prio` de un proceso sirve como base para determinar su prioridad dinámica. Dependiendo del comportamiento del proceso (su interactividad), éste puede obtener una bonificación o una penalización en su prioridad, la cual puede ser desde -5 a 5. La prioridad resultante de sumar la bonificación/penalización a `static_prio` es la prioridad dinámica, la cual es almacenada en el campo `prio` de `task_struct`.

$$\begin{aligned} \texttt{bonus} &\in [-5, 5] \\ \texttt{prio} &= \texttt{static_prio} - \texttt{bonus} \end{aligned}$$

La función `effective_prio()` calcula la prioridad dinámica de un proceso.

```c
#define HZ                   1000     /* timer interrupts/second */
#define DEF_TIMESLICE        (100 * HZ / 1000) /*100 ms*/
#define MAX_BONUS            10
#define MAX_SLEEP_AVG        (DEF_TIMESLICE * MAX_BONUS)
#define NS_TO_JIFFIES(TIME)  ((TIME) / (1000000000 / HZ))

#define CURRENT_BONUS(p) \
    (NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS / \
        MAX_SLEEP_AVG)

static int effective_prio(task_t *p)
{
    int bonus, prio;

    if (rt_task(p))
        return p->prio;

    bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;
    prio = p->static_prio - bonus;

    if (prio < MAX_RT_PRIO)
        prio = MAX_RT_PRIO;
    if (prio > MAX_PRIO-1)
        prio = MAX_PRIO-1;

    return prio;
}
```

Primero se verifica que el proceso no sea de tiempo real. Además de los 40 niveles de prioridad normales, Linux provee 100 niveles de prioridad para procesos con requerimientos de tiempo real. Los procesos de tiempo real son los de más alta prioridad y no tienen prioridad dinámica.

La base para determinar la prioridad dinámica es la variable `task_struct→sleep_avg`. Esta variable representa el tiempo promedio que un proceso duerme. Cada que un proceso despierta, el tiempo que pasó dormido se agrega a ésta variable, evitando que sobrepase el valor de `MAX_SLEEP_AVG`. Cada que el proceso va a dormir, el tiempo que pasó ejecutándose se resta a esta variable.

La bonificación/penalización se calcula con la macro `CURRENT_BONUS`, la cual, simplemente escala la variable `task_struct→sleep_avg` entre los valores `0` y `MAX_BONUS` (el cual es `10`). Al valor regresado por `CURRENT_BONUS` se le resta la mitad de `MAX_BONUS` para que la bonificación/penalización quede con valores negativos (penalización, el proceso es no interactivo) y positivos (bonificación, el proceso es interactivo). Esta bonificación/penalización se resta a la prioridad estática.

La última parte de la función simplemente revisa que la prioridad dinámica no rebase los límites permitidos para las prioridades de procesos normales.

Asignación de Tiempos de Ejecución (Quantum o Timeslice)

El tiempo de ejecución asignado a cada proceso es dependiente únicamente de su prioridad estática. La función `task_timeslice()` se encarga de calcularlo escalando los valores de la prioridad asignada por el usuario a los valores:

$$[-20 \ldots 0 \ldots 19] = [800\text{ms} \ldots 100\text{ms} \ldots 5\text{ms}]$$

```c
#define MAX_USER_RT_PRIO        100
#define MAX_RT_PRIO             MAX_USER_RT_PRIO        /* 100 */
#define MAX_PRIO                (MAX_RT_PRIO + 40)      /* 140 */

#define USER_PRIO(p)            ((p)-MAX_RT_PRIO)       /* p - 100 */
#define MAX_USER_PRIO           (USER_PRIO(MAX_PRIO))   /* 40 */

#define MIN_TIMESLICE           max(5 * HZ / 1000, 1)   /* 5 */
#define DEF_TIMESLICE           (100 * HZ / 1000)       /* 100 */

#define SCALE_PRIO(x, prio) \
    max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO/2), MIN_TIMESLICE)
/*  max(x * (    140 - prio   ) / (        40/2        ), 5 )   */

#define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
/*                              (     100      + (nice) + 20) */

static unsigned int task_timeslice(task_t *p)
```

```
{
    if (p->static_prio < NICE_TO_PRIO(0))
        return SCALE_PRIO(DEF_TIMESLICE*4, p->static_prio);
    else
        return SCALE_PRIO(DEF_TIMESLICE, p->static_prio);
}
```

SMP, SMT y NUMA

Linux, al ser un sistema operativo multi-arquitectura, ha adquirido características muy complejas. En lo que se refiere a la administración de procesos, Linux soporta arquitecturas SMP (Symmetric Multi-Processing), SMT (Simultaneous Multi-Threading) y NUMA (Non-Uniform Memory Architecture), lo cual tiene grandes implicaciones en la planificación de procesos.

La principal complejidad que se introduce en Linux para soportar arquitecturas multiprocesador es mantener distintas estructuras `runqueue_t` para cada procesador. Así, cada procesador tiene sus propias colas de procesos. Linux intenta mantener un proceso durante la mayor parte de su ejecución en un mismo procesador. Pero es posible que un procesador, en algún momento, tenga más carga de trabajo que los otros procesadores. Esto implica que se debe hacer migración de procesos entre las distintas colas de procesos. La migración de procesos consiste en determinar la carga de los procesadores y mover los procesos de una cola de procesos a otra, intentando balancear la carga entre todos los procesadores.

Para soportar NUMA, donde el tiempo de acceso a un dato depende de su localización en la memoria, Linux utiliza los "dominios de planificación" (scheduler domains). Cada proceso pertenece a un dominio (normalmente es el mismo, pero puede ser cambiado), y cada dominio tiene varios grupos de procesos. Linux agrupa los procesadores de cada nodo de una arquitectura NUMA en un grupo. Así, un proceso sólo puede ser migrado hacia procesadores dentro de su propio dominio y, preferiblemente, hacia procesadores dentro de su mismo grupo.

El balance de carga se hace en cada interrupción de reloj.

2.1.3. El Descriptor de Proceso

Para mantener información acerca de cada uno de los procesos, Linux define la estructura `task_struct` en el archivo <`linux/sched.h`>. Dicha estructura es bastante grande, por lo cual solo describiremos sus campos más importantes. La descripción completa puede verse en el apéndice B.

A través del campo `task_struct→tasks` todos los procesos creados se encuentran en una lista ligada.

El campo `task_struct→state` mantiene el estado del proceso. Ver la Sección 2.1.1.

Dos campos muy importantes son `task_struct→mm` y `task_struct→active_mm`. Antes de definir su función, es importante hacer una distinción entre los 3 tipos diferentes de procesos que se usan en Linux: *Procesos Normales*, *Hilos de Usuario* e *Hilos de Núcleo*.

Los hilos de usuario son implementados conforme al estándar IEEE 1003.1 [16]. En Linux, la única diferencia entre los procesos normales y los hilos de usuario es que estos últimos comparten su espacio de direcciones (entre otros recursos, como los archivos abiertos y los manejadores de señales). Los hilos de núcleo son procesos que realizan tareas periódicas en el núcleo. Los hilos del núcleo son procesos que no requieren acceso a memoria de usuario, es decir, no requieren tablas de páginas propias, por lo tanto, no poseen un espacio de direcciones propio.

El campo `task_struct→mm` es un apuntador a una estructura de tipo `mm_struct`, la cual representa el espacio de direcciones del proceso (ver Capítulo 3). El campo `task_struct→mm` de cada uno de los procesos normales apunta a una estructura `mm_struct` única. En cambio, dicho campo, en cada uno de los hilos de un mismo proceso, apuntan a la misma estructura `mm_struct`.

El campo `task_struct→mm` de los hilos de núcleo tiene el valor `NULL`, pues carecen de espacio de direcciones propio. Pero los hilos de núcleo aún requieren acceso a las tablas de páginas del núcleo para acceder a direcciones dentro del núcleo. Como se menciona en el capítulo 3, los procesos utilizan los primeros 3 GB de su espacio de direcciones virtuales para sus datos propios, y comparten el último GB con el núcleo, es decir, todos los procesos tienen las mismas tablas de páginas en su último GB (esto evita hacer cambios de contexto de memoria al entrar a modo núcleo). Aprovechando esto, los hilos de núcleo pueden “tomar prestado” el espacio de direcciones del proceso anterior para utilizar sus tablas de páginas sin hacer un cambio de contexto de memoria. El espacio de direcciones “prestado” se almacena en el campo `active_mm` de la estructura `task_struct`. Obviamente, los procesos normales y los hilos de usuario tienen el mismo valor en sus campos `mm` y `active_mm`.

Otro campo importante es `task_struct→binfmt`. Linux soporta diferentes formatos de archivos ejecutables (ELF, a.out y ECOFF). Para soportarlos, utiliza estructuras de tipo `linux_binfmt` definida así (en el archivo <`linux/binfmts.h`>):

```
struct linux_binfmt {
    struct linux_binfmt * next;
    struct module *module;
    int (*load_binary)(struct linux_binprm *,
                       struct  pt_regs * regs);
```

```
    int (*load_shlib)(struct file *);
    int (*core_dump)(long signr, struct pt_regs * regs,
                     struct file * file);
    unsigned long min_coredump;
};
```

Donde los apuntadores `load_binary`, `load_shlib` y `core_dump` son funciones para cargar el archivo, cargar una librería dinámica y hacer un volcado de memoria, respectivamente, y que son específicas del formato del archivo ejecutable.

Para ayudar a los intérpretes de comandos (shells) en su trabajo de controlar los procesos iniciados por el usuario, cada proceso tiene 4 identificadores:

`PID` **(Process ID)** Es el identificador principal del proceso. Cada proceso tiene un único PID. El campo `task_struct→pid` contiene el PID del proceso.

`TGID` **(Thread Group ID)** El estándar POSIX define que todos los hilos que comparten un espacio de direcciones deben compartir el mismo PID. Linux almacena en el campo `task_struct→tgid` el PID del primer hilo del grupo de hilos con el mismo espacio de direcciones. La llamada a sistema `getpid()` regresa este valor.

`PGID` **(Process Group ID)** El identificador del grupo de procesos. Un grupo de procesos son todos los procesos ejecutados por un sólo proceso, es decir, todos sus hijos y los hijos de sus hijos, etc, comparten el PGID.

`SID` **(Session ID)** . Todos los procesos ejecutados por una misma sesión del intérprete de comandos (shell) comparten el SID.

Estos identificadores son utilizados por los intérpretes de comandos para determinar, por ejemplo, cuáles procesos debe terminar cuando el usuario termina la sesión. Estos identificadores son almacenados en el arreglo `task_struct→pids`.

Para mantener organizados los procesos conforme a su relación con otros procesos, la estructura `task_struct` contiene los siguientes campos:

`task_struct→parent` Apuntador a la estructura `task_struct` del padre de este proceso.

`task_struct→children` Lista de todos los hijos de este proceso.

`task_struct→sibling` Lista de todos los hermanos de este proceso.

Otros campos importantes son: `task_struct→uid` y `task_struct→gid`. Indican, respectivamente, el identificador de usuario que inició el proceso y el identificador del grupo al cual pertenece el usuario. Pero un usuario puede pertenecer a varios grupos, para lo cual existe el campo `task_struct→group_info` que contiene información sobre los grupos a los cuales pertenece el usuario dueño del proceso.

Linux posee un mecanismo para limitar los recursos que puede poseer un usuario. El campo `task_struct→user` es un apuntador a una estructura que contiene algunas estadísticas sobre los recursos que posee un usuario. Por ejemplo, mantiene la cantidad de procesos que el usuario está ejecutando y la cantidad de archivos abiertos, entre otras.

Las primeras versiones de Unix tenían el problema de que el usuario "root" tenía todos los permisos y cualquier otro usuario carecía de todos los permisos. Actualmente, los permisos para hacer distintas acciones han sido definidas de una manera más fina y permiten que ciertos procesos puedan adquirir capacidades para realizar sólo ciertas acciones. Los siguientes campos implementan las capacidades del proceso:

`task_struct→cap_effective` Es un mapa de bits, donde cada bit indica si el proceso tiene el permiso para realizar cierta acción. En `<linux/capability.h>` están definidas las capacidades de una manera muy detallada.

`task_struct→cap_inheritable` Es un mapa de bits que determina las capacidades que el proceso mantendrá después de ejecutar la llamada a sistema `exec()`.

`task_struct→cap_permitted` Es un mapa de bits que indica las capacidades que el proceso puede adquirir.

`task_struct→keep_capabilities` Bandera que indica si las capacidades serán mantenidas después de ejecutar la llamada a sistema `exec()`.

Los siguientes campos están relacionados con la administración de archivos pero serán vistos aquí debido a que es información exclusiva de cada proceso.

`task_struct→fs` Es un apuntador a una estructura de tipo `fs_struct` definida en el archivo `<linux/fs_struct.h>`.

```
struct fs_struct {
    atomic_t count;
    int umask;
    struct dentry * root, * pwd, * altroot;
    struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
};
```

La cual, principalmente, contiene apuntadores a las estructuras `dentry` y `vfsmount` (ver capítulo 5) del directorio raíz del proceso, el directorio actual del proceso, y el directorio raíz alterno del proceso (puede ser cambiado con la llamada a sistema `chroot()`).

`task_struct→files` Es un apuntador a una estructura de tipo `files_struct` definida en <`linux/file.h`>, que contiene información sobre los archivos abiertos por el proceso (ver capítulo 5).

`task_struct→proc_dentry` Es una estructura `dentry` asociada al proceso en el sistema de archivos virtual `procfs` (montado típicamente en `/proc`). En dicho sistema de archivos hay un directorio por cada proceso con información sobre su entorno.

2.1.4. Inicialización

La función `start_kernel()` es la encargada de inicializar los distintos subsistemas básicos del núcleo. Esta función es llamada por el código en ensamblador dependiente de la arquitectura, después de haber hecho algunas inicializaciones básicas, como las tablas de páginas del mapeo del núcleo.

La Figura 2.3 muestra la secuencia de funciones relativas a la administración de procesos que son llamadas por `start_kernel()`.

`setup_arch()`, `setup_memory()`, `init_bootmem()` y `mem_init()` son funciones referentes a la administración de memoria, y sólo son incluidas aquí para mostrar un panorama más general de `start_kernel()`. Éstas funciones son discutidas en el capítulo 3.

La primera función llamada por `start_kernel()`, relativa a la administración de procesos, es `sched_init()` cuya función es inicializar la estructura `runqueue_t`. Esto requiere inicializar los apuntadores a los arreglos de prioridad `active` y `expired`, inicializar los mapas de bits de cada uno de ellos, e inicializar las `140` cabeceras de colas en el arreglo `queue`. También incrementa la cuenta de `init_mm`, la cual es una estructura `mm_struct` que se inicializa en tiempo de compilación y representa el espacio de memoria utilizado por el núcleo (ver Sección 3).

Posteriormente, llama a `init_idle()`, la cual inicializa las variables, relacionadas con el administrador de procesos, del proceso nulo (que se ejecuta mientras no existan más tareas listas) de tal manera que tenga la más baja prioridad.

La siguiente función utilizada es `pidhash_init()`, la cual inicializa 4 tablas hash, una para cada uno de los distintas formas en que Linux agrupa los procesos.

Para cada una de estas formas, se reserva un arreglo de tamaño dependiente de la memoria RAM disponible (con un mínimo de 16 elementos y un máximo de 4096). Cada posición del arreglo es una cabecera de una lista de elementos que colisionan en

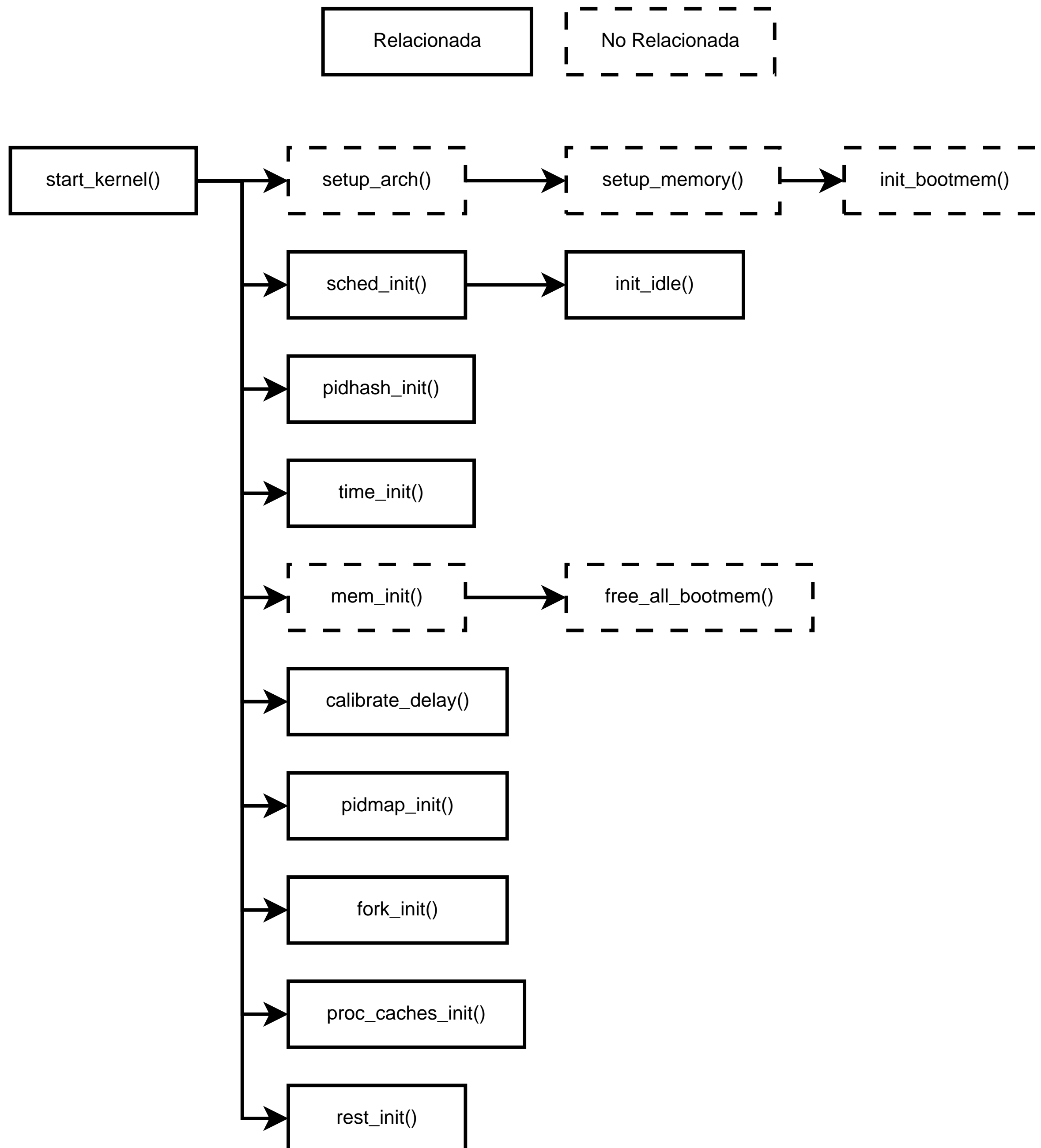

Figura 2.3.: Secuencia de Inicialización del Administrador de Procesos

dicha posición. Y cada elemento de esa lista es, así mismo, una lista de procesos con el mismo ID con el cual se están agrupando los procesos en el hash. La Figura 2.4 muestra la organización del hash basado en el TGID (Thread Group ID).

colisiones
mismo TGID
pid_hash[PIDTYPE_TGID]

Figura 2.4.: Organización de `pid_hash`

La siguiente función, `time_init()`, se encarga de guardar en la variable `xtime` el tiempo actual, leyéndolo del reloj del sistema. También se encarga de instalar el manejador de la interrupción de reloj.

`calibrate_delay()` calcula la variable global `loops_per_jiffy`, que es la cantidad de ciclos que el CPU ejecuta entre cada interrupción de reloj. Con esta medida se implementan funciones para introducir retrasos en el código como `udelay()` que espera por la cantidad de microsegundos especificados.

Después, `pidmap_init()` inicializa la primera página de un mapa de bits que se utiliza para encontrar PID's libres y reserva el PID `0`, el cual es utilizado para el proceso nulo (idle).

`fork_init()` inicializa un slab (caché de estructuras muy utilizadas, ver Capítulo 3 para más detalles) para las estructuras `task_struct` y determina la cantidad máxima de procesos que el sistema soporta dependiendo de la cantidad de memoria RAM, asegurándose que ésta cantidad no sea menor a `20`.

`proc_caches_init()` inicializa los siguientes slabs relacionados con los procesos:

`sighand_cache` Slab de estructuras `sighand_struct`; estas estructuras describen un manejador de señales de un proceso.

`signal_cache` Contiene estructuras `signal_struct` que son datos compartidos entre hilos de un mismo proceso.

`files_cache` Estructuras `files_struct` que contienen información sobre los archivos utilizados por cada proceso.

`fs_cache` Estructuras de tipo `fs_struct` que contienen la jerarquía de directorios correspondientes a cada proceso, como su directorio actual y su directorio raíz.

`vm_area_struct` Slab de estructuras `vm_area_struct`. Cada una de ellas representa una sección contigua en el espacio de direcciones virtuales de un proceso.

`mm_struct` Estructuras `mm_struct` que definen el espacio virtual completo de un proceso. Contiene varias estructuras `vm_area_struct`.

Finalmente, `rest_init()` ejecuta el programa `/sbin/init`, con PID 1, como el primer proceso ejecutado por Linux, y después entra en un ciclo (`cpu_idle()`) convirtiéndose en el proceso nulo. El programa `/sbin/init` es el padre de todos los procesos; se encarga de ejecutar todos los procesos indicados en el archivo `/etc/inittab`. Comúnmente, arranca servicios del sistema como servidores web, servidores de correo electrónico, el shell y el entorno gráfico.

2.2. Simplificación de la Administración de Procesos en eLinux

El Cuadro 2.1 muestra algunos datos sobre la cantidad de código referente a la administración de procesos en `linux 2.6.10`. Podemos ver que la administración de procesos no abarca gran cantidad del código de Linux. La administración de procesos es una parte central de Linux y no puede ser modificada de manera sustancial. Sólo se decidió remover algunas características que son opcionales en el núcleo de Linux y que agregan mucha complejidad.

Para eLinux decidimos remover el código referente al soporte de arquitecturas como SMP, SMT y NUMA. Esto implicó remover el código referente a los dominios de planificación, el balance de carga y la migración de procesos. En casi todas las funciones relacionadas con la administración de procesos existía código dependiente de estas opciones de configuración.

También se ha eliminado el soporte para distintos formatos de archivos binarios. Linux soporta los formatos: ELF, a.out y ECOFF. En eLinux sólo hemos mantenido el soporte para el formato ELF, debido a que los otros formatos son obsoletos o muy poco utilizados.

Otra opción de configuración removida fue la de Software Suspend. Esta opción permite “congelar” todos los procesos, guardarlos en disco duro, apagar la máquina y reiniciarla en el estado en que se encontraba.

Tamaño	2,6 MB
Archivos .c	160
Archivos .S (ensamblador)	11
Archivos .h	11
Total de archivos de código	182
Porcentaje del total en Linux	1,26 %
Líneas de código en archivos .c	77, 026
Líneas de código en archivos .S (ensamblador)	2, 421
Líneas de código en archivos .h	1, 061
Total de líneas de código	80, 508
Porcentaje del total en Linux	1,34 %

Cuadro 2.1.: Estadísticas sobre la administración de procesos en Linux 2.6.10 (directorios `linux-2.6.10/kernel/`, `linux-2.6.10/arch/i386/kernel` y `linux-2.6.10/init/`).

Con estas opciones removidas, la cantidad de código referente a la administración de procesos se redujo casi a la mitad. El Cuadro 2.2 muestra las mismas estadísticas del cuadro 2.1, pero respecto al código en eLinux. En él podemos ver que la administración de procesos representa una gran parte del código eLinux. Esto es debido a que en otros subsistemas se removió mucho más código del que fue posible remover en este subsistema.

2.3. Tareas y Proyectos

1. Estudiar la interrupción de reloj en eLinux.

   Linux instala la función `timer_interrupt()` como la interrupción de reloj. La Figura 2.5 muestra las funciones ejecutadas por ella. Estudiar la implementación y la funcionalidad de cada una de ellas.

2. Estudiar la implementación de la función `do_fork()` y las funciones que ésta utiliza.

   En Linux, las llamadas a sistema `fork()`, `clone()` y `vfork()` son implementadas a través de la función `do_fork()` definida en `kernel/fork.c`. En resumen, esta función hace lo siguiente:

   a) Reservar un nuevo `pid` para el nuevo proceso con ayuda de la función

Tamaño	1,5 MB
Archivos .c	93
Archivos .S (ensamblador)	8
Archivos .h	4
Total de archivos de código	105
Porcentaje del total en eLinux	27,48 %
Líneas de código en archivos .c	45, 541
Líneas de código en archivos .S (ensamblador)	1, 795
Líneas de código en archivos .h	127
Total de líneas de código	47, 463
Porcentaje del total en eLinux	15,28 %

Cuadro 2.2.: Estadísticas sobre la administración de procesos en eLinux (directorios `linux-2.6.10/kernel/`, `linux-2.6.10/arch/i386/kernel` y `linux-2.6.10/init/`).

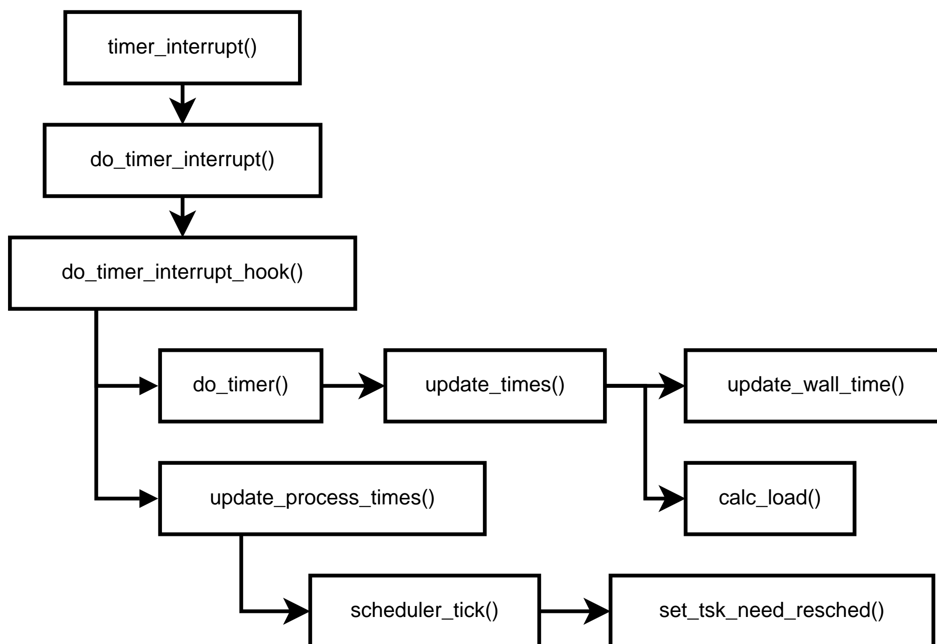

Figura 2.5.: La interrupción de reloj

`alloc_pidmap()`.

b) Ejecuta la función `copy_process()` que es la que realiza la copia completa del proceso, incluyendo su memoria, sus archivos abiertos, etc.

c) Incrementar la variable global `total_forks`, la cual lleva la cuenta de cuantas llamadas a `fork()`, `clone()` y `vfork()` se han realizado.

d) Finalmente, se regresa el `pid` del proceso hijo.

Capítulo 3.

Administrador de Memoria

Los procesos, para poder ser ejecutados, deben ser cargados en memoria. Pero la memoria siempre ha sido un recurso muy limitado y debe ser aprovechada al máximo.

El problema de encontrar la mejor forma de administrar la memoria ha llevado a establecer los siguientes requisitos que se deben satisfacer con la administración de memoria [35]:

Relocalización Permite que un programa pueda ser cargado en cualquier zona de la memoria física.

Protección Aisla a los procesos entre sí. Las direcciones que maneja un proceso no tienen alguna relación con las direcciones de otro proceso.

Compartimiento Permite, por ejemplo, que dos o más instancias de un programa compartan su sección de código en memoria física.

Modularización de los procesos Las distintas secciones de un proceso pueden tener diferentes permisos de acceso. Por ejemplo, la sección de código sólo debe contar con permisos de lectura y ejecución, mientras que la sección de datos puede tener permisos de lectura y escritura y no de ejecución.

Estos requisitos fueron satisfechos por primera vez con el uso de memoria virtual [12, 20]. En este capítulo veremos cómo Linux administra la memoria, tanto física, como virtual. Posteriormente, veremos cómo se decidió simplificarla para eLinux.

3.1. Organización de la Administración de Memoria en Linux

3.1.1. Organización de la Memoria Física

Linux tiene soporte para NUMA(Non Uniform Memory Access), una arquitectura multiprocesador en la que se tienen distintos bancos de memoria (nodos) y el tiempo de acceso a memoria depende de la localización de cada nodo.

Para administrar los distintos nodos de memoria, Linux mantiene una lista de estructuras, donde cada una representa un nodo distinto. Esta estructura está definida como `pg_data_t` y la lista es llamada `pgdat_list`.

Así mismo, cada nodo se divide en distintas zonas. Normalmente, son 3 zonas: `ZONE_DMA`, `ZONE_NORMAL` y `ZONE_HIGHMEM`. Cada zona es una región de memoria y no se traslapan entre ellas.

En la arquitectura x86, `ZONE_DMA` es la región de los primeros 16MB de memoria. Algunos dispositivos que realizan acceso directo a memoria (DMA) sólo pueden manejar hasta los primeros 16MB de memoria. la memoria en `ZONE_DMA` es utilizada para satisfacer peticiones de memoria de estos dispositivos.

`ZONE_NORMAL` es desde los 16MB hasta los 896MB. Esta zona puede accederse directamente y es la que más se usa para satisfacer peticiones de memoria.

`ZONE_HIGHMEM` es desde los 896MB en adelante. En la arquitectura x86, Linux no puede acceder directamente a zonas de memoria física mayores a 896MB (ver Sección 3.1.5). Para realizarlo, requiere mapear temporalmente esas direcciones a direcciones menores.

Cada zona es descrita por una estructura `zone`. Cada estructura `zone` contiene un arreglo de estructuras `struct page` llamado `zone_mem_map`. Cada `struct page`, representa una página de memoria física (marco de página). El tamaño de la página es de 4KB.

La Figura 3.1 muestra las relaciones de estas estructuras. Como podemos observar, la intención de estas estructuras es mantener bien organizados los marcos de página, los cuales son las estructuras básicas en la administración de la memoria.

En Linux, si no se activa el soporte para NUMA, sólo se tiene una estructura `pg_data_t`, contenida en la variable `contig_page_data`, por lo que no es necesaria la lista `pgdat_list`.

3.1.2. Tablas de Páginas

Linux maneja 3 niveles de tablas de páginas (a partir de la versión 2.6.11 Linux ya maneja 4 niveles para soportar completamente las arquitecturas de 64 bits) en todas

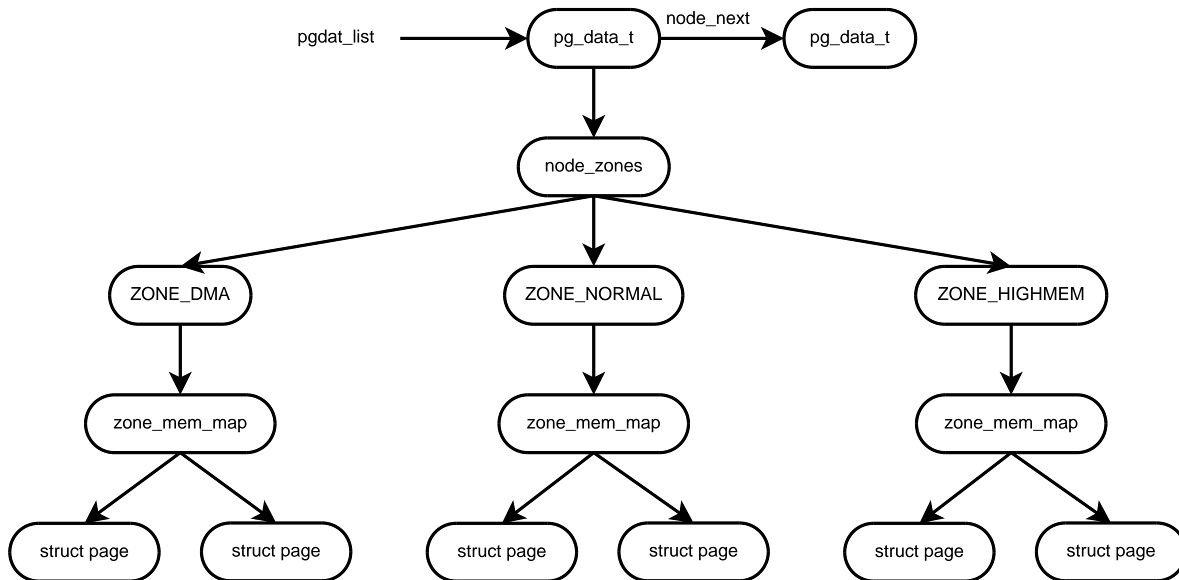

Figura 3.1.: Relaciones entre las estructuras principales en Linux

sus arquitecturas. Pero la arquitectura x86 sólo usa 2 niveles, y el código de Linux se encarga de que los accesos al nivel intermedio sean equivalentes al del nivel más alto.

La Figura 3.2 muestra la manera en que se descomponen las direcciones para acceder a las tablas de páginas en la arquitectura x86 (usando sólo 2 niveles).

3.1.3. Inicialización

Una buena manera de abordar la administración de memoria en Linux es siguiendo el proceso de inicialización de Linux. La Figura 3.3 muestra las funciones relacionadas con la administración de memoria que son ejecutadas por `start_kernel()`. En esta sección se discutirá la función de cada una de ellas. En la Figura 3.3 se incluyen algunas funciones que no están relacionadas directamente con la administración de memoria sólo para mostrar el contexto dentro del cual son ejecutadas.

Una de las primeras funciones relacionadas con la administración de memoria que se ejecuta en Linux es `setup_memory()`, la cual es llamada por `setup_arch()`. A finalizar `setup_memory()` se tiene listo el Boot Memory Allocator para ser utilizado.

3.1.4. Boot Memory Allocator

Durante el arranque de Linux, se requiere de un administrador de memoria sencillo y eficiente. `setup_memory()` se encarga de inicializar el llamado "Boot Memory Allo-

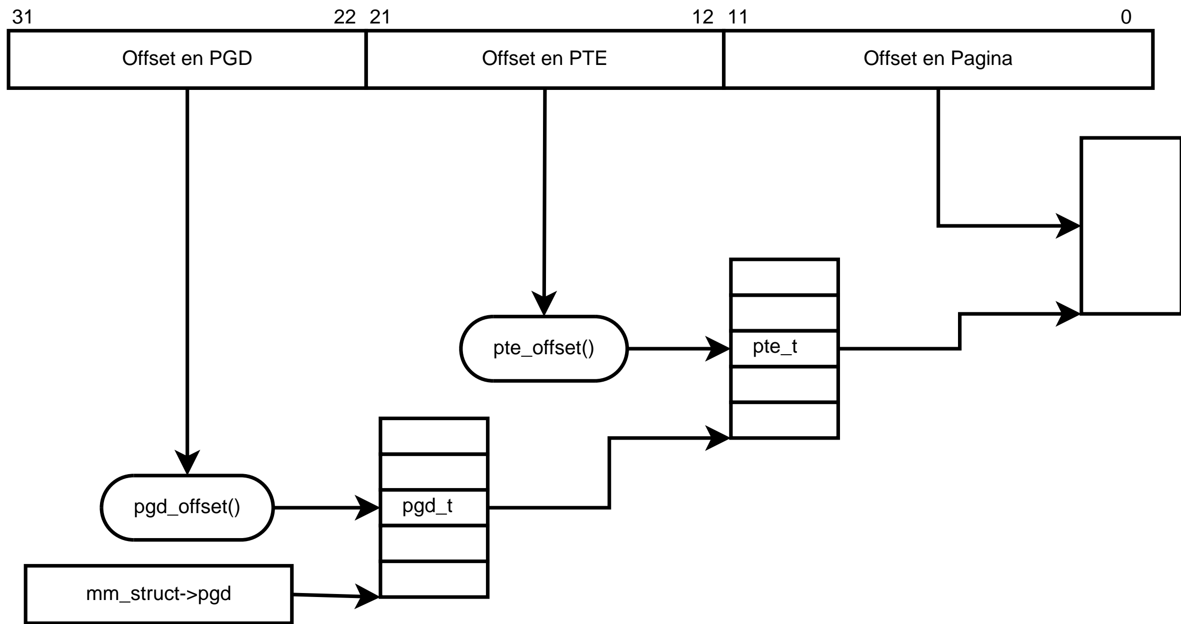

Figura 3.2.: Manejo de las tablas de páginas en eLinux

cator". Dicho administrador de memoria es del tipo "First Fit" [22], en el cual se usa un mapa de bits para representar la memoria. Para determinar el tamaño del mapa de bits se requiere conocer el tamaño de la memoria física.

En Linux, la función `setup_memory()` define los valores de 3 variables: `min_low_pfn`, `max_low_pfn` y `max_pfn`. Estas variables sirven, respectivamente, para definir el número del primer marco de página (PFN, por sus siglas en inglés) utilizable para Linux (justo después de la imagen cargada del núcleo), el fin de la `ZONE_NORMAL` y el último PFN disponible en el sistema.

En sistemas sin mucha memoria (menos de 896MB), `max_low_pfn` y `max_pfn` tienen el mismo valor, pues no se utiliza la zona de memoria alta (`ZONE_HIGHMEM`).

La función `find_max_low_pfn()` busca, en un mapa de la memoria proporcionado por el BIOS, el PFN más grande disponible. Este mapa lo construye el BIOS en tiempo de arranque y Linux lo consulta y arma el mapa durante las primeras etapas del arranque.

La Figura 3.4 muestra el mapa de la memoria física en Linux. El primer megabyte de la memoria física no es utilizado, pues existen algunos BIOS que utilizan esta zona para almacenar datos. Los siguientes 7 MB son utilizados para cargar el núcleo. El mapa de bits se almacena inmediatamente después del núcleo, es decir en la dirección especificada por `min_low_pfn`. Cuando hay menos de 896MB disponibles, `max_low_pfn` y `max_pfn` son iguales y contienen el último número de marco de página disponible.

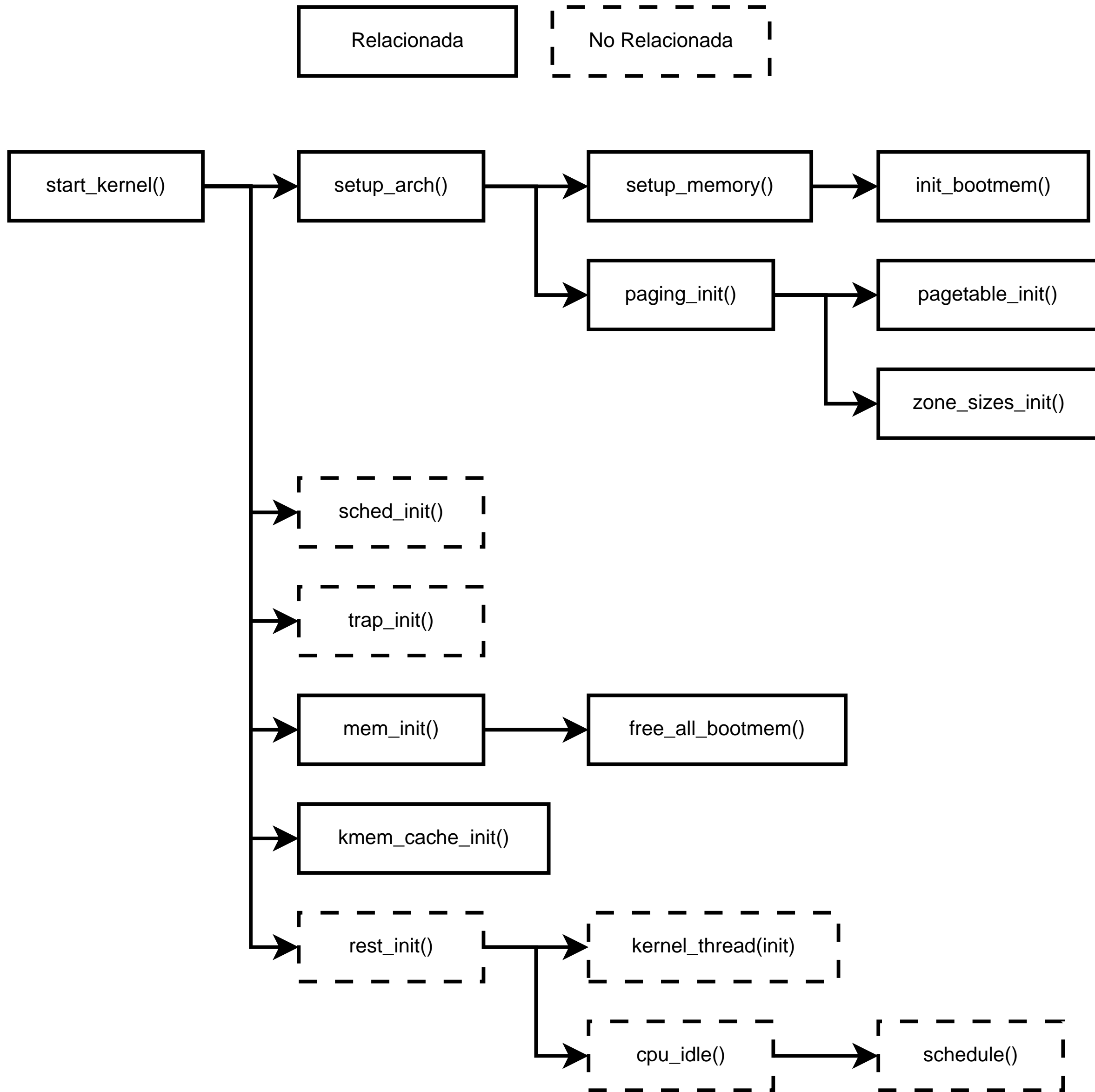

Figura 3.3.: Secuencia de inicialización del administrador de memoria

Cuando hay más de 896MB, `max_low_pfn` contiene el número de marco de página en los 896MB, y `max_pfn` contiene el último disponible.

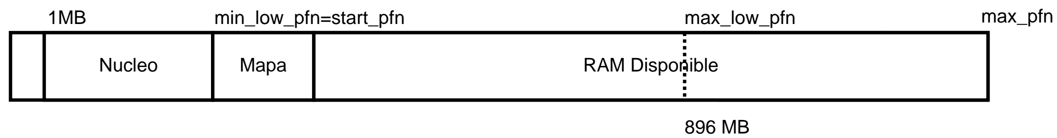

Figura 3.4.: Mapa de memoria en eLinux

3.1.5. Tablas Maestras de Páginas del Núcleo

Al finalizar `setup_memory()`, el Boot Memory Allocator tiene un mapa de toda la memoria disponible en el sistema, pero aún no se han inicializado las tablas de páginas para todas ellas.

En un inicio, el código de `startup_32()` (función de inicialización del núcleo, escrita en ensamblador, ejecutada por el cargador de arranque y que es la encargada de ejecutar la función `start_kernel()`) inicializa las tablas de páginas para los primeros 8MB de memoria virtual. En estos 8 MB está mapeada la imagen del núcleo cargado. El resto de las tablas de páginas, son inicializadas por `paging_init()`, la cual es llamada por `setup_arch()` después de `setup_memory()`.

```
void __init paging_init(void)
{
    pagetable_init();
    load_cr3(swapper_pg_dir);
    __flush_tlb_all();
    zone_sizes_init();
}
```

Esta función ejecuta `pagetable_init()` la cual hace toda la inicialización de las tablas de páginas del kernel. Carga el valor de `swapper_pg_dir` (variable que apunta al PGD (Page Global Directory) inicializado en `pagetable_init()`) en el registro cr3, el cual es el registro del procesador que tiene un apuntador al PGD del proceso actual.

`__flush_tlb_all()` limpia los Translation Lookaside Buffers (TLB) para invalidar todas las entradas, ya que acabamos de establecer las tablas de páginas.

Finalmente, ejecuta `zone_sizes_init()`. Ésta función inicializa los datos (como su tamaño) de las zonas de memoria (`ZONE_DMA`, `ZONE_NORMAL` y `ZONE_HIGHMEM`).

`pagetable_init()` crea las tablas de páginas que utiliza el núcleo conocidas como las Tablas Maestras de Páginas del Núcleo (Master Kernel Page Tables). Estas tablas

mapean desde las direcciones virtuales `0xC0000000` (definido como `PAGE_OFFSET`) hasta la `0xFFFFFFFF`, lo cual corresponde al último gigabyte de memoria virtual disponible en la arquitectura i386 (32 bits). Los primeros 896MB de este rango son un mapeo directo de la memoria física disponible, sumándole, a cada dirección física, la cantidad de `0xC0000000`. Así, la dirección física para las direcciones virtuales de los primeros 896MB arriba de los 3GB es fácilmente calculable de la siguiente manera:

$$\text{Dirección Física} = \text{Dirección Virtual} - \texttt{PAGE_OFFSET}$$

Los últimos 128MB son utilizados para distintos propósitos, como el mapeo temporal de direcciones en la zona de memoria alta (`ZONE_HIGHMEM`).

Las Tablas Maestras son copiadas por todos los procesos para acceder al núcleo. Es decir, los primeros 3GB del espacio de direcciones virtuales de cada proceso es privado a cada proceso, mientras que el último gigabyte es compartido, pero con restricciones de acceso. De esta manera, los procesos pueden entrar al modo núcleo sin requerir cambiar de espacio virtual y, por lo tanto, limpiar los TLB's, lo cual es una operación muy costosa.

3.1.6. Binary Buddy Allocator

`start_kernel()` inicializa varios subsistemas, los cuales utilizan el Boot Memory Allocator para reservar la memoria que utilizan. El Boot Memory Allocator es muy lento para ser utilizado normalmente. Por esto, después que se han inicializado los subsistemas básicos, el Boot Memory Allocator es reemplazado por el Binary Buddy Allocator. La función encargada de remover al Boot Memory Allocator e inicializar el Binary Buddy Allocator es `mem_init()`, que es llamada por `start_kernel()`.

El funcionamiento del Binary Buddy Allocator fue descrito en [21]. La implementación en Linux (en `mm/page_alloc.c`) funciona de la siguiente manera.

Linux agrupa las páginas libres en 10 listas. Cada una de estas listas, respectivamente, contiene grupos de 1, 2, 4, 8, 16, 32, 64, 128, 256 y 512 páginas libres contiguas.

Linux mantiene estas listas en un arreglo de 10 estructuras de tipo `free_area`:

```
struct free_area {
    struct list_head    free_list;
    unsigned long       *map;
};
```

El k-ésimo elemento del arreglo contiene una lista de bloques de 2^k páginas libres contiguas (k es también conocido como el "orden" de la asignación). Así mismo, cada elemento contiene un mapa de bits, donde cada bit representa un par de páginas libres: es 0 si ambas páginas están libres o ambas están reservadas, 1 en caso contrario (sólo

una de las páginas está reservada). Cabe aclarar que en las listas sólo se liga la primer página de cada bloque. La Figura 3.5 muestra el uso de estas estructuras.

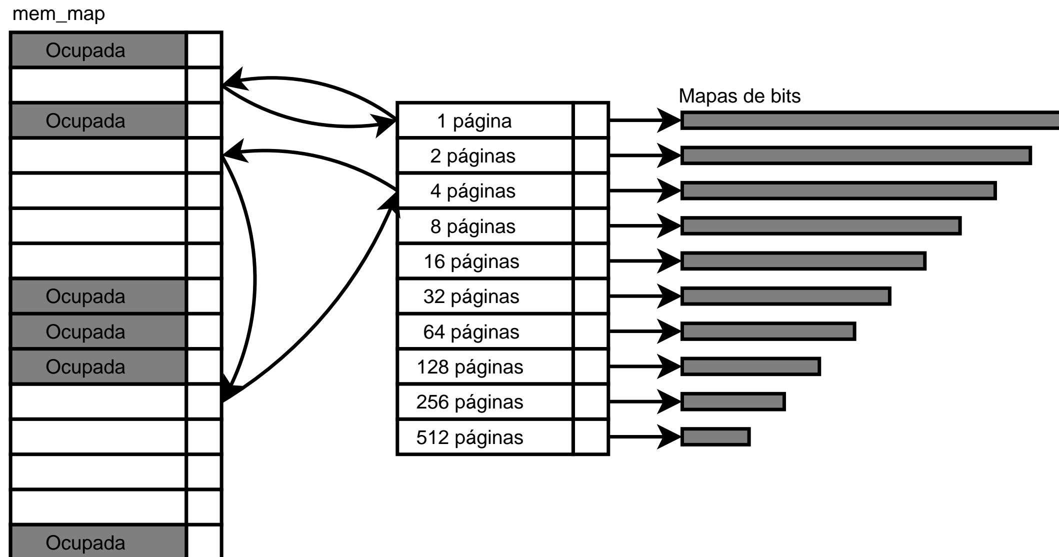

Figura 3.5.: Binary Buddy Allocator

Cuando se recibe una petición de, por ejemplo, un bloque de 4 páginas (orden 2), primero se revisa la lista de bloques de páginas de orden 2. Si hay un bloque libre, se asigna dicho bloque. En caso contrario, se revisa la lista de bloques de páginas de orden 3. Si hay un bloque libre, se divide por la mitad, asignando una mitad a la petición realizada, y la otra mitad se coloca en la lista de bloques de páginas libres de orden 2. Si no lo hay se continúa con la lista de orden 4, y así sucesivamente. La función `alloc_pages()` se encarga de atender peticiones de memoria.

Cuando se libera memoria, se revisa si el bloque de a un lado también está libre. De ser así, se juntan los dos bloques para formar uno de un orden superior. Esto es recursivo, si el nuevo bloque de orden superior queda junto a un bloque libre, entonces se forma un nuevo bloque del siguiente orden. La función `free_pages()` implementa la liberación de memoria.

El Binary Buddy Allocator es muy eficiente evitando la fragmentación externa, ya que la manera en que reserva y libera las páginas favorece el agrupamiento tanto de páginas libres como de páginas reservadas.

Como mencionamos anteriormente, `mem_init()` es la función encargada de remover el Boot Memory Allocator e inicializar el Binary Buddy Allocator. `mem_init()` lo único que hace es calcular algunas estadísticas y llamar a `free_all_bootmem_core()` que básicamente revisa el mapa de bits del Boot Memory Allocator para determinar las

páginas libres y marcarlas como libres para el Buddy Allocator haciendo uso de la función `free_pages()`.

3.1.7. Slab Allocator

El Binary Buddy Allocator usa como unidad básica de asignación a la página, es decir, bloques de 4 KB. Esto quiere decir que todas las peticiones de memoria deben ser dadas en múltiplos de 4 KB. Si se requieren sólo unos cuantos bytes de memoria, es necesario reservar toda una página, desperdiciando mucho espacio. Esto es conocido como fragmentación interna.

Linux, para evitar la fragmentación interna y para obtener algunos otros beneficios, hace uso del llamado Slab Allocator, introducido por Bonwick [6] para el sistema operativo SunOS.

El Slab Allocator surgió de la necesidad de mantener cachés de objetos. En sistemas operativos como SunOS, ciertas estructuras tenían que ser inicializadas cada ocasión que eran reservadas. El costo de la inicialización era significativamente más alto que el costo de la reservación. Es por esto, que Bonwick diseñó el Slab Allocator con constructores y destructores, con el propósito de preservar la porción "invariante" del estado inicial de los objetos. Así, las estructuras no tenían que ser destruidas, sino que eran regresadas al Slab Allocator para que fueran reutilizadas, sin el costo de una nueva inicialización.

El diseño de Linux no requiere que las estructuras sean inicializadas y destruidas. Aún así, el Slab Allocator ofrece otras ventajas. La principal es evitar llamar al Binary Buddy Allocator. A pesar de que es muy eficiente, invocarlo para cada petición de memoria puede resultar ineficiente.

Otra ventaja es evitar la fragmentación interna. Ya que el Binary Buddy Allocator utiliza como unidad básica de asignación a la página (4 KB), es necesario utilizar algún mecanismo que permita utilizar más eficientemente dichas páginas.

Una manera de hacer esto es utilizar el Slab Allocator para la reservación de las estructuras más frecuentemente utilizadas dentro del núcleo de Linux. Así,estructuras como `task_struct`, `mm_struct`, `buffer_head`, `inode`, `dentry`, entre muchas otras, son reservadas y liberadas a través del Slab Allocator.

El funcionamiento del Slab Allocator es sencillo. Su implementación se localiza en `mm/slab.c`. La organización de las cachés es a través de las estructuras: `kmem_cache_t` y `slab`. Cada caché está representada por una estructura `kmem_cache_t` y cada uno de estas contiene uno o más estructuras `slab`. La Figura 3.6 muestra la relación entre estas estructuras.

Cada `slab` tiene reservada una o más páginas de memoria utilizadas para los objetos. Al inicializar cada `slab`, se reserva una página, a través del Binary Buddy Allocator, para almacenar los objetos que quepan dentro de esa página. Dichos objetos se van

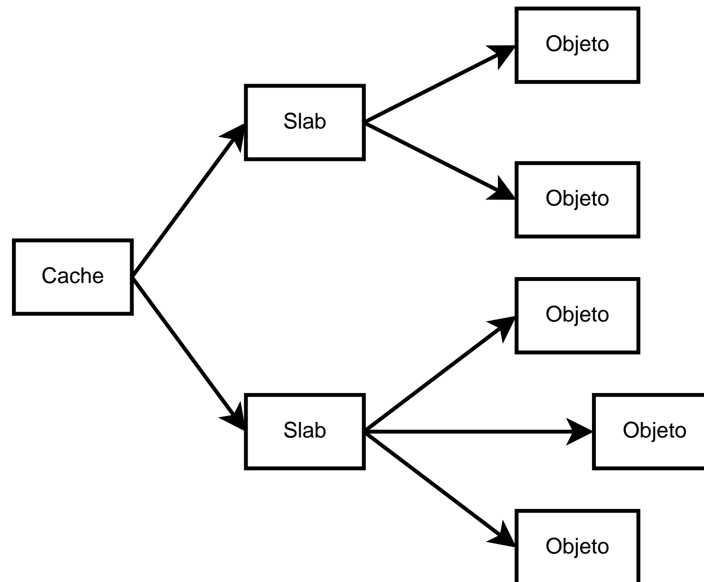

Figura 3.6.: Relación entre las estructuras del Slab Allocator.

asignando y liberando dentro de esta página. La Figura 3.7 muestra un ejemplo de una página reservada para estructuras de tipo `task_struct`. Si todos los objetos de la página están ocupados, y se requiere reservar otro objeto, se hace otra petición de memoria al Binary Buddy Allocator para reservar otra página. El algoritmo 1 muestra cómo se reserva un objeto usando el Slab Allocator.

Algoritmo 1 Reservación de un objeto.

```
if Hay un objeto libre en la caché then
    Asignarlo
else
    Reservar una página extra
    Asignar un objeto libre de la nueva página
end if
```

Si después de varias liberaciones de objetos, sucediera que algunas páginas están libre, el Slab Allocator no las libera al Binary Buddy Allocator, sino que las mantiene por la posibilidad de que nuevamente sean utilizadas. Estas páginas libres sólo son liberadas cuando el núcleo se encuentra en una situación de poca memoria disponible e intenta recuperar todas las páginas posibles.

Como ya mencionamos, existen varias cachés para las estructuras más utilizadas dentro del núcleo. Pero el núcleo también ofrece las funciones `kmalloc()` y `kfree()`

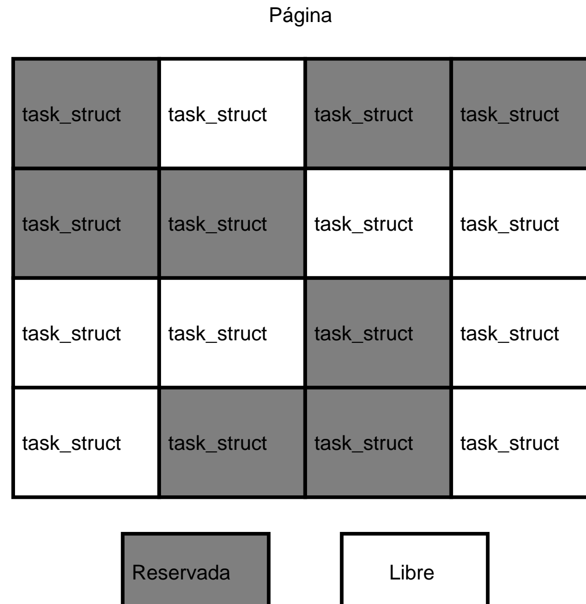

Figura 3.7.: Ejemplo de una página utilizada por el Slab Allocator.

para realizar peticiones de memoria de tamaño arbitrario (hasta 128 KB, para tamaños mayores, es recomendable usar directamente el Binary Buddy Allocator) de una manera muy similar a las funciones `malloc()` y `free()` de la librería estándar de C. Estas funciones están construidas sobre el Slab Allocator. Al inicializar el núcleo, se crean cachés para objetos de tamaños que van, en potencias de 2, desde 32 bytes hasta 128 KB.

De esta manera, `kmalloc()` sólo determina de qué caché se requiere reservar memoria, eligiendo la caché de menor tamaño donde quepa la petición realizada, y reserva un nuevo objeto de dicha caché. Así, se asegura que la fragmentación interna al usar `kmalloc()` es siempre menor al 50 %.

Inicialización del Slab Allocator

La función que inicializa el Slab Allocator es `kmem_cache_init()`, la cual es llamada por `start_kernel()` una vez que el Binary Buddy Allocator está listo para ser usado.

El Slab Allocator también contiene una caché para estructuras del tipo `kmem_cache_t`, el cual es el descriptor de las cachés. Pero para crear esta caché requiere una estructura `kmem_cache_t` previamente reservada. Esto se hace inicializando estáticamente (en tiempo de compilación) una estructura `kmem_cache_t` llamada `cache_cache`. Lo primero que hace la función `kmem_cache_init()` es inicializar completamente esta estructura. Una

vez inicializado el Slab Allocator, `kmem_cache_init()` crea las cachés para ser utilizadas por `kmalloc()`. De esta manera, las distintas partes del administrador de memoria quedan listas para ser utilizada en todo el núcleo.

3.1.8. Espacio de Direcciones de Proceso(Process Address Space)

El espacio de direcciones de un proceso está definido como el conjunto de identificadores que pueden ser usados por un proceso para acceder información [11].

El uso de memoria virtual permite que cada proceso tenga su propio espacio de direcciones (con excepción de los hilos, que comparten su espacio de direcciones). Por ejemplo, en la arquitectura x86 de 32 bits, cada proceso tiene un espacio de direcciones de 4GB de memoria virtual.

En Linux, el espacio de direcciones de cada proceso se ha limitado a los primeros 3GB. Las tablas de páginas en éste rango son privadas a cada proceso. Las tablas de páginas del último GB son compartidas por todos los procesos y representan las Tablas Maestras de Páginas del Núcleo (como se mencionó en la Sección 3.1.5) y requieren permisos privilegiados (ejecución en modo núcleo).

3.1.9. Administración del Espacio de Direcciones de Proceso en Linux

Cuando un proceso se carga en memoria, se cargan también las librerías dinámicas que utiliza. Cada librería utilizada, al igual que el programa, tiene sus propias secciones de código, de datos y de pila. Estas secciones pueden estar mapeadas en cualquier zona del espacio de direcciones del proceso. Linux asigna a cada una de estas secciones una zona diferente en el espacio de direcciones del proceso, pues cada una de ellas requiere de permisos diferentes.

Linux representa cada zona con una estructura `vm_area_struct` cuyos campos más importantes se describen a continuación (la descripción completa puede verse en el apéndice C):

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;

    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;
    unsigned long vm_flags;
```

```
    struct rb_node vm_rb;

    /* Entre otros campos ... */
};
```

`vm_area_struct→vm_mm` Indica cuál es el espacio de direcciones al que pertenece esta zona (la estructura `mm_struct` se describe más adelante).

`vm_area_struct→vm_start` Primera dirección dentro de la zona.

`vm_area_struct→vm_end` Siguiente dirección fuera de la zona (`vm_end` − `vm_start` es el tamaño de la zona).

`vm_area_struct→vm_next` Apuntador al siguiente descriptor de zona en este espacio de direcciones. Todas las `vm_area_struct` que representan un espacio de direcciones están ligadas a través de éste campo y en orden ascendente respecto a `vm_start`.

`vm_area_struct→vm_page_prot` Los permisos asignados a todas las direcciones dentro de ésta zona (lectura, escritura, ejecución).

`vm_area_struct→vm_flags` Conjunto de banderas que describen las propiedades de las direcciones dentro la zona (si se pueden compartir, si es usada como pila, entre otras).

`vm_area_struct→vm_rb` Además de que todas las `vm_area_struct` de un espacio de direcciones se encuentran organizadas en una lista ligada, se utiliza un árbol rojinegro (también conocidos como symmetric binary B-trees [5]) para su rápida localización. Este campo contiene los apuntadores necesarios.

La Figura 3.8 muestra cómo se usan estas estructuras para representar el espacio de direcciones de un proceso. Es importante señalar que los intervalos de memoria que representan cada una de estas estructuras no se traslapan, es decir, no existen intersecciones.

Para mantener organizadas estas zonas en el espacio de direcciones, Linux hace uso de una estructura de tipo `mm_struct`, almacenada en el descriptor de proceso (revisado en el Capítulo 2), la cual representa su espacio de direcciones. A continuación se muestran los campos mas importantes de la estructura y su descripción:

```
struct mm_struct {
    struct vm_area_struct * mmap;            /* lista de VMAs */
    struct rb_root mm_rb;
```

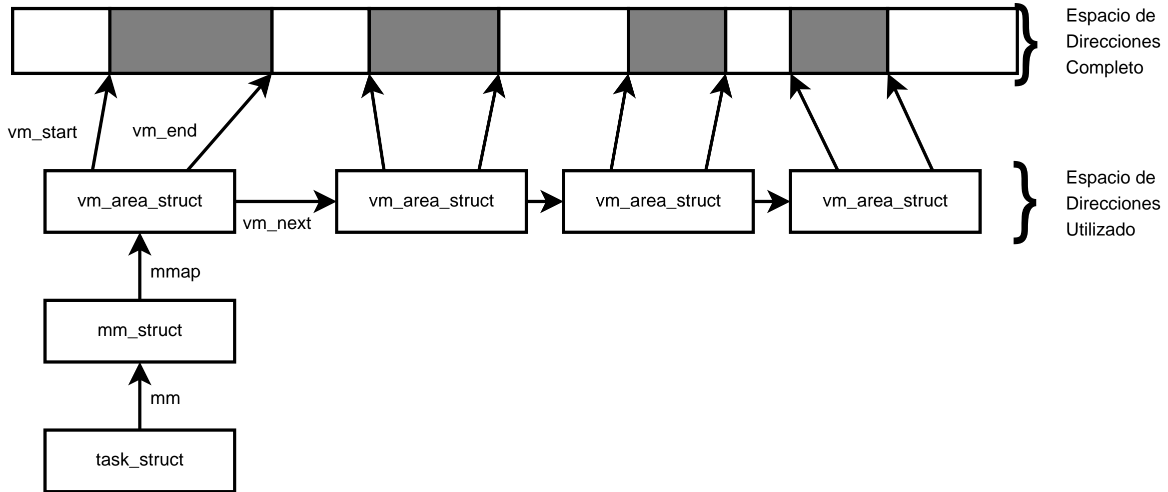

Figura 3.8.: Representación del espacio de direcciones de proceso

```
    pgd_t * pgd;
    int map_count;                              /* numero de VMAs */
    struct list_head mmlist;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;

    /* Entre otros campos ... */
};
```

`mm_struct→mmap` Apuntador a la primera estructura de tipo `vm_area_struct`, es decir, a la lista ligada de zonas de memoria utilizadas en este espacio de direcciones.

`mm_struct→mm_rb` Apuntador a la raíz del árbol rojinegro de las zonas de memoria utilizadas en este espacio de direcciones.

`mm_struct→pgd` Apuntador al Page Global Directory de este espacio de direcciones.

`mm_struct→map_count` Cantidad de estructuras `vm_area_struct` ligadas en `mmap`

`mm_struct→mm_list` Todas las estructuras `mm_struct` existentes en el sistema están ligadas a través de esta lista.

`mm_struct→start_code, end_code` Direcciones de inicio y de fin de la sección de código del proceso.

Tamaño	884 KB
Archivos .c	46
Archivos .S (ensamblador)	0
Archivos .h	11
Total de archivos de código	57
Porcentaje del total en Linux	0,39 %
Líneas de código en archivos .c	30,466
Líneas de código en archivos .S (ensamblador)	0
Líneas de código en archivos .h	2,717
Total de líneas de código	33,183
Porcentaje del total en Linux	0,55 %

Cuadro 3.1.: Estadísticas sobre la administración de memoria en Linux 2.6.10 (directorios `linux-2.6.10/mm/` y `linux-2.6.10/arch/i386/mm`).

`mm_struct→start_data, end_data` Direcciones de inicio y de fin de la sección de datos del proceso.

`mm_struct→start_brk, brk` Direcciones de inicio y de fin de la sección de area dinámica (también conocida como heap).

`mm_struct→start_stack` Dirección de inicio de la pila.

3.2. Simplificación de la Administración de Memoria en eLinux

El Cuadro 3.1 muestra algunos datos sobre la cantidad de código referente a la administración de memoria en `linux 2.6.10`. Podemos ver que la administración de memoria no abarca gran cantidad del código de Linux. La administración de memoria (al igual que la administración de procesos) es una parte central de Linux y no puede ser modificada de manera sustancial.

Para eLinux, hemos eliminado el soporte para NUMA y HIGHMEM, simplificando un poco la organización de la memoria física. De esta manera fue posible eliminar la lista de nodos `pgdat_list`, así como la zona `ZONE_HIGHMEM`. De esta manera, la organización de la memoria se reduce a lo mostrado en la Figura 3.9 en contraste con la Figura 3.1, mostrada anteriormente.

Al eliminar la zona de alta memoria `ZONE_HIGHMEM`, fue posible eliminar la variable

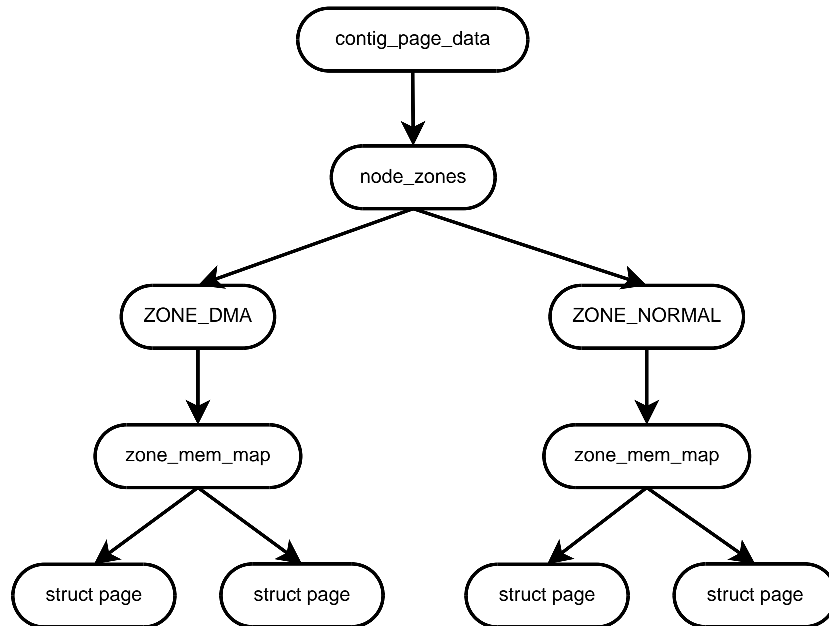

Figura 3.9.: Relaciones entre las estructuras principales en eLinux

max_pfn mencionada en la Sección 3.1.4 y sólo se hace uso de las variables min_low_pfn y max_low_pfn. Así, el mapa de la memoria física en eLinux, mientras se usa el Boot Memory Allocator, se muestra en la Figura 3.10.

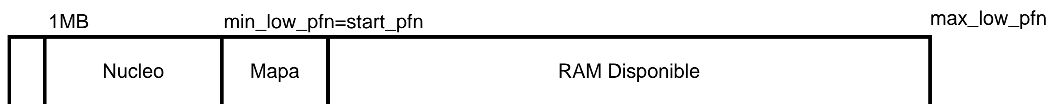

Figura 3.10.: Mapa de memoria en eLinux

El hecho de que la arquitectura x86 sólo utilice 2 niveles y que nuestro núcleo está enfocado a dicha arquitectura, nos llevó a eliminar el nivel intermedio para dejar sólo el nivel más alto (Directorio Global de Páginas, PGD) y el más bajo(PTE), sin necesidad de utilizar optimizaciones del compilador.

Con estas opciones removidas, la cantidad de código referente a la administración de memoria se redujo casi un 30 %. El Cuadro 3.2 muestra las mismas estadísticas del Cuadro 3.1, pero para el código en eLinux. En él podemos ver que la administración de memoria representa una mayor parte en el código de eLinux.

Tamaño	716 KB
Archivos .c	34
Archivos .S (ensamblador)	0
Archivos .h	9
Total de archivos de código	43
Porcentaje del total en eLinux	11,25 %
Líneas de código en archivos .c	22, 520
Líneas de código en archivos .S (ensamblador)	0
Líneas de código en archivos .h	1, 931
Total de líneas de código	24, 451
Porcentaje del total en eLinux	7,87 %

Cuadro 3.2.: Estadísticas sobre la administración de memoria en eLinux (directorios `elinux-2.6.10/mm/` y `elinux-2.6.10/arch/i386/mm`).

3.3. Tareas y Proyectos

1. Implementar una función que imprima en pantalla las zonas de memoria reservadas por un proceso. Puede implementarse como un módulo o como una llamada al sistema. Debe recibir como parámetro el `PID` del proceso. Esta función debe recorrer la lista de estructuras `vm_area_struct` ligadas a través del campo `task_struct`→`mm`→`mmap`.

Capítulo 4.

Administración de Entrada/Salida

Las primeras computadoras utilizaban tarjetas perforadas y teletipos como sus únicos dispositivos de entrada/salida. Posteriormente, las computadoras comenzaron a utilizar teclados, discos duros, y monitores para la interacción con el usuario. Hoy día, existe una gran cantidad de dispositivos que permiten interactuar con la computadora. Podemos encontrar una gran variedad de dispositivos de almacenamiento (CD's, DVD's, discos duros, discos flexibles, memorias USB), teclados, ratones, controles para videojuegos (palancas, volantes, etc.), dispositivos para red (tarjetas de red, módems, etc.), tarjetas de sonido, tarjetas de video, escáners, cámaras de video, cámaras fotográficas, impresoras, bocinas, micrófonos y muchos otros dispositivos especializados.

Cada dispositivo debe ser manejado por el sistema operativo de forma diferente. Por lo tanto, cada dispositivo debe tener su controlador.

En este capítulo, se analizará la manera en que Linux administra los distintos tipos de dispositivos y cómo se decidió simplificarlo.

4.1. Administración de la Entrada/Salida en Linux

4.1.1. Representación de dispositivos

En Linux casi todos los dispositivos se manejan como archivos. Es decir, en el sistema de archivos existen algunos archivos que representan dispositivos de hardware. De esta manera la misma interfaz para manipular archivos (funciones como `open()`, `read()`, `write()`, `close()`, `lseek()`) pueden usarse similarmente para manipular dispositivos. Normalmente, los archivos de dispositivos se encuentran en el directorio `/dev`. Por ejemplo, el archivo `/dev/hda` representa el primer disco duro del sistema.

Cada archivo de dispositivo tiene asociados 2 números: el número mayor y el número menor. Estos números son los que identifican al dispositivo. El nombre del archivo puede ser arbitrario, pero los números mayor y menor no pueden serlo. El número mayor identifica el tipo de dispositivo, por ejemplo, todos los archivos que representan discos

duros comparten el mismo número mayor. El número menor identifica al dispositivo entre los dispositivos de un mismo tipo; todos los archivos que representan discos duros tienen diferentes números menores.

Listado 1 Números mayores y menores de discos duros

```
$ ls -l /dev/hd*
brw-rw----  1 root disk   3,  0 Aug 23 04:53 /dev/hda
brw-rw----  1 root disk   3,  1 Aug 23 04:53 /dev/hda1
brw-rw----  1 root disk   3,  2 Aug 23 04:53 /dev/hda2
brw-rw----  1 root disk   3,  5 Aug 23 04:53 /dev/hda5
brw-rw----  1 root disk   3,  6 Aug 23 04:53 /dev/hda6
brw-rw----  1 root disk   3,  7 Aug 23 04:53 /dev/hda7
brw-rw----  1 root disk   3,  8 Aug 23 04:53 /dev/hda8
brw-rw----  1 root disk   3, 64 Aug 23 04:53 /dev/hdb
brw-rw----  1 root disk   3, 65 Aug 23 04:53 /dev/hdb1
brw-rw----  1 root disk   3, 66 Aug 23 04:53 /dev/hdb2
```

Como ejemplo, veamos el Listado 1. Podemos ver que existen 2 discos duros: `/dev/hda` y `/dev/hdb`. El número mayor de ambos es `3`, pero tienen diferentes números menores, `0` y `64`, respectivamente. El número menor es utilizado por el controlador de discos duros para diferenciar entre los dispositivos.

En el listado 1 también podemos ver las particiones de cada disco duro. Un disco duro puede tener hasta 63 particiones. Linux asocia un número menor al disco duro entero y un número menor para cada partición. Así, los números menores del `0` al `63` están reservadas para el primer disco duro y sus particiones. El primer número de este rango (`0`) está asociado al disco duro completo y los posteriores a la partición correspondiente. Nótese que todas las particiones de un disco duro comparten el mismo número mayor.

4.1.2. Tipos de dispositivos

Si observamos el primer atributo de los archivos del listado 1, observamos que tienen el carácter `b`, el cual indica que el dispositivo es de tipo bloque. En Linux hay dos tipos de dispositivos: de carácter y de bloque.

Los dispositivos de carácter son aquellos cuyos datos sólo son accesibles de forma serial, es decir, son flujos de datos. Ejemplos de dispositivos de carácter son los puertos seriales y el teclado. En este tipo de dispositivos siempre se lee el último dato generado,

no es posible leer datos por posiciones, es decir, no permiten el acceso aleatorio (es decir, no permiten el uso de la función `lseek()`).

Los dispositivos de bloque son aquellos que permiten la lectura de manera aleatoria a bloques de datos. Un ejemplo de estos dispositivos son los discos duros, en ellos se puede leer cualquier bloque del disco en cualquier momento (es decir, permiten el uso de la función `lseek()`).

Linux maneja de forma distinta los dispositivos de bloque y de carácter. Para manejar los dispositivos de carácter, dado que este tipo de dispositivos pueden ser muy diferentes, debe existir un controlador para cada uno de ellos, por ejemplo, uno para el teclado, uno para el puerto serial, etc.

Para los dispositivos de bloque, en cambio, existe una capa intermedia, entre los controladores y la interfaz al usuario, llamada la capa de Entrada/Salida para dispositivos de bloque (Block I/O Layer).

4.1.3. Capa de Entrada/Salida para Dispositivos de Bloque

Esta capa es necesaria porque los dispositivos de bloque son muy parecidos entre sí y su funcionamiento tiene gran impacto en el rendimiento del sistema. La lectura/escritura en discos duros es una de las operaciones que toma más tiempo en un sistema de cómputo, por lo tanto, su rendimiento es crucial. Todas las operaciones de Entrada/-Salida en dispositivos de bloque tienen que pasar a través de esta capa, pues esta capa se encarga de planificar las peticiones de Entrada/Salida de tal manera que se obtenga el mejor rendimiento del sistema.

Planificadores de Entrada/Salida

Una de las operaciones más lentas en una computadora es la búsqueda en discos. Si se enviaran las peticiones de lectura/escritura en el mismo orden en que se realizan, el rendimiento del sistema sería muy pobre. Por ejemplo, si dos procesos están leyendo diferentes archivos al mismo tiempo, los procesos generarían peticiones de lectura en zonas distintas del disco. Si estas zonas están muy lejanas, el brazo lector del disco tendría que hacer movimientos muy largos, pasando más tiempo en este movimiento que transfiriendo datos. Es por esto que un planificador de entrada/salida tiene 2 objetivos: minimizar la cantidad de búsquedas en disco y así maximizar el rendimiento global del disco ("global throughput").

Dos estrategias comunes para minimizar la cantidad de búsquedas en disco son: el ordenamiento y la fusión de las peticiones. El ordenamiento consiste en mantener todas las peticiones ordenadas en base al sector del disco a leer/escribir. De esta manera, los movimientos del brazo del disco pueden realizarse primero en una dirección hasta

llegar a la última petición posible en dicha dirección y después comenzar en la dirección opuesta, además de que los movimientos del brazo son muy cortos. Esto es parecido a los movimientos de los elevadores de los edificios, razón por la cual los planificadores de entrada/salida también son conocidos como elevadores.

La fusión de peticiones consiste en fundir 2 o más peticiones a sectores del disco adyacentes en una sola petición. Los discos duros actuales son muy eficientes transfiriendo varios sectores adyacentes en una sola operación; es decir, realizar 3 peticiones diferentes pero adyacentes, cada una de un solo sector, es mucho menos eficiente que realizar una sola petición de 3 sectores.

Todos los planificadores de entrada/salida realizan el ordenamiento y la fusión de peticiones para obtener un mejor rendimiento global. El favorecer el rendimiento global del disco, implica desfavorecer el rendimiento para ciertas aplicaciones. Es por esto que Linux ofrece 4 distintos planificadores de entrada/salida, los cuales realizan otras operaciones y que pueden ser adecuados para distintas cargas de trabajo. Los 4 planificadores que ofrece Linux son:

- **Planificador Nulo (No-op scheduler).** Realiza solo las 2 operaciones básicas: ordenamiento y fusión de peticiones.
- **Planificador con Plazos (Deadline scheduler).** Utiliza plazos suaves (no estrictos) en los que las peticiones son atendidas. Bueno para servidores de bases de datos.
- **Planificador Anticipatorio (Anticipatory scheduler).** Intenta "adivinar" las peticiones posteriores. Es buena opción en la mayoría de las cargas de trabajo.
- **Planificador CFQ (Complete Fairness Queueing).** Intenta ofrecer la misma tasa de transferencia a todas las aplicaciones. Es bueno para el uso en máquinas de escritorio.

Estructuras Básicas

Para administrar la entrada/salida en dispositivos de bloque, Linux utiliza 2 estructuras básicas: `buffer_head` y `bio`. A continuación describiremos estas estructuras de datos, mencionando sólo sus campos más importantes y sus relaciones con otras estructuras. La definición completa de todas las estructuras pueden verse en el apéndice D.

Buffer Heads Los discos tienen como unidad básica de transferencia al sector. La mayoría de los discos pueden transferir sectores contiguos de una manera muy eficiente.

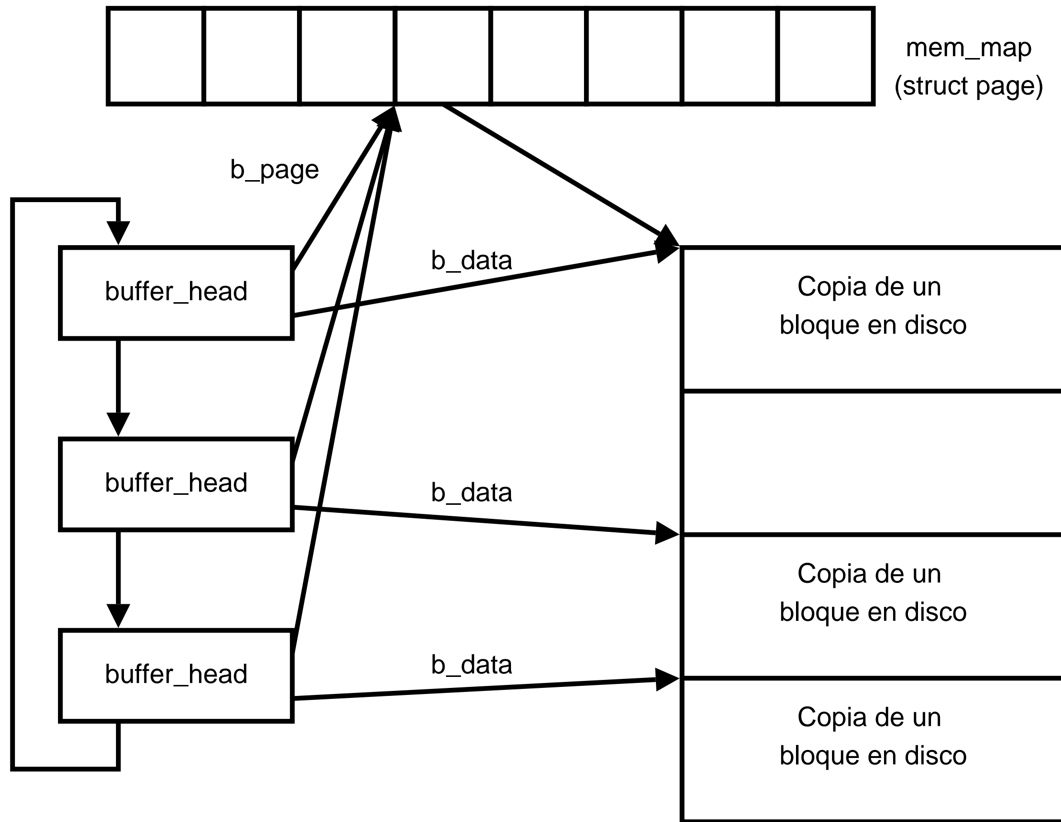

Figura 4.1.: Organización de la estructura `buffer_head`.

Aprovechando esto, se puede mejorar el rendimiento del disco agrupando sectores. Es por esto que todas las operaciones en discos se realizan tomando como unidad lógica el bloque. El tamaño de un bloque debe ser múltiplo del tamaño del sector del disco, pero también Linux requiere que sean múltiplos en potencias de 2. También se requiere que su tamaño no sea mayor al de la página, es decir, 4 KB. Por lo tanto, dado que los sectores más pequeños encontrados en los discos son de 512 bytes, los tamaños permitidos son: 512, 1024, 2048 y 4096 bytes.

En una operación de lectura se transfiere un bloque completo del disco a la memoria. Dado que dichas transferencias son muy costosas, es conveniente mantener en memoria los bloques transferidos la mayor cantidad de tiempo posible para evitar hacer transferencias innecesarias. Así, cuando se hace una lectura, se mantiene una copia del bloque en memoria, así como información referente a su localización en el disco y su estado. Si se requiere escribir sobre el bloque, la escritura se realiza en la copia y se marca su estado como "sucio" sin hacer la escritura en disco. Esta información se encuentra en

estructuras de tipo `buffer_head`, la cual está definida en <`linux/buffer_head.h`>.

Cada estructura `buffer_head`, para describir su bloque del dispositivo correspondiente, contiene los campos `buffer_head`→`b_bdev` y `buffer_head`→`b_blocknr`. El primero apunta a un descriptor del dispositivo de bloque al que pertenece el búfer; el segundo indica a cuál bloque del dispositivo corresponde el búfer.

Para conocer el estado del búfer (recién leído del disco, sucio, etc.) existe el campo `buffer_head`→`b_state`.

Para localizar los datos en el búfer (los datos copiados del disco), se tienen los campos `buffer_head`→`b_page`, el cual es un apuntador al descriptor de la página que contiene al búfer, y `buffer_head`→`b_data`, que es un apuntador al inicio de los datos.

Como los búfers pueden ser menores 4 KB (el tamaño de una página), puede haber varios búfers en una misma página. El campo `buffer_head`→`b_this_page` sirve para mantener una lista de todos los búfers contenidos en una misma página.

La Figura 4.1 muestra un conjunto de buffer heads cuyos datos se encuentran en la misma página.

Estructura bio La estructura `bio` es la unidad principal de entrada/salida en Linux. Cuando se va a realizar una transferencia de entrada/salida a disco, se utiliza una estructura `bio` para describir la operación. Cada estructura `bio` describe una región contigua del disco que debe ser leída/escrita cuyos contenidos serán copiados a/desde uno o más segmentos de memoria no necesariamente contiguos. En otras palabras, la estructura `bio` permite que una región contigua en disco sea mapeada a segmentos no contiguos en memoria.

Cada estructura `bio` contiene un apuntador (`bi_io_vec`) a un arreglo de estructuras de tipo `bio_vec`, cada una de las cuales representa un segmento en memoria que debe transferirse como parte de la transacción. Cada estructura `bio_vec` contiene campos para indicar la página donde se encuentran los datos, la posición dentro de la página y el tamaño de los datos.

El arreglo de estructuras `bio_vec` se va recorriendo al momento de realizar las transferencias. Conforme se van realizando cada una de las transferencias, el campo `bio`→`bi_idx` se va incrementando para indicar el índice en el arreglo `bi_io_vec` que será el siguiente segmento en transferirse. La Figura 4.2 muestra las relaciones entre estas estructuras.

4.1.4. Inicialización

La Figura 4.3 muestra las funciones relacionadas con la administración de entrada/salida que son ejecutadas por `start_kernel()`. En la Figura 4.3 se incluyen algunas

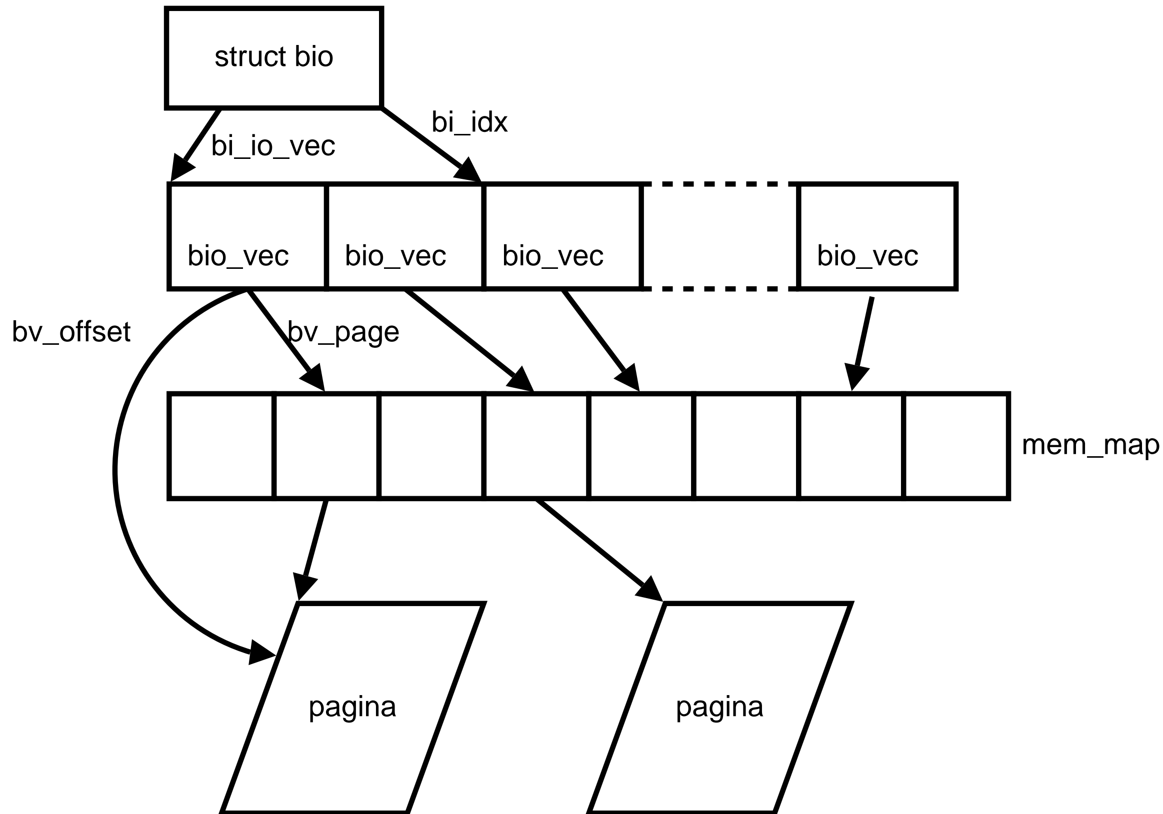

Figura 4.2.: Relaciones entre las estructuras `bio`, `bio_vec` y `page`.

funciones que no están relacionadas directamente con la administración de entrada/salida sólo para mostrar el contexto dentro del cual son ejecutadas.

La primera función relacionada directamente con la administración de entrada/salida que se ejecuta en Linux es `console_init()`, la cual se encarga de inicializar la terminal, de tal forma que sea utilizable para imprimir mensajes a través de ella.

Otra función ejecutada por `start_kernel()` es `buffer_init()`, la cual crea el `slab` (ver Sección 3) para las estructuras `buffer_head`.

4.2. Simplificación de la Administración de Entrada/Salida de Linux

Linux soporta una cantidad muy grande de dispositivos de entrada/salida. Bovet y Cesati [7] identificaron los dispositivos de hardware de un sistema de cómputo y los

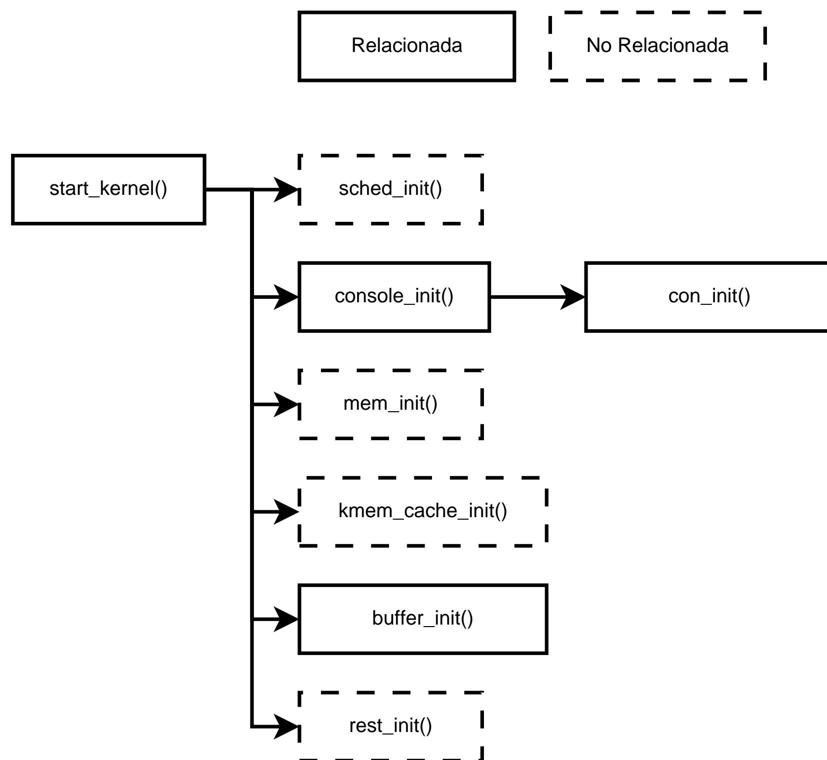

Figura 4.3.: Secuencia de inicialización en la administración de entrada/salida

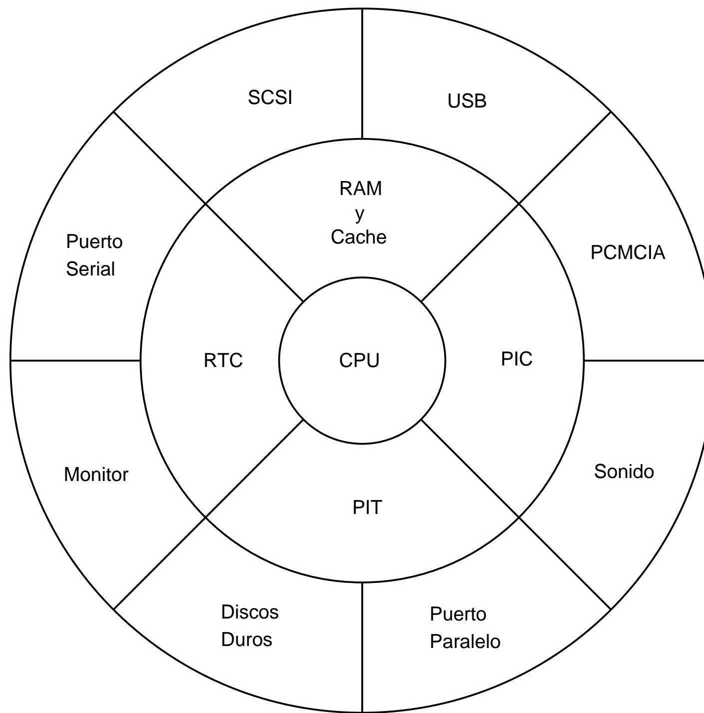

Figura 4.4.: Dispositivos de hardware de una computadora.

organizaron en capas según su importancia, como se muestra en la Figura 4.4 (es la Figura 1.2 repetida aquí por conveniencia).

En el código de Linux, se incluyen todos los controladores de dispositivos soportados por Linux. El Cuadro 4.1 muestra algunos datos sobre la cantidad de código referente a la administración de entrada/salida en `linux 2.6.10`. Podemos ver que los controladores de dispositivos constituyen una gran cantidad del código de Linux (48,31 % del total de líneas de código).

El propósito de un curso de Sistemas Operativos se debe centrar en la organización del sistema operativo y sus algoritmos, por lo que no es necesario mantener el código de tantos controladores de dispositivos.

Es por esto que, basándonos en la organización descrita por Bovet y Cesati [7], decidimos eliminar el soporte a una gran cantidad de dispositivos.

En la Figura 4.4, los dispositivos en el centro (el CPU) y en la primera capa son indispensables para el funcionamiento del sistema y el soporte de estos dispositivos no puede ser removido. En la capa más externa están los dispositivos que no son indispensables para el correcto funcionamiento del sistema y su utilidad depende del uso que se le esté dando al mismo. Así, los controladores de dispositivos ubicados en la última

Tamaño	98 MB
Archivos .c	2,481
Archivos .S (ensamblador)	7
Archivos .h	1,228
Total de archivos de código	3,716
Porcentaje del total en Linux	25,80 %
Líneas de código en archivos .c	2,486,458
Líneas de código en archivos .S (ensamblador)	3,813
Líneas de código en archivos .h	391,884
Total de líneas de código	2,882,155
Porcentaje del total en Linux	48,31 %
Cantidad de opciones de configuración	2,072
Porcentaje del total en Linux	40,92 %

Cuadro 4.1.: Estadísticas sobre los controladores de dispositivos en Linux 2.6.10 (directorio `linux-2.6.10/drivers/`).

capa de la Figura 4.4 fueron los candidatos a remover.

Se decidió mantener sólo el soporte para los dispositivos que son indispensables para tener un sistema mínimo funcional. Así, los dispositivos soportados en eLinux pueden verse en la Figura 4.5, donde se aprecia que la capa más externa es la única afectada y es drásticamente reducida.

El Cuadro 4.2 muestra de una manera más detallada los dispositivos cuyo soporte se ha mantenido en eLinux. En este cuadro, se muestran las opciones de configuración referentes a los controladores mantenidos en eLinux.

En eLinux se ha mantenido casi intacta la capa de entrada/salida para dispositivos de bloque (descrita en la Sección 4.1.3). La diferencia es que se decidió eliminar 3 de los 4 planificadores de entrada/salida (ver Sección 4.1.3). El planificador que se mantuvo es el no-op scheduler pues es el más sencillo, pero muestra las operaciones básicas que debe realizar cualquier planificador de entrada/salida: ordenamiento y fusión de las peticiones.

El Cuadro 4.3 muestra algunas estadísticas sobre el código referente a la administración de entrada/salida en eLinux.

4.3. Tareas y Proyectos

1. Estudiar el planificador con Plazos de Linux.

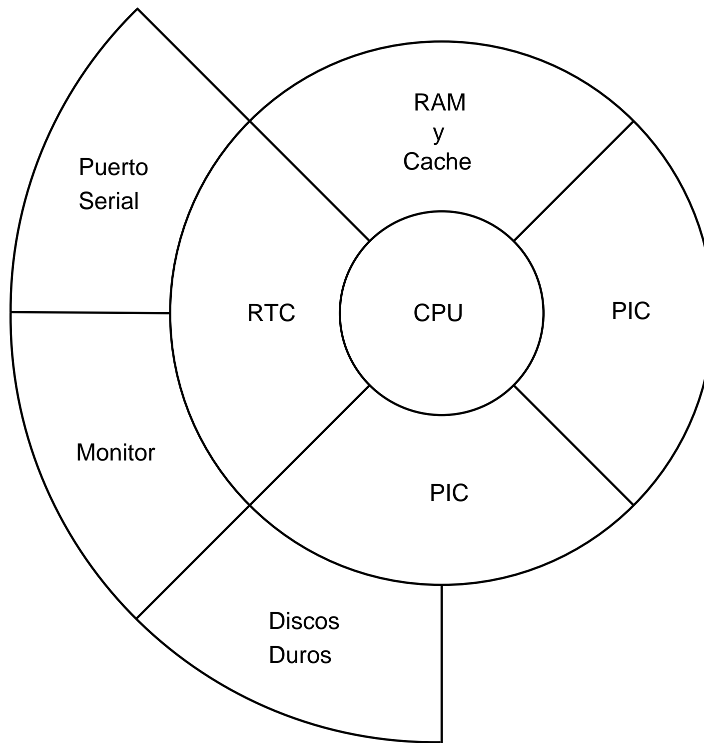

Figura 4.5.: Dispositivos de hardware soportados en eLinux.

Opción de configuración	Descripción
BLK_DEV_FD	Soporte para discos flexibles
IDE	Soporte para ATA/ATAPI
BLK_DEV_IDE	Soporte mejorado para discos y discos flexibles IDE
BLK_DEV_IDEDISK	Soporte para discos IDE/ATA-2
BLK_DEV_IDEFLOPPY	Soporte para discos flexibles IDE/ATAPI
IDE_GENERIC	Soporte genérico para el chipset IDE
INPUT_KEYBOARD	Soporte genérico para teclados
KEYBOARD_ATKBD	Soporte para teclados AT
SERIO	Soporte para puerto Serial
SERIO_I8042	Soporte para el controlador de teclados de PC i8042
SERIAL_8250	Soporte para puertos seriales 8250/16550
SERIAL_8250_CONSOLE	Soporte para consolas en puertos 8250/16550
VT	Soporte para terminales virtuales
VT_CONSOLE	Soporte para consolas en terminales virtuales
VGA_CONSOLE	Soporte para consolas en monitores VGA

Cuadro 4.2.: Dispositivos soportados por eLinux

El problema de hacer sólo las operaciones de ordenamiento y fusión es que si un proceso realiza muchas peticiones continuamente a una zona del disco y después otro proceso hace otras peticiones a una zona distante del disco, el segundo proceso podría estancarse ("starvation") y nunca ser atendido.

El planificador con plazos intenta evitar el estancamiento introduciendo 2 colas FIFO: una para lecturas y otra para escrituras. Cada vez que una petición de lectura o escritura es emitida, ésta se introduce en la cola FIFO respectiva, además de ser introducida en la cola ordenada por el bloque. Normalmente, el planificador toma de la cola ordenada la siguiente petición a atender, pero si un elemento en cualquiera de las 2 colas FIFO ha permanecido ahí por un periodo mayor a un plazo predeterminado, se elige esa petición en lugar de la petición en la cola ordenada. El plazo para la cola FIFO de lecturas es de 500 ms, mientras que el plazo para la cola FIFO de escritura es de 5 segs.

Esto evita totalmente el estancamiento.

2. Estudiar el planificador Anticipatorio de Linux.

   Este planificador es la implementación para Linux del *anticipatory scheduler* implementado para FreeBSD por Iyer y Druschel [17].

Tamaño	2 MB
Archivos .c	60
Archivos .S (ensamblador)	0
Archivos .h	8
Total de archivos de código	68
Porcentaje del total en eLinux	17,80 %
Líneas de código en archivos .c	58,687
Líneas de código en archivos .S (ensamblador)	0
Líneas de código en archivos .h	1,560
Total de líneas de código	60,247
Porcentaje del total en eLinux	19,40 %
Cantidad de opciones de configuración	51
Porcentaje del total en eLinux	39,80 %

Cuadro 4.3.: Estadísticas sobre los controladores de dispositivos en eLinux (directorio `elinux-2.6.10/drivers/`).

Además del ordenamiento y la fusión de peticiones, el planificador anticipatorio aprovecha que la mayoría de las aplicaciones leen o escriben archivos secuencialmente, es decir, bloques de archivos contiguos. Los sistemas de archivo también intentan mantener los archivos de manera contigua en disco. Así, es predecible que, cuando una aplicación hace una petición de lectura a un bloque en disco, la siguiente lectura será a un bloque contiguo al anterior.

Lo que hace el planificador anticipatorio es, después de cada petición, esperar por un pequeño periodo de tiempo sin atender las peticiones pendientes, con la idea de que la aplicación a la cual se le acaba de atender una petición emita otra petición, la cual muy posiblemente sea contigua o muy cercana a la anterior.

La implementación para Linux se realizó sobre el planificador con plazos, es decir, funciona de la misma manera, pero se le introduce la espera después de cada petición. El periodo de espera en Linux es de aproximadamente 6 ms.

3. Estudiar el planificador CFQ de Linux.

   Este planificador toma como base al Stochastic Fairness Queuing ([25] y [26]), el cual es utilizado para ofrecer calidad de servicio en la transmisión de paquetes en redes. Este planificador intenta ofrecer la misma cantidad de operaciones de entrada/salida a todos los procesos.

   El concepto principal es mantener una cierta cantidad de colas en las cuales se

insertan las peticiones. La elección de la cola en la cual se insertará la petición se realiza a través de una función hash de algún identificador del proceso (normalmente el TGID).

Las peticiones se atienden haciendo un Round Robin en las colas.

4. Jake Moilanen [28] está trabajando en introducir un algoritmo genético que ajuste los parámetros de los planificadores anticipatorio y con plazos. Estudiar su implementación.

Sería interesante modificar los parámetros de los planificadores y medir sus tiempos de respuesta para diferentes cargas de trabajo. Para probar el rendimiento de los planificadores, comúnmente se utilizan las siguientes pruebas (benchmarks):

- contest
- tiobench
- nickbench
- Oracle Simulator

Capítulo 5.

Administración de Archivos

Un sistema de archivos es una de las abstracciones más útiles que ofrece un sistema operativo. Los dispositivos de almacenamiento están organizados físicamente como un arreglo de bloques accesibles de manera arbitraria. Un sistema de archivos organiza estos bloques para que los usuarios puedan acceder de una manera sencilla y eficiente a sus datos.

La unidad básica de un sistema de archivos es el archivo. Para los usuarios, un archivo es una secuencia de bytes; para el sistema operativo, es un conjunto ordenado de bloques de bytes. Para hacer mejor uso del disco, los datos de un archivo no se almacenan, necesariamente, de manera secuencial en el disco. Un archivo es dividido en bloques que son almacenados en el disco en bloques determinados por el sistema de archivos. Así, un archivo puede ser visto como una lista de bloques pertenecientes a él. El sistema de archivos se encarga de manejar estas listas y ofrecer a los usuarios el acceso secuencial a sus archivos.

5.1. Sistemas de Archivos en Linux

Una de las razones de la gran aceptación de Linux es que, dada la filosofía abierta de Linux, sus desarrolladores siempre han tenido en mente la interoperabilidad de Linux con otros sistemas operativos. Una parte muy importante de los sistemas operativos son los sistemas de archivos y es necesario, para la interoperabilidad con otros sistemas, que Linux soporte una gran cantidad de sistemas de archivos.

Linux soporta sistemas de archivos tipo Unix (ver Sección 5.1.1), sistemas de archivos de otros sistemas operativos, sistemas de archivos en red y sistemas de archivos virtuales.

5.1.1. Los sistemas de archivos tipo Unix

Los sistemas de archivos tipo Unix están organizados con las siguientes estructuras básicas:

- archivos
- directorios
- nodos-i (inodes)
- superbloque

Los archivos son secuencias de bytes que pueden contener cualquier tipo de información.

En un sistema de archivos tipo Unix, un directorio es también un archivo cuyo contenido es la lista de archivos y subdirectorios pertenecientes a dicho directorio.

Tanto los archivos como los directorios tienen asociado a cada uno de ellos su meta-información. Esta meta-información es almacenada en los nodos-i. La meta-información contenida en los nodos-i incluye: permisos de acceso, tamaño, propietario, grupo, tipo, fecha de último acceso, fecha de última modificación, entre otros datos. Los nodos-i son típicamente almacenados en bloques del disco duro especialmente reservados para este propósito. Los nodos-i también contienen la información sobre dónde están localizados los datos del archivo que representan.

El superbloque almacena información sobre el sistema de archivos como un todo. La información que típicamente se encuentra en el superbloque es el tamaño del sistema de archivos, espacio utilizado, espacio libre, tamaño de bloque utilizado, localización de los nodos-i, etc.

En los sistemas de archivos tipo Unix, su estructura en disco está determinada por las anteriores estructuras. Comúnmente, un disco que contiene un sistema de archivos tipo Unix está dividido de una manera similar a la Figura 5.1. Al inicio del disco está el superbloque; los nodos-i se encuentran en una o varias zonas de bloques reservados para ellos; y los archivos se encuentran dispersos en la zona de bloques de datos.

Figura 5.1.: Sistema de archivos tipo Unix

5.1.2. Sistemas de Archivos Soportados por Linux

Sistemas de archivos tradicionales

Estos son algunos de los sistemas de archivos tradicionales que soporta Linux con una breve descripción.

- **Ext2 (Second extended file system).** Es el sistema de archivos nativo de Linux.
- **Minix.** Es el sistema de archivos del sistema operativo Minix. Este sistema de archivos fue utilizado en las primeras versiones de Linux.
- **ROMFS.** Es un sistema de archivos muy sencillo, de solo lectura, desarrollado con la intención de ser utilizado en instalaciones mínimas de Linux (discos de arranque, routers, etc.)
- **ISO9660 (CDROM file system).** Es el sistema de archivos utilizado en los CD-ROM's. Incluye soporte para las extensiones Rock-Ridge (para incluir información de sistemas de archivos tipo Unix como permisos, propietario, etc.) y Joliet (permite eliminar algunas restricciones en los nombres de archivos impuestas por ISO9660).
- **UDF (Universal Disk Format).** Es un sistema de archivos utilizado recientemente en CD-ROM's y DVD's.
- **VFAT (Windows-95 file system).** Es el sistema de archivos utilizado por los sistemas operativos Microsoft Windows 95/98/Me/NT.
- **UMSDOS.** Permite utilizar un sistema de archivos de MS-DOS y, creando sobre él las extensiones necesarias, simular un sistema de archivos tipo Unix.
- **NTFS.** Es el sistema de archivos utilizado por los sistemas operativos Microsoft Windows NT/2000/XP/2003.
- **ADFS (Acorn Disc Filing System).** Sistema de archivos del sistema operativo RiscOS.
- **AFFS (Amiga Fast File System).** Sistema de archivos del sistema operativo AmigaOS.
- **HFS (Apple Macintosh file system).** Sistema de archivos del sistema operativo MacOS (antes de la versión 8.0).

- **HFSPLUS (Apple Extended HFS file system).** Sistema de archivos del sistema operativo MacOS (desde la versión 8.0).
- **BeFS (BeOS file system).** Sistema de archivos del sistema operativo BeOS.
- **BFS (Boot File System).** Sistema de archivos del sistema operativo SCO UnixWare. Utilizado por su cargador de arranque.
- **EFS.** Sistema de archivos utilizado por el sistema operativo IRIX (antes de la versión 6.0).
- **CRAMFS (Compressed ROM file system).** Es un sistema de archivos comprimido. Su intención es ser utilizado en sistemas "embebidos".
- **VXFS (FreeVxFS file system).** Sistema de archivos del sistema operativo SCO UnixWare.
- **HPFS (OS/2 HPFS file system).** Sistema de archivos del sistema operativo OS/2.
- **QNX4FS (QNX4 file system).** Sistema de archivos de los sistemas operativos de tiempo real QNX 4 y QNX 6.
- **SYSV (System V/Xenix/V7/Coherent file system).** Sistema de archivos de los sistemas operativos SCO, Xenix y Coherent.
- **UFS.** Sistema de archivos utilizados por los sistemas operativos SunOS, FreeBSD, NetBSD, OpenBSD y NeXTstep.

Sistemas de archivos con bitácora

Los siguientes sistemas de archivos incluyen el uso de bitácora (journals) para asegurar la integridad de los datos. La bitácora es un registro de las últimas transacciones realizadas en el sistema de archivos. El mecanismo de bitácora es muy similar a los mecanismos de transacciones utilizados en Bases de Datos. En caso de que ocurriese alguna falla en el sistema que no haya permitido desmontar el sistema de archivos, la bitácora permite revisar las últimas operaciones y revertirlas para regresar al último estado consistente del sistema de archivos. Este tipo de sistemas de archivos son los más utilizados actualmente para almacenar archivos.

- **Ext3 (Second extended file system with journaling support).** Es similar al sistema de archivos Ext2, pero incluye soporte para bitácora.

- **ReiserFS.** Es un sistema de archivos de alto rendimiento desarrollado por Hans Reiser. Utiliza árboles B como su estructura básica de organización.
- **JFS (IBM's Journaled Filesystem).** Sistema de archivos de alto rendimiento utilizado por IBM en sus sistemas operativos AIX y OS/2. Utiliza árboles B como su estructura básica de organización.
- **XFS (SGI XFS Filesystem).** Sistema de archivos de alto rendimiento utilizado por SGI en sus sistema operativos IRIX. Utiliza árboles B como su estructura básica de organización. Incluye una extensión para ser utilizado en sistemas de tiempo real.
- **JFFS (Journaling Flash File System).** Desarrollado por Axis Communications para ser utilizado en sistemas "embebidos" sin disco. Solo puede ser utilizado en chips tipo flash.
- **JFFS2 (Journaling Flash File System v2).** Segunda generación de JFFS desarrollada por Redhat Inc. [38]. Las mejoras sobre JFFS incluyen recolección de basura, compresión y soporte de ligas duras.

Sistemas de archivos en red

Los siguientes sistemas de archivos permiten el acceso a sistemas de archivos remotos. Todos ellos permiten montar el sistema de archivos remoto en la jerarquía de directorios local, dando la impresión de que los archivos se encuentran almacenados localmente.

- **NFS (Network File System).** Permite montar discos duros remotos en sistemas de archivos locales a través de TCP/IP.
- **SMB (Server Message Block).** SMB es el protocolo usado por los sistemas operativos Microsoft Windows para compartir archivos e impresoras.
- **CIFS (Common Internet File System).** Es el sucesor de SMB en sistemas operativos Microsoft Windows.
- **NCP (NetWare Core Protocol).** Permite montar discos duros remotos en sistemas de archivos locales a través de IPX.
- **CODA (Coda file system).** Similar a NFS pero con características avanzadas como: modo de operación desconectado, replicación, autenticación y encriptación.
- **AFS (Andrew File System).** Es un sistema de archivos distribuido desarrollado en la universidad de Carnegie Mellon.

Sistemas de archivos virtuales

Estos sistemas de archivos no utilizan algún disco duro u otro medio de almacenamiento secundario, los archivos son generados al momento de leerlos ó son almacenados en memoria RAM.

- **PROC (/proc file system).** Es un sistema de archivos que provee información acerca del estado del sistema. Este sistema de archivos es comúnmente montado en `/proc`. En él se puede encontrar información sobre los procesos que se estén ejecutando, sobre los dispositivos, sobre la memoria, etc.

- **SYSFS (sysfs file system).** Este sistema fue diseñado para permitir el intercambio de información entre los usuarios y el núcleo. A través de este sistema de archivos, el núcleo exporta información y parámetros que pueden ser leídos y modificados por el usuario, sobre los dispositivos del sistema.

- **TMPFS (Virtual memory file system).** Este sistema de archivos mantiene todos los archivos en memoria. Permite montar directorios que no requieren persistencia de datos al reiniciar el sistema. Es utilizado para montar directorios como `/dev` (utilizando `udev`), y `/dev/shm` (requerido por POSIX para compartir memoria a través de funciones como `shm_open()`).

5.2. La capa VFS

Como podemos ver, Linux soporta una gran cantidad de sistemas de archivos. Esto es posible gracias a la capa llamada VFS (Virtual File System). Todos los sistemas de archivos utilizan esta capa para definir sus funciones. Esta capa permite utilizar las mismas llamadas a sistema (`open()`, `read()`, `write()`, `close()`, etc.) para acceder diferentes sistemas de archivos. Esta capa se encarga de, por ejemplo, llamar la función `open()` del sistema de archivos correspondiente cuando un proceso ejecuta la llamada a sistema `open()`.

La capa VFS está diseñada para soportar sistemas de archivos tipo Unix (descritos en la Sección 5.1.1). A pesar de esto, los sistemas de archivos que no han sido diseñados de esta manera pueden ser soportados por Linux simulando las características no soportadas. Es decir, aún si un sistema de archivos no hace uso de los nodos-i, su implementación para Linux debe manejar las estructuras `inode`, simulando su existencia.

La Figura 5.2 muestra las principales estructuras de datos utilizadas por la capa VFS y sus relaciones. A continuación describimos estas estructuras, mencionando sólo sus

campos más importantes y sus relaciones con otras estructuras. La definición completa de todas las estructuras pueden verse en el Apéndice E.

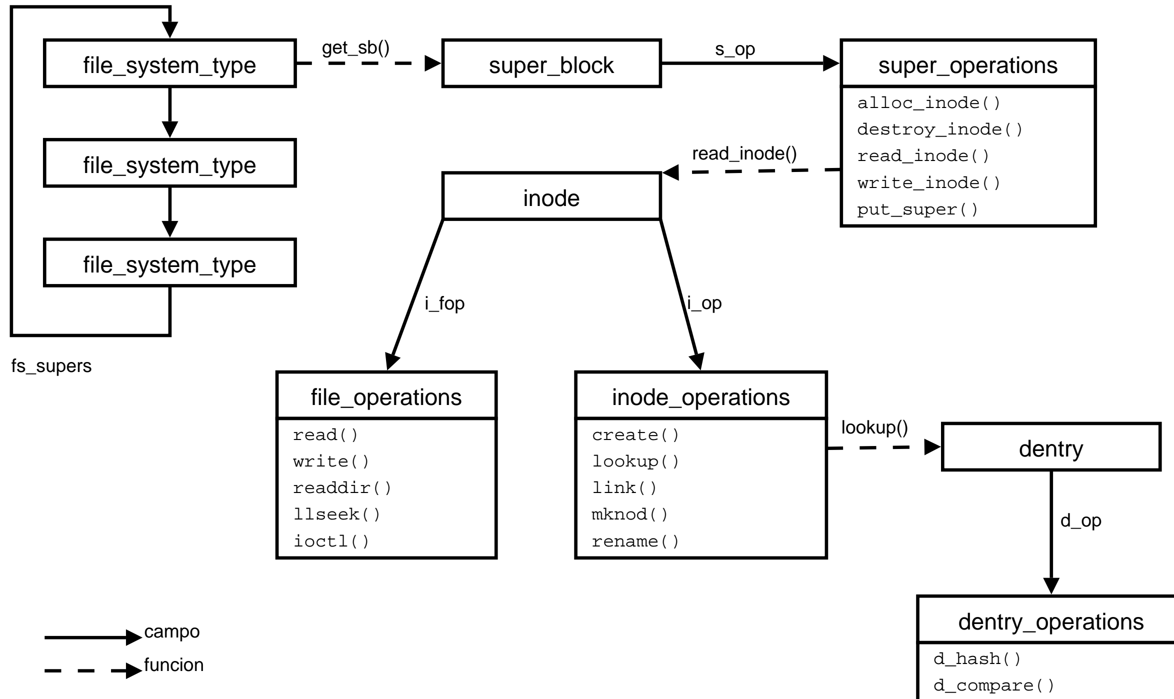

Figura 5.2.: La capa VFS.

La estructura `file_system_type` describe un sistema de archivos; incluye datos como su nombre, su tipo (virtual o físico) y un apuntador (`get_sb`) a una función específica del sistema de archivos cuya función es determinar si un dispositivo contiene un sistema de archivos de su tipo y, de ser así, debe llenar una estructura `super_block`.

La estructura `super_block` describe las características generales de un sistema de archivos como: tamaño de bloque utilizado, tamaño máximo de un archivo, modo de montaje (sólo lectura, escritura, etc.), entre otras. Esta estructura contiene sólo los datos que la capa VFS requiere para manejar el sistema de archivos. Cada sistema de archivos puede almacenar mucha más información en el superbloque físico, pero esa información es dependiente del sistema de archivos. El campo `super_block`→`s_fs_info` es utilizado para almacenar información específica del sistema de archivos; comúnmente es un apuntador a una estructura que representa el superbloque físico. El campo `super_block`→`s_op`, de tipo `super_operations`, contiene apuntadores a funciones utilizadas, principalmente, para manipular estructuras `inode`. Por ejemplo, la función referenciada por `super_operations`→`read_inode`, lee información sobre un nodo-i del

disco y llena una estructura `inode`.

La estructura `inode` representa la meta-información de un directorio o archivo. Esta meta-información incluye: número de nodo-i (único), permisos de acceso, propietario, grupo, tamaño, etc. Igualmente que otras estructuras de la capa VFS, esta estructura incluye sólo lo necesario para la capa VFS; los datos específicos del sistema de archivos pueden ser almacenados en el campo `inode→generic_ip`. El campo `inode→i_op` es un apuntador a una estructura `inode_operations` que contiene apuntadores a funciones relacionadas con la búsqueda, creación, eliminación y modificación de los atributos de archivos/directorios. El campo `inode→i_fop` es un apuntador a una estructura `file_operations` que contiene apuntadores a funciones relacionadas con la manipulación del contenido de los archivos/directorios, tales como su lectura y escritura.

Las estructuras de tipo `dentry` describen cada uno de los elementos de una ruta. Por ejemplo, en la ruta `/usr/bin/gcc`, cada uno de los elementos `/`, `usr`, `bin` y `gcc` es descrito por una estructura `dentry`. Esta estructura no tiene una estructura en disco asociada (como los nodos-i o el superbloque); es usada por el núcleo para ir "navegando" las rutas de los archivos/directorios, por lo que su existencia es temporal. La capa VFS descompone las rutas de archivos/directorios en todos sus componentes y, con ayuda de funciones como `inode_operations→lookup` define si la ruta es válida o no, para después proceder con otras operaciones como `file_operations→read` o `file_operations→write`.

Hay otras estructuras importantes en la capa VFS. Una de ellas es `vfsmount`. Cada estructura `vfsmount` representa un punto de montaje. Por cada sistema de archivos montado se crea una estructura de este tipo y se liga a través de campos como `vfsmount→mnt_parent` y `vfsmount→mnt_child`, respectivamente, al sistema de archivos donde se encuentra el directorio donde está montado y los sistemas de archivos que están montados en directorios de este sistema de archivos. De esta manera, se puede tener control sobre la jerarquía de los sistemas de archivos montados.

Otra estructura importante es `files_struct`, la cual contiene, entre otras cosas, un arreglo (`files_struct→fd_array`) con todos los archivos abiertos por un proceso. Este es un arreglo de estructuras `file`. Esta estructura contiene información como el identificador del usuario dueño del archivo, el grupo, el modo en que se abrió el archivo, la estructura `dentry` correspondiente al archivo y la posición actual en el archivo.

5.2.1. Inicialización

La Figura 5.3 muestra la secuencia de funciones relativas a la administración de archivos que son llamadas por `start_kernel()`.

La primera función llamada por `start_kernel()`, relativa a la administración de archivos, es `vfs_caches_init_early()` cuya función es crear las tablas hash para la

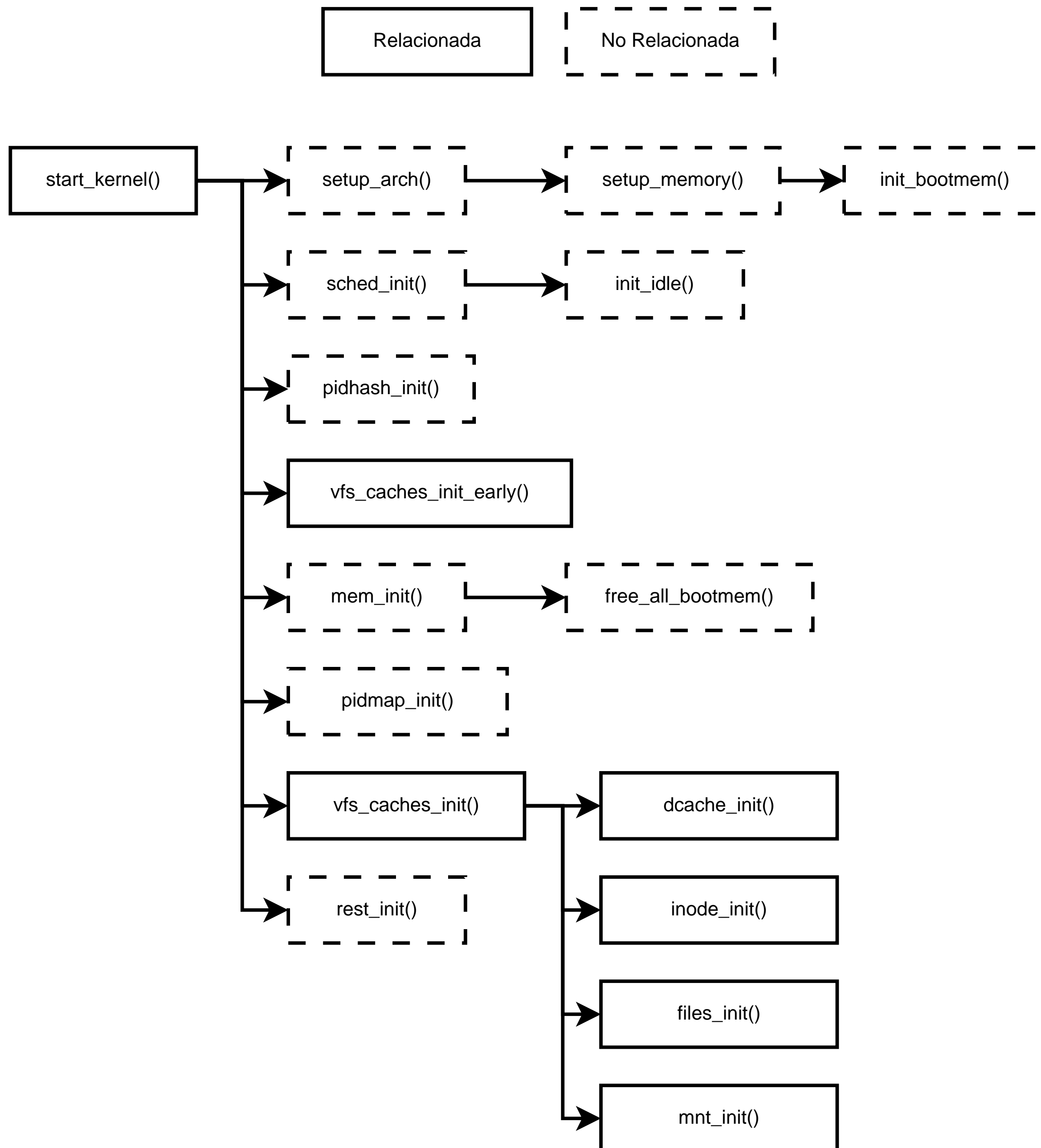

Figura 5.3.: Secuencia de Inicialización del Administración de Archivos

rápida localización de las estructuras `inode` y `dentry`. Estas estructuras son accesadas muy frecuentemente.

La siguiente función llamada por `start_kernel()` es `vfs_caches_init()`, la cual llama las funciones `dcache_init()`, `inode_init()`, `files_init()` y `mnt_init()`. Las funciones `dcache_init()` y `inode_init()` crean slabs (ver Capítulo 3) para las estructuras `dentry` y `inode`, respectivamente. `files_init()` calcula el máximo número de archivos abiertos que se pueden tener en memoria, de tal manera que las estructuras asociadas a un archivo abierto no sobrepasen del 10 % del total de la memoria RAM disponible. Finalmente, `mnt_init()` monta el sistema de archivos raíz.

5.3. Simplificación de la Administración de Archivos de Linux

Tanto la capa VFS como todos los sistemas de archivos soportados por Linux se encuentran dentro del código de Linux en el directorio `linux-2.6.10/fs`. El Cuadro 5.1 muestra algunos datos sobre la cantidad de código referente a la administración de archivos en `linux 2.6.10`.

Tamaño	19 MB
Archivos .c	653
Archivos .S (ensamblador)	0
Archivos .h	242
Total de archivos de código	895
Porcentaje del total en 2.6.10	6,21 %
Líneas de código en archivos .c	489, 004
Líneas de código en archivos .S (ensamblador)	0
Líneas de código en archivos .h	42, 509
Total de líneas de código	531, 513
Porcentaje del total en 2.6.10	8,91 %
Cantidad de opciones de configuración	196
Porcentaje del total en 2.6.10	3,87 %

Cuadro 5.1.: Estadísticas sobre los sistemas de archivos en Linux 2.6.10 (directorio `linux-2.6.10/fs/`).

La simplificación para eLinux consistió en remover el código de la mayoría de los sistemas de archivos. eLinux consiste sólo de los sistemas de archivos que lo hacen

confiable y fácil de usar, así como de la implementación para Linux (y eLinux) del sistema de archivos de XINU que permite a los estudiantes aprender el uso de la capa VFS y conocer de manera sencilla cómo se implementa un sistema de archivos (más detalles sobre la implementación en la Sección 5.4).

Así, eLinux consiste de los sistemas de archivos: `ext3`, `proc`, `sysfs`, y `xinufs`.

`proc` y `sysfs` son los sistemas de archivos fundamentales para el funcionamiento de Linux y no pueden ser removidos.

`ext3` fue elegido para soportar un sistema de archivos en el cual se almacene el sistema operativo y el software básico. Se eligió `ext3` por su soporte de bitácora. Cuando se realizan modificaciones al núcleo, es natural que se introduzcan errores que bloqueen el sistema y sea necesario reiniciarlo. Los sistemas de archivos sin bitácora requieren que, cuando se apaga de manera incorrecta el sistema, se realice una revisión al sistema de archivos (utilizando el programa `fsck`) al momento de reiniciarlo, proceso que es muy lento. Los sistemas de archivos con bitácora no requieren tal revisión, simplemente revisan su bitácora y restauran el sistema de archivos al último estado consistente. Es por esto que se decidió mantener el sistema de archivos `ext3`.

Otra opción que se removió es el soporte de cuotas (`CONFIG_QUOTA`). Las cuotas permiten limitar la cantidad de almacenamiento a la que cada usuario tiene acceso. Linux implementa las cuotas con una interfaz general que cada sistema de archivos puede implementar o no. En eLinux, se eliminó la interfaz general y la implementación de cuotas en `ext3`.

También se removió la estructura `export_operations` que es utilizada por NFS para comunicarse con los distintos sistemas de archivos. Esta estructura contiene apuntadores a funciones que NFS utiliza para transformar sus estructuras internas (conocidas como `file handles`) en estructuras `dentry`. Esta estructura no es necesaria en eLinux porque se ha eliminado el soporte de red y, por lo tanto, NFS.

También se removió el soporte para realizar "snapshots" del sistema de archivos, es decir, inhabilitar cambios al sistema de archivos. Esto es utilizado por LVM (Logical Volume Manager). LVM es un mecanismo que permite una administración más flexible de los discos duros y sus particiones; permite administrar los discos duros en volúmenes lógicos en lugar de particiones. Estos volúmenes lógicos pueden concatenarse, redimensionarse y removerse, incluso mientras están siendo utilizados.

Otra característica removida fue el soporte para el mecanismo dnotify (correspondiente a la opción de configuración `CONFIG_DNOTIFY`), el cual permite monitorear cambios al sistema de archivos. `dnotify` es usado por aplicaciones que requieren monitorear actividad en algunos directorios, comúnmente son aplicaciones de seguridad o aplicaciones de escritorio. Estas aplicaciones no son necesarias para desarrollar con el núcleo de Linux, por lo que no es necesario soportarlas en eLinux, además de que el código de `dnotify` es incluido en las llamadas principales de la capa VFS como `vfs_read()`,

`vfs_write()`, etc y afecta el seguimiento del código por parte de un estudiante.

El Cuadro 5.2 muestra algunas estadísticas sobre el código referente a la administración de archivos en eLinux.

Tamaño	1,8 MB
Archivos .c	77
Archivos .S (ensamblador)	0
Archivos .h	7
Total de archivos de código	84
Porcentaje del total en eLinux	9,56 %
Líneas de código en archivos .c	59, 135
Líneas de código en archivos .S (ensamblador)	0
Líneas de código en archivos .h	497
Total de líneas de código	59, 632
Porcentaje del total en eLinux	19,20 %
Cantidad de opciones de configuración	10
Porcentaje del total en eLinux	6,66 %

Cuadro 5.2.: Estadísticas sobre los sistemas de archivos en eLinux (directorio `eLinux-2.6.10/fs/`).

5.4. El Sistema de Archivos Xinu sobre eLinux

El sistema de archivos de Xinu fue desarrollado por Douglas Comer [10] para el sistema operativo Xinu. Su inclusión como parte de eLinux tiene como propósito mostrar cómo funciona un sistema de archivos y cómo se usa la capa VFS en un contexto sencillo.

Aquí analizaremos la implementación del sistema de archivos de Xinu para Linux. Esta implementación está basada en la implementación encontrada en Xinix [9], la cual tiene unas ligeras modificaciones sobre el original sistema de archivos de Xinu.

5.4.1. Estructura

El sistema de archivos Xinu divide el disco en 3 secciones.

- Bloque de directorio.
- Bloques de iblocks.

- Bloques de datos.

El bloque de directorio puede considerarse como el superbloque de un sistema de archivos tipo Unix (ver Sección 5.1.1), en los que el superbloque contiene información general sobre el sistema de archivos almacenado en el disco. Los iblocks pueden considerarse como los inodes, pues contienen información específica sobre un archivo en disco. Físicamente, las 3 secciones se pueden ver como en la Figura 5.4 y con más detalle en la Figura 5.5.

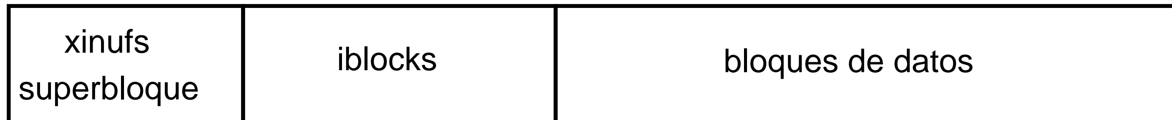

Figura 5.4.: Sistema de Archivos Xinu

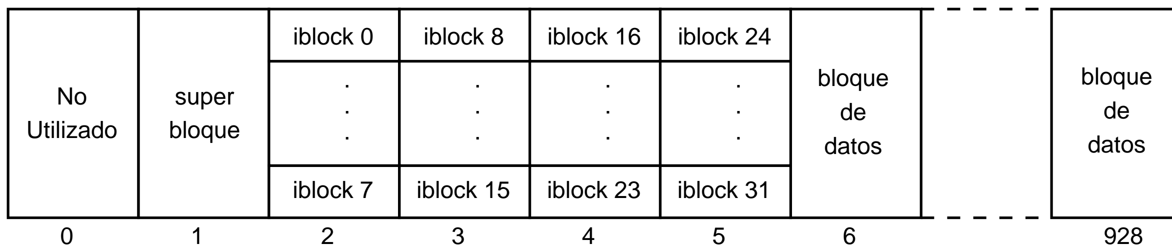

Figura 5.5.: Sistema de Archivos Xinu

El sistema de archivos de Xinu asume que el disco se encuentra dividido en bloques de 512 bytes. El primer bloque (bloque 0) no es utilizado por Xinu. Así que el superbloque es almacenado en el bloque 1 del disco. Ahora veamos la información contenida en el superbloque:

```
#define FDNLEN   10             /* Máxima longitud del
                                   nombre de archivo + 1 */
#define NFDES    28             /* Máximo número de archivos */

struct fdes  {                  /* Descripción de cada archivo */
    int32_t    fdlen;           /* Longitud en bytes */
    int16_t    fdiba;           /* Primer iblock */
    char       fdname[FDNLEN];  /* Nombre del archivo */
};
```

```
struct xinufs_super_block {
    int16_t     d_iblks;          /* Número de iblocks en el disco */
    u_int16_t   d_fblst;          /* Primer bloque de datos libre */
    u_int16_t   d_nfblks;         /* Número de bloques de datos libres */
    int16_t     d_filst;          /* Primer iblock libre */
    int16_t     d_id;             /* Número de identificaión del disco */
    int16_t     d_nfiles;         /* Número de archivos en el disco */
    char        d_xinuid[6];      /* Identificador de xinu */
    char        d_dummy[46];      /* Campo de relleno para
                                     llenar un bloque */
    struct fdes d_files[NFDES];   /* Descripción de los archivos */
};
```

Para comprender mejor los datos del superbloque, podemos apoyarnos en la Figura 5.6.

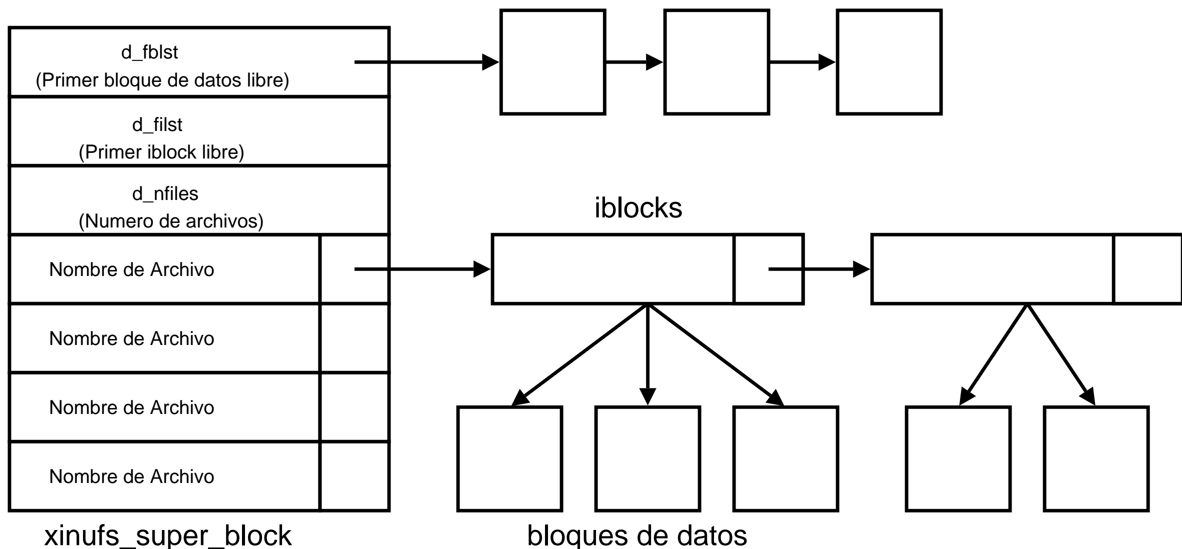

Figura 5.6.: Sistema de Archivos Xinu

La lista de bloques de datos libres se mantiene de la siguiente manera. El campo `xinufs_super_block→d_fblst` indica el número del primer bloque de datos libre. Los primeros 2 bytes de cada bloque de datos libre indican el número del siguiente bloque libre. De esta manera se tiene una lista ligada simple de bloques de datos libres. El último elemento en la lista apunta a un valor especial: `DBNULL` (`0xFFFF`). De la misma manera se mantiene la lista de iblocks libres, pero utilizando como cabecera de la lista al campo `xinufs_super_block→d_filst`.

Para cada archivo se mantiene una lista de iblocks utilizados por él. El propósito de cada iblock es mantener el registro de los bloques de datos utilizados por el archivo al que pertenecen.

Veamos el contenido de cada iblock:

```
#define IBLEN    29   /* Número de bloques de datos en cada iblock */
#define IBNULL   -1   /* Valor especial para el iblock NULL */
#define IBAREA    2   /* Indice del primer bloque de iblocks */

struct  xinufs_iblk {
    u_int32_t ib_byte;        /* Primer byte indexado por
                                 este iblock */
    int16_t   ib_next;        /* Índice del siguiente iblock */
    u_int16_t ib_dba[IBLEN];  /* Índices de los bloques de
                                 datos utilizados */
};

/* Regresa el bloque en disco donde está localizado el iblock ib */
#define ibtodb(ib)   (((ib) >> 3)+IBAREA)
```

Como podemos ver, cada iblock puede indexar hasta 29 bloques de datos. Si un archivo requiere más de 29 bloques de datos, se reserva otro iblock para el archivo, almacenando su índice en la variable `xinufs_iblk→ib_next`. El último elemento de ésta lista contiene el valor de `IBNULL`.

5.4.2. Implementación

Veamos cómo se implementó el sistema de archivos Xinu en Linux. Las funciones que un sistema de archivos para Linux debe realizar son:

- Registrarse ante la capa VFS (ver Sección 5.4.3).
- Asociar funciones para el manejo de los inodes (ver Sección 5.4.4).
- Asociar funciones para la lectura/escritura de directorios (ver Sección 5.4.5).
- Asociar funciones para la lectura/escritura de archivos (ver Sección 5.4.6).

5.4.3. Registro

Iniciaremos con el registro del sistema de archivos con el núcleo de Linux. Primero se inicializa una estructura `file_system_type`:

```
static struct file_system_type xinufs_fs_type = {
    .owner      = THIS_MODULE,
    .name       = "xinufs",
    .fs_flags   = FS_REQUIRES_DEV,
    .get_sb     = xinufs_get_sb,
    .kill_sb    = kill_block_super,
};
```

El campo `file_system_type→name` contiene el nombre del sistema de archivos. En nuestro sistema de archivos, requerimos que el campo `file_system_type→fs_flags` contenga el valor `FS_REQUIRES_DEV`, lo cual significa que nuestro sistema de archivos requiere la existencia de un dispositivo físico (es decir, no es un sistema basado en RAM como `procfs` o `sysfs`). El campo `file_system_type→kill_sb` lo definimos como `kill_block_super`, el cual es el nombre de una función genérica de la capa VFS que se llama al desmontar un sistema de archivos previamente montado. El campo `file_system_type→get_sb` debe contener una función específica del sistema de archivos que sea capaz de reconocer si un disco contiene o no el formato apropiado. En nuestro caso, la función encargada de esto es `xinufs_get_sb()` y es llamada cada vez que se monta un sistema de archivos de tipo `xinufs`.

Para registrar el sistema de archivos se hace uso de la función `register_filesystem()`, la cual es una función de la capa VFS y recibe como argumento una estructura del tipo `file_system_type`. La función `init_xinufs_fs()` es la encargada de registrar el sistema de archivos ante Linux.

```
static int __init init_xinufs_fs(void)
{
    int err = init_inodecache();
    if (err)
        goto out;
    err = register_filesystem(&xinufs_fs_type);
    if (err)
        goto clean;
    return 0;
 clean:
    destroy_inodecache();
 out:
    return err;
}
```

Esta función, al contener el atributo `__init`, es ejecutada por Linux durante la inicialización del núcleo (si es compilado como parte del núcleo) o al momento de cargar el módulo (si es compilado como módulo). `init_xinufs_fs()` inicializa una caché de

inodes (a través del Slab Allocator) y registra el sistema de archivos con ayuda de la función `register_filesystem()`.

Cuando se monta un sistema de archivos Xinu se ejecuta la función `xinufs_get_sb()`:

```
struct super_block *xinufs_get_sb(struct file_system_type *fs_type,
                                  int flags, const char *dev_name,
                                  void *data)
{
    return get_sb_bdev(fs_type, flags, dev_name, data,
                       xinufs_fill_super);
}
```

Como podemos ver, sólo llama a `get_sb_bdev()`, la cual es una función genérica de la capa VFS, enviándole como argumentos algunos de los que recibió y el nombre de una función que `get_sb_bdev()` llamará y que se debe encargar de leer el superbloque del disco y llenar una estructura `super_block` con los datos correctos. La función `xinufs_fill_super()` es la encargada de esto y la definimos como:

```
1  static int xinufs_fill_super(struct super_block *s, void *data,
2                               int silent)
3  {
4      struct buffer_head *bh;
5      struct xinufs_super_block *xsb;
6      struct xinufs_sb_info *sbi;
7      struct inode *root;
8
9      sbi = kmalloc(sizeof(struct xinufs_sb_info), GFP_KERNEL);
10     if (!sbi)
11         return -ENOMEM;
12     s->s_fs_info = sbi;
13     memset(sbi, 0, sizeof(struct xinufs_sb_info));
14
15     if (!sb_set_blocksize(s, XINUFS_BSIZE))
16         goto out_bad_hblock;
17
18     /* Read superblock from disk */
19     bh = sb_bread(s, XINUFS_SB);
20     if (!bh)
21         goto out_bad_sb;
22
23     xsb = (struct xinufs_super_block *)bh->b_data;
24     sbi->s_xsb = xsb;
25     sbi->s_sbh = bh;
```

```
26
27      xinufs_debug_superblock(xsb);
28
29      s->s_magic = le32_to_cpu(*(unsigned long *)(xsb->d_xinuid));
30      if (s->s_magic != (unsigned long)XINUFS_MAGIC)
31          goto out_no_fs;
32
33      /* set up enough so that it can read an inode */
34      s->s_op = &xinufs_ops;
35
36      /* get root inode */
37      root = iget(s, XINUFS_ROOT_INO);
38      if (!root || is_bad_inode(root))
39          goto out_no_root;
40
41      s->s_root = d_alloc_root(root);
42      if (!s->s_root)
43          goto out_iput;
44
45      return 0;
46  out_iput:
47      iput(root);
48      goto out_release;
49  out_no_root:
50      printk("XINUFS: get root inode failed\n");
51      goto out_release;
52  out_no_fs:
53      printk("VFS: Can't find a XINUFS filesystem on device "
54             "%s.\n", s->s_id);
55  out_release:
56      brelse(bh);
57      goto out;
58
59  out_bad_hblock:
60      printk("XINUFS: blocksize too small for device.\n");
61      goto out;
62
63  out_bad_sb:
64      printk("XINUFS: Unable to read superblock.\n");
65  out:
66      s->s_fs_info = NULL;
```

```
67        kfree(sbi);
68        return -EINVAL;
69 }
```

9-13: Reservamos memoria para una estructura de tipo `xinufs_sb_info`. Un apuntador a ésta estructura se almacena en el campo `super_block→s_fs_info` de la estructura `super_block` reservada para el sistema que se está montando. La estructura `xinufs_sb_info` está definida de la siguiente manera:

```
struct xinufs_sb_info
{
    struct buffer_head *s_sbh;
    struct xinufs_super_block *s_xsb;
};

struct xinufs_sb_info *xinufs_sbi(struct super_block *sb)
{
    return sb->s_fs_info;
}
```

Esta estructura almacena el `buffer_head` correspondiente al superbloque leído del disco, así como un apuntador al superbloque mismo. Un `buffer_head` representa un bloque leído del disco y contiene un apuntador a los datos leídos (ver Sección 4.1.3). Por esto, almacenamos estos datos en el campo `super_block→s_fs_info` para mantener siempre una referencia a ellos. La función `xinufs_sbi()` regresa la referencia a la estructura `xinufs_sb_info` contenida en el superbloque.

15-16: Definimos el tamaño del bloque deseado (`XINUFS_BSIZE` = `512` bytes).

19-25: Leemos el superbloque del disco. Para esto, usamos la función `sb_bread()`, a la cual se le pasa una estructura `super_block` y el número del bloque a leer. En este caso, le pasamos la macro `XINUFS_SB` cuyo valor es `1` (recordemos que el bloque `0` no lo utilizamos).

`sb_bread()` regresa un apuntador a un `buffer_head`, el cual lo almacenamos en nuestra estructura `xinufs_sb_info`. El campo `buffer_head→b_data` contiene un apuntador a los datos leídos del disco, por lo que le hacemos un cast a nuestra estructura de superbloque y lo almacenamos en nuestra estructura `xinufs_sb_info`.

29-31: El campo `xinufs_super_block→d_xinuid` de todos los sistemas de archivos Xinu contiene la cadena "xinud". En la implementación para Linux tratamos los primeros 4 bytes de dicho campo como un entero de 32 bits y lo comparamos con

XINUFS_MAGIC cuyo valor es 0x756E6978 (los valores ascii de la cadena "xinu" invertida, pues se debe considerar, al hacer el cast a entero, que la arquitectura i386 es little-endian). Si la comparación es incorrecta, se regresa error indicando que el sistema de archivos que se está montando no es tipo Xinu.

34: Se definen las operaciones que soporta nuestro sistema de archivos en base a la estructura super_operations:

```
static struct super_operations xinufs_ops = {
    .alloc_inode    = xinufs_alloc_inode,
    .destroy_inode  = xinufs_destroy_inode,
    .read_inode     = xinufs_read_inode,
    .write_inode    = xinufs_write_inode,
    .put_super      = xinufs_put_super,
};
```

37-43: Se lee el inode correspondiente al directorio raíz del sistema de archivos. Esto se hace con ayuda de la función iget() de la VFS, la cual llama a la función read_inode() de nuestro sistema de archivos. Una vez que se ha leído el inode raíz, se le asocia una estructura dentry y se almacena en el campo super_block→s_root del superbloque.

El resto de la función es el manejo de errores que se pueden presentar en distintas partes de la función. El uso de goto en Linux se utiliza, principalmente, para evitar repetición de código en el manejo de errores.

5.4.4. Manejo de archivos (inodes)

Ahora veamos cómo se realiza el manejo de los inodes. Para esto, definimos la estructura xinufs_inode_info y la función xinufs_i():

```
/* inode en memoria */
struct xinufs_inode_info
{
    struct xinufs_iblk iblk;
    struct fdes *fdes;          /* Apuntador al descriptor de
                                   archivo en el superbloque */
    struct inode vfs_inode;
};

struct xinufs_inode_info *xinufs_i(struct inode *inode)
{
```

```
    return container_of(inode, struct xinufs_inode_info, vfs_inode);
}
```

Como podemos ver, cada `xinufs_inode_info` contiene una copia del iblock en disco, un apuntador al descriptor de archivo correspondiente que se encuentra en el superbloque y una estructura `inode`.

La estructura `inode` es encapsulada dentro de nuestra estructura `xinufs_inode_info`. La capa VFS no sabe nada acerca de nuestra estructura y siempre maneja estructuras `inode`, por lo que es necesario definir la función `xinufs_i()` que, a partir de la estructura `inode`, regresa la estructura `xinufs_inode_info` que la contiene. `xinufs_i()` utiliza la macro `container_of()`, la cual es utilizada en varias partes de Linux y que, con ayuda del compilador, determina el offset del campo dentro de la estructura y retorna la estructura completa.

Como ya mencionamos anteriormente, la función `super_operations→read_inode` es llamada por la capa VFS cada vez que se requiere información sobre un archivo. Esta función la deben implementar todos los sistemas de archivos. Esta función debe recibir como parámetro una estructura `inode`. La capa VFS se encarga de que el campo `inode→i_ino` contenga el número de inode que se desea leer de disco.

Para el sistema de archivos Xinu, la función `xinufs_read_inode()` está definida de la siguiente manera:

```
1  static void xinufs_read_inode(struct inode *inode)
2  {
3      struct xinufs_sb_info *sbi;
4      struct xinufs_super_block *xsb;
5      struct xinufs_inode_info *xi;
6      struct buffer_head *bh;
7      struct xinufs_iblk *iblk;
8
9      sbi = xinufs_sbi(inode->i_sb);
10     xsb = sbi->s_xsb;
11
12     if(inode->i_ino == XINUFS_ROOT_INO ) {
13         /* Directorio: rw-r--r-- */
14         inode->i_mode = S_IFDIR |
15                         S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
16         inode->i_uid = current->fsuid;
17         inode->i_gid = current->fsgid;
18
19         inode->i_ctime = CURRENT_TIME;
20         inode->i_mtime = inode->i_atime = inode->i_ctime;
```

```
21
22            inode->i_nlink = 1;
23            inode->i_size = le16_to_cpu(xsb->d_nfiles);
24            inode->i_blksize = XINUFS_BSIZE;
25
26            inode->i_op = &xinufs_dir_inode_operations;
27            inode->i_fop = &xinufs_dir_operations;
28        } else {
29            xi = xinufs_i(inode);
30            iblk = xinufs_get_first_iblk(inode->i_sb,inode,&bh);
31            if( !iblk ) {
32                make_bad_inode(inode);
33                return;
34            }
35            xi->fdes = &xsb->d_files[inode->i_ino -1];
36            xi->iblk = *iblk;
37            brelse(bh);
38            /* Archivos: rw-r--r-- */
39            inode->i_mode = S_IFREG |
40                            S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
41            inode->i_uid = current->fsuid;
42            inode->i_gid = current->fsgid;
43
44            inode->i_ctime = CURRENT_TIME;
45            inode->i_mtime = inode->i_atime = inode->i_ctime;
46
47            inode->i_nlink = 1;
48            inode->i_size = le32_to_cpu(xi->fdes->fdlen);
49            inode->i_blksize = XINUFS_BSIZE;
50
51            inode->i_fop = &xinufs_file_operations;
52            inode->i_data.a_ops = &xinufs_aops;
53        }
54    }
```

9-10: Obtenemos un apuntador al superbloque de Xinu.

12-25 Si el número de inode a leer es el del directorio raíz (identificado con la macro `XINUFS_ROOT_INO`), le damos un tratamiento especial: en el campo `inode→i_mode` indicamos que es un directorio con los permisos `rw-r--r--`.

También definimos los campos `inode→i_uid` y `inode→i_gid` con los equivalentes del proceso que hace la lectura del inode, pues el sistema de archivos Xinu no almacena alguna información referente al usuario/grupo dueño del archivo. `current` es una variable global del núcleo que es un apuntador al proceso actual.

Los campos relativos al tiempo de creación, modificación y acceso al inode los inicializamos con la macro `CURRENT_TIME`. Esta macro regresa el tiempo actual del sistema, tomado de la variable global `xtime`.

El campo `inode→i_nlink` lo inicializamos a `1` pues Xinu no soporta enlaces. El campo `inode→i_size` de un inode correspondiente a un directorio debe ser el número de archivos que contiene, por lo que leemos del superbloque el campo `d_nfiles`.

Los campos `inode→i_op` y `inode→i_fop` se definen con los siguientes atributos:

```
static struct file_operations xinufs_dir_operations = {
    .read       = generic_read_dir,
    .readdir    = xinufs_readdir,
};

static struct inode_operations xinufs_dir_inode_operations = {
    .lookup     = xinufs_lookup,
    .create     = xinufs_create,
    .mknod      = xinufs_mknod,
    .link       = xinufs_link,
};
```

Mas adelante se mostrarán estas funciones.

27-48 Si el número de inode es diferente al del directorio raíz, entonces es el de un archivo, por lo que hay que leer datos del disco. Para esto hace uso de la función `xinufs_get_first_iblk()`, la cual lee del disco la estructura `xinufs_iblk` correspondiente al número de inode recibido como argumento.

El número del inode lo hemos definido como la posición del archivo en el arreglo `xinufs_super_block→d_files` más uno.

```
struct xinufs_iblk *xinufs_get_first_iblk(
                    struct super_block *sb,
                    struct inode *inode,
                    struct buffer_head **bh)
{
    struct xinufs_sb_info *sbi = xinufs_sbi(sb);
```

```
    struct xinufs_super_block *xsb = sbi->s_xsb;
    struct xinufs_iblk *iblk;
    ino_t ino = inode->i_ino;
    int iblock;

    if( !ino || ino > NFDES ) {
        printk("XINU-fs: Bad inode number on dev %s: "
                "%ld is out of range\n", sb->s_id, (long)ino);
        return NULL;
    }
    ino--;

    iblock = le16_to_cpu( xsb->d_files[ino].fdiba );
    iblk = xinufs_get_iblk(sb, iblock, bh);
    return iblk;
}
```

`xinufs_get_first_iblk()` obtiene el índice del iblock correspondiente al inode (leyendo esta información del superbloque residente en memoria) y llama a la función `xinufs_get_iblk()`, la cual regresa un apuntador a la estructura `xinufs_iblk` correspondiente al número de iblock solicitado.

```
struct xinufs_iblk *xinufs_get_iblk(struct super_block *sb,
                        int16_t iblock,
                        struct buffer_head **bh)
{
    struct xinufs_iblk *iblk;
    int block;

    block = ibtodb(iblock);
    *bh = sb_bread(sb, block);
    if (!*bh) {
        printk (KERN_CRIT "XINUFS: Unable to read iblock: %d "
                  "DB: %i.\n", iblock, block );
        return NULL;
    }
    iblk = (struct xinufs_iblk *)(*bh)->b_data + iblock;
    return iblk;
}
```

`xinufs_get_iblk()` determina el bloque en disco que contiene al iblock, lee dicho bloque del disco, localiza el iblock dentro del bloque leído y regresa un apuntador

al iblock.

Una vez que `xinufs_read_inode()` ha leído el iblock, requiere llenar una estructura `xinufs_inode_info` por cada inode, por lo que una vez que se lee una estructura `xinufs_iblk`, se guarda una copia de ésta en el campo `xinufs_inode_info→iblk` y un apuntador al descriptor de archivo `fdes` (residente en el superbloque).

En esta ocasión, el campo `inode→i_mode` del inode indica que es un archivo con los permisos `rw-r--r--`. El tamaño del archivo se lee del descriptor de archivo. Los otros campos del `inode` son inicializados de la misma manera que en el caso del directorio raíz con excepción de los campos `inode→i_fop` y `inode→i_data.a_ops`, los cuales se definen con los siguientes atributos:

```
static struct file_operations xinufs_file_operations = {
    .read           = generic_file_read,
    .write          = generic_file_write,
};

static struct address_space_operations xinufs_aops = {
    .readpage       = xinufs_readpage,
    .prepare_write  = xinufs_prepare_write,
    .commit_write   = generic_commit_write,
};
```

Mas adelante se mostrarán estas funciones.

5.4.5. Lectura y Escritura de Directorios

Como vimos anteriormente, las operaciones de archivo (estructura `file_operations`) del inode correspondiente al directorio raíz sólo definen 2 campos:

`read` Definida como `generic_read_dir()`, la cual es una función de la capa VFS y que sólo regresa el error: "Is a directory" pues un directorio no puede ser leído con la llamada al sistema `read()`.

`readdir` Definida como `xinufs_readdir()`:

```
static int xinufs_readdir(struct file *filp, void *dirent,
                          filldir_t filldir)
{
    char *name;
    unsigned len;
    int i, files;
```

```
        struct xinufs_sb_info *sbi = xinufs_sbi(filp->f_dentry->d_sb);
        struct xinufs_super_block *xsb = sbi->s_xsb;

        if( filp->f_pos > 0 )
            return 1;

        for(files=0, i=0;
            files<le16_to_cpu(xsb->d_nfiles) && i<NFDES;
            i++)
        {
            name = xsb->d_files[i].fdname;
            len = strlen(name);
            if ( len>0 ) {
                if( filldir(dirent, name , len, filp->f_pos++,
                             i+1, DT_REG ) )
                    return 1;
                files++;
            }
        }
        return 1;
    }
```

La cual lee del superbloque el arreglo `d_files` y va llenando un búfer del usuario con ayuda de la función `filldir()` de la VFS indicando, para cada archivo, el número del inode correspondiente.

Las operaciones de inode (estructura `inode_operations`) del directorio raíz definen las siguientes funciones:

`lookup` Definida como `xinufs_lookup()`:

```
    static struct dentry *xinufs_lookup(struct inode *dir,
                                        struct dentry *dentry,
                                        struct nameidata *nd)
    {
        struct inode *inode = NULL;
        ino_t ino;

        if (dir->i_ino != XINUFS_ROOT_INO)
            return ERR_PTR(-ENOENT);

        if (dentry->d_name.len > FDNLEN)
```

```
            return ERR_PTR(-ENAMETOOLONG);

        ino = xinufs_inode_by_name(dentry);
        if (ino) {
            inode = iget( dir->i_sb, ino );
            if (!inode)
                return ERR_PTR(-EACCES);
        }
        /* Ligar el inode a la dentry */
        d_add(dentry, inode);
        return NULL;
}
```

Esta función es llamada para determinar si un nombre de archivo (en el parámetro `dentry`) existe en directorio cuyo inode recibe como parámetro. Hace uso de la función `xinufs_inode_by_name()`, la cual busca en el directorio el nombre del archivo y, si existe, regresa su número de inode ó `0` en caso contrario. Una vez que se conoce el número de inode, usa la función `iget()` para obtener el inode del disco y se liga este `inode` con la estructura `dentry` correspondiente con la ayuda de la función `d_add()` de la capa VFS.

`create` Definida como `xinufs_create()`:

```
static int xinufs_create (struct inode * dir,
                          struct dentry * dentry,
                          int mode,
                          struct nameidata *nd)
{
    return xinufs_mknod(dir, dentry, mode, 0);
}
```

Esta función sólo llama a `xinufs_mknod()`, que es explicada a continuación.

`mknod` Definida como `xinufs_mknod()`:

```
static int xinufs_mknod (struct inode * dir,
                         struct dentry * dentry,
                         int mode,
                         dev_t rdev)
{
    struct inode * inode = xinufs_new_inode (dir, dentry, mode);
    int err = PTR_ERR(inode);
    if (!IS_ERR(inode)) {
```

```
            inode->i_fop = &xinufs_file_operations;
            inode->i_data.a_ops = &xinufs_aops;
            d_instantiate(dentry, inode);
            mark_inode_dirty(inode);
            return 0;
        }
        return err;
    }
```

Esta función, usa a la función `xinufs_new_inode()` para crear el nuevo archivo y, si no hubo error, define las operaciones correspondientes a los archivos y crea la estructura `dentry` correspondiente,

`link` Definida como `xinufs_link()`:

```
static int xinufs_link(struct dentry * old_dentry,
                       struct inode * dir,
                       struct dentry *dentry)
{
    return -EINVAL;
}
```

La cual sólo regresa el valor `-EINVAL` indicando que es una operación inválida, pues el sistema de archivos Xinu no soporta la creación de ligas.

5.4.6. Lectura y Escritura de Archivos

En el caso de los archivos, las operaciones de archivo (estructura `file_operations`) del inode correspondiente sólo definen 2 campos:

`read` Definida como `generic_file_read()`, la cual es una función genérica de la capa VFS y que hace uso de la función referenciada por el campo `readpage` (`xinufs_readpage()`) de la estructura `inode→i_data→a_ops`.

`xinufs_readpage()` sólo hace uso de la función genérica `block_read_full_page()` de la VFS, enviándole como argumentos el descriptor de la página y el nombre de la función `xinufs_get_block()`.

```
static int xinufs_readpage(struct file *file, struct page *page)
{
    return block_read_full_page(page, xinufs_get_block);
}
```

```
static inline int xinufs_get_block(struct inode * inode,
                   sector_t block,
                   struct buffer_head *bh_result,
                   int create)
{
    int realblock;

    xinufs_debug("Called.\n");
    xinufs_debug("block:%lu create:%d\n", block, create);

    if(create)
        realblock = xinufs_alloc_datablock(inode, block);
    else
        realblock = xinufs_find_datablock(inode, block);

    if( realblock < 0 )
        return realblock;

    map_bh( bh_result, inode->i_sb, realblock );
    return 0;
}
```

`xinufs_get_block()` debe traducir el número del bloque que se desea leer del archivo al número de bloque en disco que lo contiene. Es decir, si se requiere leer el segundo bloque de datos del archivo, debe transformarlo al número de bloque que lo contiene, referente al inicio del disco.

El parámetro `create` determina si el bloque pedido debe ser creado. Si el bloque debe ser creado, se llama a la función `xinufs_alloc_datablock()` (esta función será mostrada mas adelante). En caso contrario, se llama a la función `xinufs_find_datablock()`.

Una vez que se ha obtenido el bloque solicitado (sea nuevo o no), se utiliza la función `map_bh()`, la cual realiza un mapeo entre el `buffer_head` y el bloque en disco. De esta manera, si se requiere leer el mismo bloque, ya no es necesario ir a disco por él. Igualmente, si se escribe sobre el mismo buffer, se marca como sucio, y Linux, en algún momento, lo sincronizará al disco.

`xinufs_find_datablock()` está definida de la siguiente manera:

```
static inline sector_t xinufs_find_datablock(
    struct inode *inode, sector_t block)
```

```
{
    struct xinufs_iblk *iblk;
    struct buffer_head *bh = NULL;
    struct xinufs_inode_info *xinode = xinufs_i(inode);
    struct super_block *sb = inode->i_sb;

    if (!xinode)
        return -EIO;

    iblk = &xinode->iblk;

    while (block >= IBLEN) {
        if (bh)
            brelse(bh);
        iblk = xinufs_get_iblk(sb, le16_to_cpu(iblk->ib_next),
                                  &bh);
        if(!iblk) {
            printk (KERN_CRIT "XINU-fs: "
                    "Unable to read iblock\n");
            return -EIO;
        }
        block -= IBLEN;
    }
    if (bh)
        brelse(bh);
    return le16_to_cpu(iblk->ib_dba[block]);
}
```

Esta función recorre los iblocks del archivo (recordemos que cada iblock contiene la ubicación de sólo 29 bloques de datos, si un archivo tiene mas de 29 bloques de datos, utiliza otro iblock) y regresa la ubicación del bloque de datos solicitado.

`write` Definida como `generic_file_write()`, la cual es una función genérica de la capa VFS. Esta función hace uso de las funciones referenciadas por los campos `prepare_write` y `commit_write` (en el caso de Xinu, `xinufs_prepare_write()` y `generic_commit_write()`, respectivamente) que se encuentran definidas en el campo `inode→i_data→a_ops`.

`generic_file_write()` llama a `prepare_write()` antes de hacer una escritura para ligar una estructura `buffer_head` a un bloque del disco, reservándolo del disco si es necesario, copia los datos a escribir en el bloque y, posteriormente, llama a `commit_write()`, la cual hemos definido como `generic_commit_write()`

(función de la capa VFS), cuya función es marcar el `buffer_head` como sucio, para que Linux lo sincronice con el disco.

`xinufs_prepare_write()` está definida de la siguiente manera:

```
static int xinufs_prepare_write(struct file *file,
                                struct page *page,
                                unsigned from,
                                unsigned to)
{
    return block_prepare_write(page,from,to,xinufs_get_block);
}
```

Como vemos, `xinufs_prepare_write()` hace uso de otra función genérica de la capa VFS: `block_prepare_write()`. Esta función también hace uso de la función `xinufs_get_block()` para determinar el bloque del disco que se debe escribir. Ahora veamos la función `xinufs_alloc_datablock()`:

```
sector_t xinufs_alloc_datablock(struct inode *inode,
                                sector_t block)
{
    struct xinufs_sb_info *sbi;
    struct xinufs_super_block *xsb;
    struct xinufs_iblk *iblk;
    struct buffer_head *bh_iblk = NULL;
    struct buffer_head *bh_db = NULL;
    struct xinufs_inode_info *xinode = xinufs_i(inode);
    struct super_block *sb = inode->i_sb;
    struct xinufs_freeblk *freeblk;
    int freedb;

    if (!xinode)
        return -EIO;

    iblk = &xinode->iblk;

    /* Encontrar el iblock correspondiente*/
    while (block >= IBLEN) {
        if (bh_iblk)
            brelse(bh_iblk);
        iblk = xinufs_get_iblk(sb, le16_to_cpu(iblk->ib_next),
                               &bh_iblk);
        if(!iblk) {
```

```
                printk (KERN_CRIT "XINU-fs: "
                    "Unable to read iblock\n");
                return -EIO;
            }
            block -= IBLEN;
        }

        sbi = xinufs_sbi(sb);
        xsb = sbi->s_xsb;
        /* Tomamos el primer bloque de datos disponible */
        freedb = le16_to_cpu(xsb->d_fblst);
        if( freedb == DBNULL ) {
            printk (KERN_CRIT "XINU-fs: No space left on device");
            return -ENOSPC;
        }
        bh_db = sb_bread(sb, freedb);
        if (!bh_db) {
            printk (KERN_CRIT "XINU-fs: "
                "Unable to read datablock:%d\n", freedb);
            return -EIO;
        }
        /* Actualizamos la lista de bloques libres */
        freeblk = (struct xinufs_freeblk *)bh_db->b_data;
        xsb->d_fblst = freeblk->fbnext;

        /* Reservamos el bloque libre */
        iblk->ib_dba[block] = cpu_to_le16(freedb);

        if(bh_iblk)
            mark_buffer_dirty(bh_iblk);
        brelse(bh_db);

        mark_inode_dirty(inode);
        return freedb;
    }
```

Esta función localiza el iblock correspondiente al bloque que se quiere crear, toma un nuevo bloque de datos de la lista de bloques de libres del superbloque, actualiza la lista y asigna el bloque libre al iblock correspondiente.

5.5. Tareas y Proyectos

1. Implementar la eliminación de archivos para el sistema de archivos de Xinu para Linux. La implementación actual no permite el borrado de archivos. Para esto será necesario implementar las funciones `inode_operations→unlink` y `super_operations→delete_inode`. `unlink()` es llamada para borrar una entrada de directorio. `delete_inode()` es llamada cuando la entrada eliminada era la última liga a un inode. En el sistema de archivos Xinu, estas funciones se ejecutarán siempre de manera consecutiva pues no soporta el uso de ligas.

2. El sistema de archivos de Xinu no soporta la creación de directorios. Diseñar e implementar el manejo de subdirectorios.

3. El sistema de archivos de Xinu sólo puede manejar hasta 928 bloques de datos de 512 bytes. Diseñar e implementar el manejo de más bloques hasta donde la capacidad del disco alcance, modificando dinámicamente la estructura actual del sistema de archivos.

4. Optimizar el manejo de bloques libres del sistema de archivos. Manejar uno o varios bloques del disco donde se encuentre un mapa de bits que represente los bloques libres (con un bit `1`) y los bloques ocupados (con un bit `0`).

Capítulo 6.

Conclusiones

La robustez, confiabilidad, estabilidad, disponibilidad del código fuente y su presencia en el mercado hacen de Linux el sistema operativo ideal para su uso dentro de cursos de Sistemas Operativos. Sin embargo, su soporte para distintas arquitecturas de procesador, de una gran cantidad de dispositivos de hardware y su amplio rango de características, hacen de Linux un sistema operativo demasiado grande y complejo para un estudiante.

En esta tesis hemos presentado eLinux, una versión con propósitos educativos de Linux, que contiene sólo la funcionalidad necesaria para ser utilizado en un curso completo de Sistemas Operativos.

Tamaño descomprimido	227 MB
Archivos .c	6,906
Archivos .S (ensamblador)	6,773
Archivos .h	721
Total de archivos de código	14,400
Líneas de código en archivos .c	4,589,722
Líneas de código en archivos .S (ensamblador)	1,119,080
Líneas de código en archivos .h	256,488
Total de líneas de código	5,965,290
Cantidad de opciones de configuración	**5063**

Cuadro 6.1.: Algunas estadísticas de Linux 2.6.10.

Con la eliminación de las características mencionadas en los capítulos anteriores, fue posible reducir en gran proporción el código de Linux. El Cuadro 6.1 (es el Cuadro 1.4 repetido aquí por conveniencia) muestra las estadísticas de Linux 2.6.10. El Cuadro 6.2 muestra las estadísticas finales de eLinux. Como podemos observar, el total de código fuente fue reducido un 95 %.

Tamaño descomprimido	12 MB
Archivos .c	327
Archivos .S (ensamblador)	16
Archivos .h	425
Total de archivos de código	768
Porcentaje del total en Linux	5,33 %
Líneas de código en archivos .c	201, 749
Líneas de código en archivos .S (ensamblador)	5, 761
Líneas de código en archivos .h	62, 204
Total de líneas de código	269, 714
Porcentaje del total en Linux	4,52 %
Cantidad de opciones de configuración	146
Porcentaje del total en Linux	2,88 %

Cuadro 6.2.: Algunas estadísticas de eLinux 2.6.10.

El código de eLinux es mucho más legible que el de Linux, no sólo por su menor cantidad de líneas de código, sino por su menor cantidad de código condicional. Esto fue posible gracias a que eLinux está configurado para una sola arquitectura de procesador (`i386`) y para una sola configuración del sistema.

El correcto funcionamiento de eLinux ha sido comprobado ejecutando el conjunto de pruebas estándar LTP (Linux Test Project) que es utilizado para verificar cada versión de Linux (ver Sección 6.1).

eLinux puede ser instalado junto con cualquier instalación normal de Linux. Es compatible con las aplicaciones existentes aunque no ofrece todas las funcionalidades que muchas de ellas esperan.

Puede ser usado junto con emuladores como Bochs [1] y QEMU [3] para facilitar su ejecución, prueba y depuración evitando reiniciar la computadora. De esta manera, eLinux ofrece un entorno de desarrollo mucho más conveniente y real que otros sistemas operativos con propósitos educativos.

6.1. Verificación del Núcleo

Para verificar el correcto funcionamiento de eLinux utilizamos el conjunto de pruebas del proyecto Linux Test Project (LTP) [2]. LTP es un proyecto desarrollado por Silicon Graphics Inc. (SGI) e IBM con el objetivo de crear casos de prueba para validar la robustez, confiabilidad y la estabilidad del núcleo de Linux.

LTP incluye más de $2,900$ casos de prueba. Las pruebas incluidas en LTP pueden dividirse en 2 categorías: pruebas de regresiones (regression tests) y pruebas de estrés (stress testing).

Las pruebas de regresiones utilizan una gran cantidad de pruebas unitarias (unit testing). Las pruebas unitarias son utilizadas para probar el correcto funcionamiento de un módulo de software. En el caso del núcleo de Linux, las pruebas unitarias verifican el correcto funcionamiento de las llamadas a sistema. Así, LTP incluyen una o varias pruebas para cada llamada a sistema implementada en Linux. Una regresión sucede cuando alguna llamada a sistema, después de un cambio al código, reporta resultados diferentes a los esperados.

Las pruebas de estrés permiten analizar la estabilidad del sistema con cargas de trabajo muy pesadas. En LTP se incluyen pruebas de estrés para el administrador de memoria, el planificador de procesos, la administración de entrada/salida, etc.

Debido a la naturaleza de las modificaciones realizadas para eLinux, el conjunto de pruebas de LTP fue utilizado para verificar el correcto funcionamiento de nuestras modificaciones. Se utilizaron tanto las pruebas unitarias como las pruebas de estrés.

Es importante mencionar que ninguna llamada a sistema fue eliminada en eLinux. De esta manera, eLinux es compatible con todas las aplicaciones creadas para Linux. Sin embargo, algunas llamadas a sistema son referentes a funciones eliminadas del núcleo. Por ejemplo, `listen()`, `accept()` y todas las llamadas a sistema relacionadas con sockets en eLinux, regresan el error `ENOSYS` (Function not implemented).

A continuación se muestran los resultados de las pruebas de LTP. La salida de las pruebas de LTP son muy extensas por lo que sólo mostraremos el resumen final presentado por LTP.

dio Es un conjunto de pruebas de estrés para la entrada/salida en disco. Estas pruebas ejecutan una gran cantidad de las llamadas a sistemas `read()` y `write()`. Algunas de estas pruebas ejecutan varios hilos que leen y escriben simultáneamente un archivo.

```
LTP test: dio
-----------------------------------------------
Total Tests: 28
Total Failures: 0
Kernel Version: 2.6.10-eLinux
Machine Architecture: i686
Hostname: debian
```

ipc Es un conjunto de pruebas de estrés para la comunicación entre procesos (IPC).

Estas pruebas crean varios procesos que se comunican a través de tuberías con nombre o sin nombre.

```
LTP test: ipc
-----------------------------------------------
Total Tests: 8
Total Failures: 0
Kernel Version: 2.6.10-eLinux
Machine Architecture: i686
Hostname: debian
```

mm Es un conjunto de pruebas de estrés para la administración de memoria. Estas pruebas realizan una gran cantidad de peticiones de memoria, tanto pequeñas como muy grandes (hasta del 80 % del tamaño de la memoria RAM).

```
LTP test: mm
-----------------------------------------------
Total Tests: 21
Total Failures: 0
Kernel Version: 2.6.10-eLinux
Machine Architecture: i686
Hostname: debian
```

sched Es un conjunto de pruebas de estrés para el planificador de procesos. Estas pruebas crean una gran cantidad de hilos y prueban las distintas primitivas de sincronización entre hilos.

```
LTP test: sched
-----------------------------------------------
Total Tests: 3
Total Failures: 0
Kernel Version: 2.6.10-eLinux
Machine Architecture: i686
Hostname: debian
```

syscalls Es un conjunto de pruebas unitarias para cada una de las llamadas a sistema de Linux. Para algunas llamadas a sistema existen más de una prueba.

```
LTP test: syscalls
-----------------------------------------------
Total Tests: 654
Total Failures: 25
Kernel Version: 2.6.10-eLinux
Machine Architecture: i686
Hostname: debian
```

Esta es la única prueba en la que se existen pruebas falladas. Las pruebas falladas son las siguientes:

accept01	bind01
bind02	connect01
getpeername01	getsockname01
getsockopt01	listen01
recv01	recvfrom01
recvmsg01	send01
sendfile02	sendmsg01
sendto01	setsockopt01
socketcall01	socketcall02
socketcall03	socketcall04
sockioctl01	swapoff01
swapoff02	swapon01
swapon02	

Como podemos observar, las pruebas falladas son las relacionadas con la manipulación de sockets y con la activación/desactivación de la memoria de intercambio (swap). Estas características fueron eliminadas para eLinux y su valor de retorno es `ENOSYS` por lo que LTP las considera como falladas.

Apéndice A.

Entorno de Desarrollo del Núcleo de eLinux

Existen varias maneras de configurar el entorno para trabajar con eLinux. Aquí mostraremos 2 maneras convenientes de hacerlo. Lo descrito en este apéndice puede ser realizado en cualquier distribución de Linux (se recomienda una versión reciente).

A.1. Entorno de Desarrollo Normal

Al entorno aquí descrito lo llamamos "normal" debibo a que es el mismo procedimiento para actualizar el núcleo de Linux con una versión más reciente del mismo. La diferencia es que aquí no actualizamos el núcleo de Linux, sino que lo sustituimos por el núcleo de eLinux.

A.1.1. Compilando el código de eLinux

La compilación de eLinux es idéntica a la de Linux. Es recomendable revisar la guía para compilar e instalar el núcleo de Linux disponible en [24]. Aquí describiremos la forma más simple de compilar e instalar eLinux dentro de una instalación normal de Linux (cualquier distribución).

El software necesario para compilar eLinux es el siguiente:

- Gnu C compiler.
- Gnu make
- binutils

Los cuales deben estar instalados en el sistema. En Debian, pueden instalarse con:

```
$ apt-get install build-essential
```

Es conveniente crear un directorio donde se realizará todo lo que aquí describiremos. Aquí crearemos un directorio en `$HOME`.

```
$ cd $HOME
$ mkdir elinux
$ cd elinux
```

Ahora es necesario descargar el código fuente de eLinux. El núcleo de eLinux se encuentra en un archivo comprimido que puede descargarse de `http://computacion.cs.cinvestav.mx/~rdominguez/elinux/`.

Los siguientes comandos descargan y descomprimen eLinux:

```
$ wget http://computacion.../~rdominguez/elinux/elinux-2.6.10.tar.gz
$ tar zxvf elinux-2.6.10.tar.gz
```

Esto creará el directorio `elinux-2.6.10` dentro del directorio actual con el código de eLinux descomprimido.

El siguiente paso es configurar el núcleo. En este paso se determinan las opciones que se compilarán dentro del núcleo. Por ejemplo, aquí se puede elegir si se desea o no compilar el soporte al sistema de archivos `ext2`, los dispositivos que se desean soportar, etc. Para muchas de las opciones, se puede elegir entre: no incluirla, incluirla dentro del núcleo o incluirla como módulo. Incluir una opción dentro del núcleo quiere decir que el código de dicha opción es compilado y enlazado dentro de la imagen final del núcleo; incluirla como módulo implica que el código se compilará pero no se enlazará al núcleo, sino que se cargará y enlazará dinámicamente sólo si es necesario. Los módulos compilados son instalados dentro del directorio `/usr/lib/2.6.10-eLinux/`.

Para configurar el núcleo es necesario ejecutar:

```
$ cd elinux-2.6.10
$ make menuconfig
```

Aparecerá un menú de configuración del núcleo como el mostrado en la pantalla 1. En este menú aparecen una lista de submenús, y, dentro de ellos, las opciones a elegir. Una vez elegida la configuración del núcleo, ésta es guardada al salir del menú.

A diferencia de Linux, debido a todas las características removidas, eLinux tiene muy pocas opciones de configuración. eLinux contiene una configuración por omisión y es conveniente usarla pues eLinux contiene un conjunto mínimo de opciones y remover alguna de ellas podría ocasionar que el sistema no arrancara. Por lo tanto, no es necesario configurar eLinux através del menú y podemos sustituir ese paso con:

Pantalla 1 Menú de configuración del núcleo

```
 Linux Kernel v2.6.10cut Configuration
-----------------------------------------------------------------------------
  |--------------------- Linux Kernel Configuration -----------------------|
  |  Arrow keys navigate the menu.  <Enter> selects submenus --->.          |
  |  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, |
  |  <M> modularizes features.  Press <Esc><Esc> to exit, <|> for Help, </> |
  |  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >       |
  | |-------------------------------------------------------------------| |
  | |                Code maturity level options  --->                  | |
  | |                General setup  --->                                | |
  | |                Loadable module support  --->                      | |
  | |                Processor type and features  --->                  | |
  | |                Executable file formats  --->                      | |
  | |                Device Drivers  --->                               | |
  | |                File systems  --->                                 | |
  | |                Kernel hacking  --->                               | |
  | |            ---                                                    | |
  | |                Load an Alternate Configuration File               | |
  | |                Save Configuration to an Alternate File            | |
  | ------------------------------------------------------------------- |
  |-----------------------------------------------------------------------|
  |                    <Select>    < Exit >    < Help >                   |
  |-----------------------------------------------------------------------|
```

```
$ cd elinux-2.6.10
$ make defconfig
```

El cual crea una configuración por omisión para eLinux y que es muy recomendable usar.

Una vez que se ha configurado el núcleo, las opciones seleccionadas se guardan en el archivo `.config`, el cual, posteriormente, es utilizado para generar el archivo <`include/linux/autoconf.h`> que contiene una serie de `#define`'s con las características seleccionadas.

Para compilar el núcleo y los módulos seleccionados, sólo es necesario ejecutar `make` dentro del directorio de eLinux:

```
$ make
```

```
  Using /home/user/elinux/elinux-2.6.10 as source for kernel
  CHK     include/linux/version.h
  SPLIT   include/linux/autoconf.h -> include/config/*
...
  LD      vmlinux
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
...
  OBJCOPY arch/i386/boot/compressed/vmlinux.bin
  GZIP    arch/i386/boot/compressed/vmlinux.bin.gz
  LD      arch/i386/boot/setup
  LD      arch/i386/boot/compressed/piggy.o
  LD      arch/i386/boot/compressed/vmlinux
  OBJCOPY arch/i386/boot/vmlinux.bin
  BUILD   arch/i386/boot/bzImage
Root device is (3, 5)
Boot sector 512 bytes.
Setup is 4623 bytes.
System is 678 kB
Kernel: arch/i386/boot/bzImage is ready
```

Como se puede ver en la salida anterior, el archivo `arch/i386/boot/bzImage` es el núcleo. Este archivo es el que debe ser cargado al arrancar el sistema y el cargador de arranque (normalmente GRUB o LILO) se debe configurar para que utilice este núcleo. Generalmente, este archivo es copiado al directorio `/boot` donde residen los núcleos instalados y la configuración de GRUB (LILO utiliza `/etc`).

```
$ cp arch/i386/boot/bzImage /boot/vmlinuz-2.6.10-eLinux
```

A.1.2. Configurando el Cargador de Arranque

Posteriormente, es necesario configurar el cargador de arranque. En GRUB, se debe *agregar* una nueva entrada como la siguiente en el archivo `/boot/grub/menu.lst`:

```
title           eLinux 2.6.10
root            (hd0,4)
kernel          /boot/vmlinuz-2.6.10-eLinux root=/dev/hda5 ro
savedefault
boot
```

Los parámetros `(hd0,4)` y `/dev/hda5` varían entre sistemas y pueden ser tomados de la entrada en GRUB correspondiente al núcleo de la distribución instalada. Es importante que se agregue esta entrada y no se sustituyan las existentes, de tal manera que se pueda elegir, al momento de arrancar la máquina, entre el núcleo original de Linux y el núcleo de eLinux.

Si se compilaron opciones como módulos, es necesario instalarlos. Esto se hace con el comando:

```
$ make modules_install
```

Lo cual instala los módulos en el directorio `/lib/modules/$version/` donde `$version` es la versión del kernel, en este caso, `2.6.10-eLinux`. Con la configuración por defecto (defconfig) no se compila ningún módulo, por lo que no es necesario ejecutar este comando.

Una vez instalados los módulos y configurado el cargador de arranque con el nuevo núcleo, se debe reiniciar la máquina para que GRUB lea el nuevo archivo de configuración y ofrezca la opción de arrancar eLinux.

A.1.3. Desventajas

Con las instrucciones descritas en esta sección es posible utilizar una instalación normal de Linux para compilar el núcleo y ejecutarlo dentro de la misma instalación, es decir, usando el mismo software pero con diferente núcleo. Sin embargo, usar esta configuración puede ser problemática por distintos motivos. Uno es que es muy lento; se requiere reiniciar el sistema completo para probar las modificaciones realizadas. Otro motivo es por seguridad; al programar con el núcleo se pueden introducir errores que causen corrupción de datos e incluso se puede dañar algún dispositivo.

Otra posibilidad es tener 2 instalaciones de Linux en una misma máquina, una de ellas dedicada para eLinux. De esta manera, se aísla eLinux en su propio sistema de archivos sin que pueda interferir con la otra instalación. Pero aún existe el problema de los reinicios lentos y la posibilidad de dañar el hardware.

El entorno de desarrollo ideal es tener una máquina de desarrollo donde se realizan la edición y la compilación del código, y una o varias máquinas de prueba en las que se realizan tanto la prueba como la depuración del núcleo. En un entorno así, el núcleo puede copiarse a un disco flexible y arrancar la máquina de prueba con él. También existe la posibilidad de arrancar la máquina de prueba por red, es decir, que cargue el núcleo desde un servidor TFTP, pero la máquina de prueba debe soportar este tipo de arranque.

El contar con máquinas de prueba puede resultar bastante difícil. Afortunadamente existe la alternativa de utilizar emuladores.

A.2. Entorno de Desarrollo con QEMU

QEMU es un software que emula un sistema de cómputo completo, incluyendo el procesador y varios periféricos como el teclado, ratón, monitor, puertos seriales, discos duros, etc. Soporta completamente la emulación de distintas plataformas como lo son: Intel x86, Intel x86-64 y PowerPC.

El uso de este emulador es de gran ayuda para el desarrollo con el núcleo de eLinux, pues permite instalar y probar rápidamente el núcleo desarrollado sin la necesidad de reiniciar el sistema para probarlo, lo cual es bastante lento para el ciclo normal de programación (edición-compilación-ejecución-depuración). Además provee la seguridad de no arriesgar el sistema de desarrollo debido a errores graves en el núcleo desarrollado que pudieran ocasionar daños fatales, por ejemplo, en el sistema de archivos.

El entorno de desarrollo sugerido para eLinux es instalar una distribución de Linux dentro de una máquina virtual de QEMU.

A.2.1. Instalación y Configuración de QEMU

Para instalar QEMU es conveniente utilizar el administrador de programas utilizado por la distribución de Linux instalada. En Debian se puede instalar QEMU con:

```
$ apt-get install qemu
```

QEMU requiere que se le indiquen los dispositivos a emular. Uno de ellos es el disco duro a utilizar. QEMU puede utilizar cualquier archivo y utilizarlo como disco duro, pero dicho archivo debe contener la estructura estándar de un disco duro (sector de inicio, particiones, etc.). Un archivo con la estructura de un disco duro es conocido como "disco duro virtual".

Para ayudar a crear discos duros virtuales, QEMU incluye el comando `qemu-img`. Ahora, crearemos un disco duro virtual de 500 MB, en el directorio `$HOME/elinux/qemu`:

```
$ cd $HOME/elinux
$ mkdir qemu
$ cd qemu
$ qemu-img create hda.img 500M
```

El último comando genera un archivo llamado `hda.img` de 500 MB con la estructura adecuada. QEMU puede utilizar este archivo como el disco duro de donde leerá el sector de arranque.

A.2.2. Instalación de Linux dentro de QEMU

El siguiente paso es instalar en este disco duro virtual una distribución de Linux. Es recomendable instalar cualquier distribución que contenga alguna versión de Linux 2.6 (la mayoría de las versiones actuales de las distribuciones incluyen un núcleo 2.6).

Para instalar la distribución se requiere contar con los CD's de instalación de la misma. Con el primer CD de la distribución insertado en la unidad de CD-ROM, se debe arrancar QEMU indicándole el disco duro a utilizar (que recién creamos) y la unidad de CD-ROM a utilizar:

```
$ qemu -hda hda.img -cdrom /dev/cdrom -boot d
```

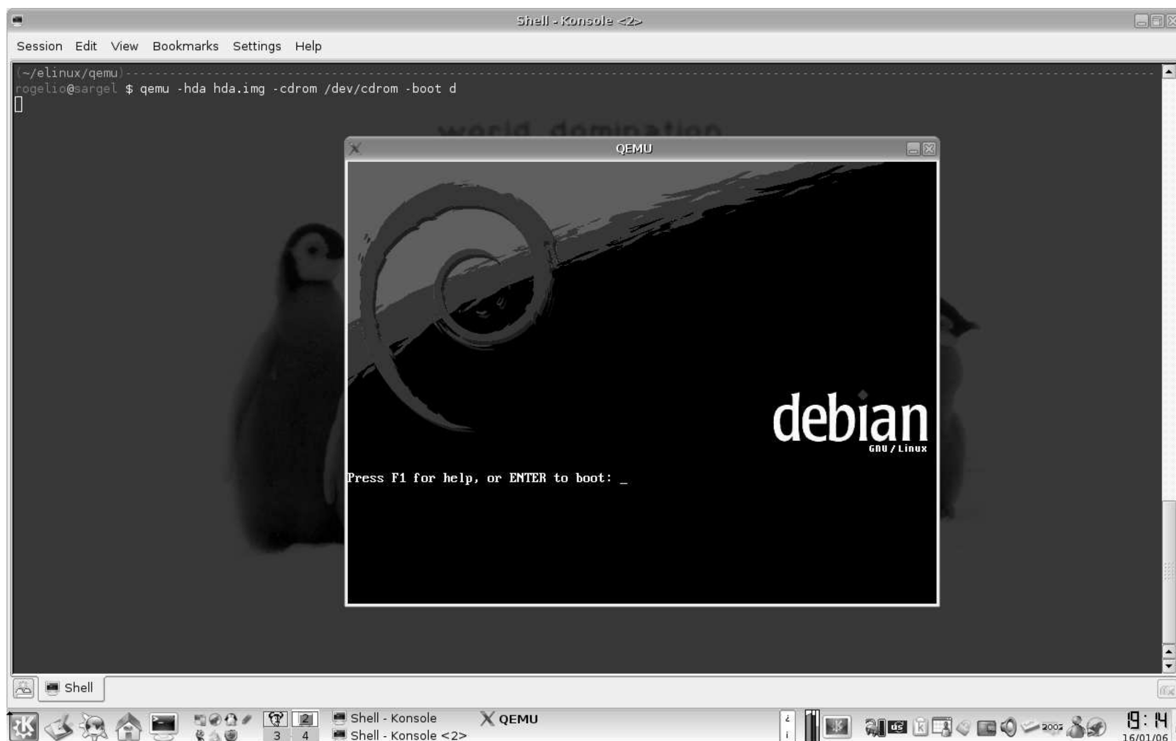

Figura A.1.: Instalando Debian en QEMU

Al ejecutar el anterior comando, QEMU abrirá una ventana como la de la Figura A.1 y arrancará desde el CD-ROM, lo leerá y procederá con la instalación de la distribución contenida en el CD, tal y como sucedería al instalarla en una máquina real. Se debe proceder con la instalación normal, asegurándose de utilizar el sistema de archivos `ext3` como formato de las particiones utilizadas en el disco duro virtual. Esto es debido

a que sustituiremos el núcleo de Linux de la distribución con el núcleo de eLinux, el cual sólo soporta sistemas de archivos `ext3`. También es recomendable sólo hacer una instalación básica de la distribución, sin muchos servicios ni entorno gráfico, pues algunos de ellos no podrán ejecutarse debido a las características removidas en eLinux. Es recomendable instalar dentro de QEMU el editor de texto de su preferencia (`vi`, `emacs`, etc.), pues en ocasiones podría requerirse editar archivos dentro de QEMU. De hecho, puede realizarse toda la edición y compilación de eLinux dentro de QEMU, pero presenta la inconveniencia de que la emulación es lenta y la compilación tarda más tiempo en realizarse.

Una vez finalizada la instalación, se estará listo para arrancar QEMU con la distribución elegida.

Alternativamente, Debian ofrece una manera más rápida y sencilla de crear una instalación de Debian dentro de un disco duro virtual para QEMU. El paquete `qemu` de Debian incluye un script llamado `qemu-make-debian-root`. El comando:

```
$ qemu-make-debian-root 500 sarge http://ftp.de.debian.org/debian/ hda.img
```

instalará el sistema base de la versión "Sarge" de Debian, descargando los paquetes del servidor especificado, en un disco duro virtual de 500 MB en el archivo `hda.img`. Es decir, todos los pasos descritos en esta sección se resumen en el anterior comando, instalando Debian en el disco duro virtual `hda.img`.

A.2.3. Ejecutando QEMU

Una vez que se tiene Linux instalado en un disco duro virtual, se puede arrancar la instalación con:

```
$ cd $HOME/elinux/qemu
$ qemu -hda hda.img
```

Al ejecutar el anterior comando, se ejecutará QEMU con el sistema instalado, el cual puede usarse normalmente.

A.2.4. Ejecutando eLinux en QEMU

Una opción muy útil de QEMU es que permite cargar un núcleo sin tenerlo instalado dentro del disco duro virtual, es decir, hace la función del cargador de arranque. Por ejemplo, para arrancar la máquina virtual con el núcleo de eLinux compilado en la sección anterior, podemos usar el siguiente comando:

```
$ cd $HOME/elinux
$ qemu -hda qemu/hda.img -kernel elinux-2.6.10/arch/i386/boot/bzImage \\
  -append "root=/dev/hda1"
```

En resumen, para compilar el núcleo de eLinux y ejecutar QEMU con él, se ejecutan los siguientes comandos:

```
$ cd $HOME/elinux/elinux-2.6.10
$ make defconfig
$ make
$ cd ..
$ qemu -hda qemu/hda.img -kernel elinux-2.6.10/arch/i386/boot/bzImage \\
  -append "root=/dev/hda1"
```

A.2.5. Copiando archivos al disco duro virtual de QEMU

Si se requiere copiar archivos al disco duro virtual de QEMU, se puede montar la primera partición del disco duro virtual con:

```
$ mkdir /mnt/virtual
$ mount $HOME/elinux/qemu/hda.img /mnt/virtual -o loop,offset=32256
```

Y así, en el directorio `/mnt/virtual`. se puede acceder al sistema de archivos contenido en el disco duro virtual de QEMU.

Apéndice B.

Estructuras del Administrador de Procesos

B.1. prio_array

Esta estructura contiene las colas de prioridades de procesos.

```
#define MAX_PRIO      140
#define BITMAP_SIZE  ((((MAX_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long))

struct prio_array {
    unsigned int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};

typedef struct prio_array prio_array_t;
```

`nr_active`→ La cantidad de procesos en este arreglo.

`bitmap`→ Un mapa de bits con un bit por cada prioridad que Linux maneja (140).

`queue`→ Un arreglo de colas de procesos. Contiene una cola por cada prioridad.

B.2. runqueue

En esta estructura se encuentran todos los procesos listos, ya sea en la cola de procesos que aún tienen tiempo de ejecución asignado o en la cola de procesos que han expirado su tiempo de ejecución.

```
struct runqueue {
    spinlock_t lock;
    unsigned long nr_running;
    unsigned long long nr_switches;
    unsigned long nr_uninterruptible;
    unsigned long expired_timestamp;
    unsigned long long timestamp_last_tick;
    task_t *curr, *idle;
    struct mm_struct *prev_mm;
    prio_array_t *active, *expired, arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;
};

typedef struct runqueue runqueue_t;
```

`lock`→ Semáforo para evitar el acceso simultáneo a esta estructura.

`nr_running`→ Número de procesos listos.

`nr_switches`→ Número de cambios de contexto realizados hasta el momento.

`nr_uninterruptible`→ Número de procesos en estado `UNINTERRUPTIBLE`

`expired_timestamp`→ Tiempo del último cambio de los arreglos de prioridad.

`timestamp_last_tick`→ Tiempo de la última interrupción del reloj.

`curr`→ Proceso que actualmente está usando el procesador.

`idle`→ Proceso nulo (idle).

`prev_mm`→ `mm_struct` del último proceso que utilizó el procesador.

`active`→ Arreglo de prioridad activo.

`expired`→ Arreglo de prioridad expirado.

`arrays`→ Contiene los dos anteriores arreglos de prioridad.

`best_expired_prio`→ Prioridad más alta de los procesos en el arreglo de prioridad `expired`.

`nr_iowait`→ Número de procesos esperando por una operación de Entrada/Salida.

B.3. task_struct

Es el descriptor de proceso. Es una estructura muy grande, por lo que la describiremos por partes.

```
struct task_struct {
    volatile long state;
    struct thread_info *thread_info;
    unsigned long flags;
```

state→ Indica el estado del proceso. Los posibles estados de un proceso son:

- TASK_RUNNING
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_STOPPED
- TASK_ZOMBIE

thread_info→ Estructura que contiene datos dependientes de la arquitectura y es utilizada para mantener datos cuyo acceso debe ser eficiente.

flags→ Banderas sobre el estado y propiedades del proceso. Los posibles valores son:

PF_STARTING El proceso está siendo creado.

PF_EXITING El proceso está destruyéndose.

PF_DEAD El proceso está muerto.

PF_FORKNOEXEC El proceso no ha ejecutado fork()

PF_SUPERPRIV El proceso ha obtenido privilegios de root (no necesariamente los tiene ahora)

PF_DUMPCORE El proceso ha realizado un volcado de memoria.

PF_SIGNALED El proceso ha sido terminado por una señal.

PF_MEMALLOC El proceso está reservando memoria.

`PF_MEMDIE` El proceso ha sido terminado por falta de memoria en el sistema.

`PF_FLUSHER` El proceso es un hilo del núcleo llamado `pdflush()` que se encarga de escribir en disco todos los búfers sucios.

`PF_KSWAPD` El proceso es un hilo del núcleo llamado `kswapd()` que se encarga de liberar memoria cuando el sistema tiene poca memoria.

`PF_SYNCWRITE` El proceso está haciendo una escritura síncrona.

`PF_BORROWED_MM` El proceso es un hilo del núcleo.

```
int prio, static_prio;
struct list_head run_list;
prio_array_t *array;

unsigned long sleep_avg;
long interactive_credit;
unsigned long long timestamp, last_ran;
int activated;

unsigned long policy;
cpumask_t cpus_allowed;
unsigned int time_slice, first_time_slice;
```

Todos los campos anteriores son utilizados por el algoritmo de planificación y su uso es descrito en la sección 2.1.2.

```
struct list_head tasks;
struct mm_struct *mm, *active_mm;
```

`tasks`→ A través de este campo todas las tareas están ligadas.

`mm`→ Apuntador a la estructura de tipo `mm_struct` que representa el espacio de direcciones del proceso.

`active_mm`→ Apuntador a la estructura de tipo `mm_struct` que representa el espacio de direcciones del proceso utilizado por el proceso.

```
struct linux_binfmt *binfmt;
long exit_state;
int exit_code, exit_signal;
int pdeath_signal;
unsigned did_exec:1;
```

binfmt→ Linux soporta diferentes formatos de archivos ejecutables (ELF, a.out y ECOFF). Para soportarlos, utiliza estructuras de tipo `linux_binfmt` definida así (en el archivo <`linux/binfmts.h`>):

```
struct linux_binfmt {
    struct linux_binfmt * next;
    struct module *module;
    int (*load_binary)(struct linux_binprm *,
                       struct  pt_regs * regs);
    int (*load_shlib)(struct file *);
    int (*core_dump)(long signr, struct pt_regs * regs,
                     struct file * file);
    unsigned long min_coredump;
};
```

Donde los apuntadores `linux_binfmt`→`load_binary`, `linux_binfmt`→`load_shlib` y `linux_binfmt`→`core_dump` son funciones para cargar el archivo, cargar una librería dinámica y hacer un volcado de memoria, respectivamente, y que son específicas del formato del archivo ejecutable.

exit_state→ Indica el estado de terminación del proceso. Puede tomar 2 valores:

`EXIT_ZOMBIE` El proceso ha terminado pero se encuentra en estado "zombie".

`EXIT_DEAD` El proceso ha finalizado y está en proceso de destrucción.

exit_code→ Guarda el código de retorno del proceso al terminar.

exit_signal→ Guarda el número de la señal que terminó este proceso (si ése fue el motivo).

pdeath_signal→ Guarda la señal que será enviada a este proceso cuando alguno de sus procesos hijos terminen.

did_exec→ Este campo indica si el proceso ha ejecutado la función `exec()`. Es puesto a `1` por llamadas a sistema como `exec()` y en `0` por `fork()`.

```
pid_t pid;
pid_t tgid;
struct task_struct *parent;
struct list_head children;
struct list_head sibling;
struct task_struct *group_leader;
struct pid pids[PIDTYPE_MAX];
```

`pid`→ El PID (Process ID) del proceso.

`tgid`→ El estándar POSIX define que todos los hilos que comparten un espacio de direcciones deben compartir el mismo PID. Linux almacena en el campo `tgid` el PID del primer hilo del grupo de hilos con el mismo espacio de direcciones. La llamada a sistema `getpid()` regresa este valor.

`parent`→ Apuntador a la estructura `task_struct` del padre de este proceso.

`children`→ Lista de todos los hijos de este proceso.

`sibling`→ Lista de todos los hermanos de este proceso.

`group_leader`→ Apuntador al proceso líder de un grupo de procesos.

`pids[PIDTYPE_MAX]`→ Cada proceso tiene 4 PID's:

`PID` Process ID.

`TGID` Thread Group ID.

`PGID` Process Group ID.

`SID` Session ID.

```
unsigned long min_flt, maj_flt;
uid_t uid,euid,suid,fsuid;
gid_t gid,egid,sgid,fsgid;
struct group_info *group_info;
struct user_struct *user;
```

`min_flt, maj_flt`→ El administrador de memoria mantiene estadísticas sobre la cantidad de fallos de páginas por proceso. El campo `min_flt` cuenta la cantidad de fallas menores del proceso y el campo `max_flt` cuenta la cantidad de fallas mayores.

`uid`→ Es el identificador del usuario que inició el proceso.

`euid`→ Effective User Identifier. Un proceso puede adquirir los permisos de un usuario diferente durante su ejecución (por ejemplo el comando `sudo`). Este campo guarda el valor del identificador de usuario cuyos permisos ha adquirido este proceso.

`suid`→ Saved User Identifier. Cuando un programa cambia su `euid`, es también guardado en `suid`. De esta manera, un proceso puede volver temporalmente a su `uid` original y regresar posteriormente a su `euid`.

`fsuid`→ Este campo indica el identificador de usuario utilizado para el acceso al sistema de archivos. Siempre que se cambia el `euid`, el `fsuid` también es cambiado. Raramente es cambiado, solamente programas como el servidor NFS realizan cambios en este campo.

`gid, egid, sgid , fsgid`→ Estos campos tienen la misma funcionalidad que lo anteriores pero estos se refieren al grupo al cual pertenece el usuario dueño del proceso.

`group_info`→ Es una estructura que contiene información sobre los grupos a los cuales pertenece el usuario dueño del proceso (un usuario puede pertenecer a varios grupos).

`user`→ Es una estructura que contiene algunas estadísticas sobre los recursos que posee un usuario. Por ejemplo, mantiene la cantidad de procesos que el usuario está ejecutando y la cantidad de archivos abiertos

```
kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
unsigned keep_capabilities:1;
```

`cap_effective`→ Es un mapa de bits, donde cada bit indica si el proceso tiene el permiso para realizar cierta acción. En <`linux/capability.h`> están definidas las capacidades de una manera muy detallada.

`cap_inheritable`→ Es un mapa de bits que determina las capacidades que el proceso mantendrá después de ejecutar la llamada a sistema `exec()`.

`cap_permitted`→ Es un mapa de bits que indica las capacidades que el proceso puede adquirir.

`keep_capabilities`→ Bandera que indica si las capacidades serán mantenidas después de ejecutar la llamada a sistema `exec()`.

```
struct thread_struct thread;
struct fs_struct *fs;
struct files_struct *files;
struct namespace *namespace;
struct dentry *proc_dentry;
```

`thread`→ Es una estructura que contiene datos específicos de la arquitectura. Contiene el estado del procesador en el momento en que fue detenido por un cambio de contexto. En x86, contiene copia de registros como `eip`, `esp`, `cr2`, entre otros.

`fs`→ Es un apuntador a una estructura de tipo `fs_struct`, la cual está definida en `<linux/fs_struct.h>`.

```
struct fs_struct {
    atomic_t count;
    int umask;
    struct dentry * root, * pwd, * altroot;
    struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
};
```

La cual, principalmente, contiene apuntadores a las estructuras `dentry` y `vfsmount` del directorio raíz del proceso, el directorio actual del proceso, y el directorio raíz alterno del proceso (puede ser cambiado con programas como `chroot`).

`files`→ Apuntador a una estructura de tipo `files_struct` que contiene información sobre los archivos abiertos por el proceso.

`namespace`→ En Linux es posible que un proceso tenga una "vista" diferente del sistema de archivos (la jerarquía de directorios) que los demás procesos. Es por esto que cada proceso tiene asociada una estructura `namespace` que describe cuál su jerarquía de directorios.

`proc_dentry`→ Estructura `dentry` asociada al proceso en el sistema de archivos virtual `procfs` (montado típicamente en `/proc`).

```
struct signal_struct *signal;
struct sighand_struct *sighand;
sigset_t blocked;
struct sigpending pending;
```

`signal`→ Estructura de tipo `signal_struct` que contiene información sobre el grupo de procesos que comparten los mismos manejadores de señales.

`sighand`→ Estructura de tipo `sighand_struct` que contiene información sobre los manejadores de señales.

`blocked`→ Mapa de bits que indica cuáles señales han sido bloqueadas por el proceso.

`pending`→ Estructura de tipo `sigpending` que contiene una lista de las señales pendientes del proceso.

Apéndice C.

Estructuras del Administrador de Memoria

C.1. vm_area_struct

Esta estructura representa un intervalo asignado del espacio de direcciones virtuales de un proceso.

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;
    unsigned long vm_flags;

    struct rb_node vm_rb;

    struct vm_operations_struct * vm_ops;

    unsigned long vm_pgoff;
    struct file * vm_file;
    void * vm_private_data;
};
```

`vm_mm`→ Estructura `mm_struct` a la cual pertenece esta area.

`vm_start`→ Dirección inicial de este intervalo.

`vm_end`→ Dirección final de este intervalo.

`vm_next`→ Apuntador a la siguiente área de este proceso.

`vm_page_prot`→ Permisos de acceso.

`vm_flags`→ Banderas que indican información sobre el contenido de esta área.

`vm_rb`→ Estas estructuras están ordenadas en un árbolo roji-negro. Este campo liga los nodos del árbol.

`vm_ops`→ Es un conjunto de funciones utilizadas para zonas de memoria que representan archivos mapeados a memoria.

`vm_pgoff`→ Es el offset en el archivo que está mapeado en esta zona.

`vm_file`→ Es el descriptor de archivo que está mapeado en esta zona.

C.2. mm_struct

Representa el espacio de direcciones del proceso.

```
struct mm_struct {
    struct vm_area_struct * mmap;
    struct rb_root mm_rb;

    unsigned long (*get_unmapped_area) (struct file *filp,
                unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags);
    void (*unmap_area) (struct vm_area_struct *area);

    pgd_t * pgd;
    atomic_t mm_users;
    atomic_t mm_count;
    int map_count;

    struct list_head mmlist;

    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;

};
```

`mmap`→ Apuntador a la primera estructura de tipo `vm_area_struct`, es decir, a la lista ligada de zonas de memoria utilizadas en este espacio de direcciones.

`mm_rb`→ Apuntador a la raíz del árbol rojinegro de las zonas de memoria utilizadas en este espacio de direcciones.

`get_unmapped_area`→ Es un apuntador a una función que es llamada en cada ocasión que se requiere encontrar un intervalo libre en el espacio de direcciones. Comúnmente, este apuntador hace referencia a la función `arch_get_unmapped_area()`.

`unmap_area`→ Es un apuntador a una función que es llamada en cada ocasión que un intervalo de memoria es removido del espacio de direcciones. Comúnmente, este apuntador hace referencia a la función `arch_unmap_area()`.

`pgd`→ Apuntador al Page Global Directory de este espacio de direcciones.

`mm_users`→ Cantidad de hilos utilizando este espacio de direcciones.

`mm_count`→ Cantidad de referencias a este espacio de direcciones. Todos los hilos que utilizan este espacio de direcciones sólo cuentan como 1 en `mm_count`. Los hilos del kernel son los otros usuarios que pueden incrementar esta cuenta.

`map_count`→ Cantidad de estructuras `vm_area_struct` ligadas en `mmap`

`mm_list`→ Todas las estructuras `mm_struct` existentes en el sistema están ligadas a través de esta lista.

`start_code, end_code`→ Direcciones de inicio y de fin de la sección de código del proceso.

`start_data, end_data`→ Direcciones de inicio y de fin de la sección de datos del proceso.

`start_brk, brk`→ Direcciones de inicio y de fin de la sección de area dinámica (también conocida como heap).

`start_stack`→ Dirección de inicio de la pila.

`arg_start, arg_end`→ Direcciones de inicio y de fin de la lista de argumentos del proceso.

`env_start, env_end`→ Direcciones de inicio y de fin de la lista de variables de entorno del proceso.

Apéndice D.

Estructuras de la Capa de Entrada/Salida para Dispositivos de Bloque

D.1. buffer_head

Representa un bloque de disco cuya copia se mantiene en memoria.

```
struct buffer_head {
    unsigned long b_state;
    struct buffer_head *b_this_page;
    struct page *b_page;
    atomic_t b_count;
    u32 b_size;

    sector_t b_blocknr;
    char *b_data;

    struct block_device *b_bdev;
    bh_end_io_t *b_end_io;
    void *b_private;
    struct list_head b_assoc_buffers;
};
```

b_state→ Indica el estado del búfer. Algunos de los valores que puede tomar son:

- **BH_Uptodate** Indica que el búfer contiene datos válidos.
- **BH_Dirty** Indica que el bloque en memoria es más actual que el bloque en disco y requiere ser sincronizado.

BH_Lock Indica que el búfer está siendo sincronizado con el disco y se debe evitar su acceso concurrente.

BH_Req Indica que el búfer está, o está siendo preparado, para una transferencia de Entrada/Salida.

BH_Mapped Indica que el búfer es un mapeo válido de un bloque en disco.

BH_New Indica que el búfer es nuevo y que no ha sido leído ni escrito.

BH_Write_EIO Indica que el búfer contiene datos válidos pero que no se ha podido sincronizar al disco por que hubo error al intentar escribirlo.

`b_this_page`→ Es un apuntador a la siguiente estructura `buffer_head` que comparte la misma página. Dado que el tamaño del bloque puede ser menor que el tamaño de la página, es posible que una página contenga 2 o más búfers. Con ayuda de este campo se mantiene una lista circular de todos los búfers contenidos en una página.

`b_page`→ Es un apuntador al descriptor de la página que contiene al búfer.

`b_count`→ Es la cuenta de los usuarios de este búfer.

`b_size`→ Indica el tamaño del bloque.

`b_blocknr`→ Indica el número del bloque del disco asociado con este búfer.

`b_data`→ Es un apuntador al inicio de los datos.

`b_dev`→ Es un apuntador a una estructura de tipo `block_device` que representa el dispositivo de bloque asociado con este búfer.

`b_end_io`→ Es un apuntador a una función que se ejecuta al finalizar una operación de escritura o lectura en este búfer. Es necesario pues las operaciones a realizar al finalizar dichas operaciones varían dependiendo de la operación realizada.

`b_private`→ Este campo puede ser usado para pasar parámetros adicionales a la función apuntada por el campo `b_end_io`

`b_assoc_buffers`→ Este campo se utiliza para ligar todos los búfers asociados con un archivo mapeado en memoria.

D.2. Estructura bio

La estructura bio es la unidad principal de entrada/salida en Linux. Representa un conjunto de transferencias a disco.

```
struct bio {
    sector_t              bi_sector;
    struct bio            *bi_next;
    struct block_device   *bi_bdev;
    unsigned long         bi_flags;
    unsigned long         bi_rw;
    unsigned short        bi_vcnt;
    unsigned short        bi_idx;

    unsigned int          bi_size;

    unsigned int          bi_max_vecs;

    struct bio_vec        *bi_io_vec;

    bio_end_io_t          *bi_end_io;
    atomic_t              bi_cnt;

    void                  *bi_private;
    bio_destructor_t      *bi_destructor;
};
```

bi_sector→ Sector del disco asociado a esta operación.

bi_next→ Todas las estructuras bio están ligadas a través de este campo.

bi_bdev→ Dispositivo asociado.

bi_flags→ Banderas que indican el estado de la transacción.

bi_rw→ Indica si la transacción será de lectura o escritura.

bi_vcnt→ Tamaño del arreglo bi_io_vec.

bi_idx→ Indica el índice en el arreglobi_io_vec de la estructura bio_vec que será el siguiente segmento en transferirse.

bi_size→ Tamaño en bytes de la operación.

`bi_max_vecs`→ Cantidad máxima de estructuras `bio_vec` que se pueden tener en el siguiente arreglo.

`bi_io_vec`→ apuntador a un arreglo de estructuras de tipo `bio_vec`.

`bi_end_io`→ Apuntador a una función que se llama al finalizar la transacción.

`bi_cnt`→ Cantidad de referencias a esta estructura.

`bi_private`→ Datos para las funciones en `bi_end_io` y/o `bi_destructor`.

`bi_destructor`→ Apuntador a una función que se llama al liberar la memoria de esta estructura.

Apéndice E.

Estructuras de la capa VFS

E.1. file_system_type

Linux mantiene un registro de los sistemas de archivo soportados a través de la estructura `file_system_type` definida en `<include/linux/fs.h>`.

```
struct file_system_type {
    const char              *name;
    int                     fs_flags;
    struct super_block *(*get_sb) (struct file_system_type *, int,
                               const char *, void *);
    void (*kill_sb) (struct super_block *);
    struct module           *owner;
    struct file_system_type * next;
    struct list_head        fs_supers;
};
```

`name`→ Contiene el nombre del sistema de archivos.

`fs_flags`→ Algunas banderas sobre características del sistema de archivos. La bandera más importante aquí es `FS_REQUIRES_DEV`, la cual indica si el sistema de archivos requiere un dispositivo.

`get_sb`→ Apuntador a una función específica del sistema de archivos cuya función debe ser leer el superbloque del disco y llenar una estructura de tipo `super_block` con los datos leídos del disco.

`kill_sb`→ Apuntador a una función específica del sistema de archivos cuya función debe ser liberar todos los recursos relacionados con el sistema de archivos montado.

`owner`→ Módulo asociado con el sistema de archivos.

`next`→ Linux mantiene todos las estructuras `file_system_type` en una lista ligada a través de este campo.

`fs_supers`→ Cabecera de una lista de superbloques pertenecientes a este tipo de sistema de archivos.

E.2. `super_block`

La estructura `super_block` describe las características generales de un sistema de archivos.

```
struct super_block {
    struct list_head        s_list;
    dev_t                   s_dev;
    unsigned long           s_blocksize;
    unsigned long           s_old_blocksize;
    unsigned char           s_blocksize_bits;
    unsigned char           s_dirt;
    unsigned long long      s_maxbytes;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    unsigned long           s_flags;
    unsigned long           s_magic;
    struct dentry           *s_root;
    struct rw_semaphore     s_umount;
    struct semaphore        s_lock;
    int                     s_count;
    int                     s_syncing;
    int                     s_need_sync_fs;
    atomic_t                s_active;

    struct list_head        s_dirty;
    struct list_head        s_io;
    struct list_head        s_files;

    struct block_device     *s_bdev;
    struct list_head        s_instances;
    struct quota_info       s_dquot;

    int                     s_frozen;
    wait_queue_head_t       s_wait_unfrozen;
```

```
    char                          s_id[32];

    void                          *s_fs_info;
};
```

s_list→ Campo para mantener todos las estructuras super_block en una lista.

s_dev→ Identificador del dispositivo asociado.

s_blocksize→ Tamaño del bloque en bytes.

s_old_blocksize→ Anterior tamaño del bloque en bytes. Los dispositivos utilizan un tamaño de bloque por defecto, la mayoría de los sistemas de archivos permiten utilizar diferentes tamaños de bloque por lo que requieren cambiar el valor por defecto. Esta variable permite restaurar dicho valor.

s_blocksize_bits→ Tamaño de bloque en bits.

s_dirt→ Bandera que indica si los datos del superbloque en memoria difieren de los datos del superbloque en disco.

s_maxbytes→ Tamaño máximo de un archivo.

s_type→ Apuntador a la estructura file_system_type correspondiente al tipo de sistema de archivos de este superbloque.

s_op→ Apuntador a las operaciones de superbloque implementadas por este sistema de archivos (ver más adelante).

s_flags→ Banderas que indican el modo en que se montó el sistema de archivos.

s_magic→ Indentificador único para cada sistema de archivos.

s_root→ Apuntador a la estructura dentry (ver más adelante) correspondiente al directorio donde este sistema de archivos se encuentra montado.

s_umount→ Semáforo utilizado cuando se desmonta el sistema de archivos.

s_lock→ Semáforo para serializar el acceso al superbloque.

s_count→ Cuenta de referencias temporales a este superbloque.

s_syncing→ Bandera que indica que el supebloque está siendo escrito a disco.

`s_need_sync_fs`→ Bandera utilizada durante el proceso de sincronizar todos los sistemas de archivos.

`s_active`→ Referencias activas a este superbloque.

`s_dirty`→ Lista de inodos sucios.

`s_io`→ Lista temporal de inodos que serán sincronizados a disco.

`s_files`→ Lista de archivos abiertos.

`s_bdev`→ Dispositivo de bloque asociado a este superbloque.

`s_instances`→ Todos los sistemas de archivos de un mismo tipo son mantenidos en una lista a través de este campo.

`s_id[32]`→ Nombre del sistema de archivos.

`s_fs_info`→ Información específica del sistema de archivos.

E.3. `super_operations`

La estructura `super_operations` contiene apuntadores a funciones que implementan la funcionalidad del sistema de archivos relacionada con el superbloque.

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);

    void (*read_inode) (struct inode *);

    void (*dirty_inode) (struct inode *);
    int  (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int  (*sync_fs)(struct super_block *sb, int wait);
    int  (*statfs) (struct super_block *, struct kstatfs *);
    int  (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
};
```

`alloc_inode`→ Crea e inicializa un inodo en memoria.

`destroy_inode`→ Destruye un inode en mmoria.

`read_inode`→ Lee y llena un inodo desde el disco.

`dirty_inode`→ Se llama cuando los datos del inodo son cambiados.

`write_inode`→ Escribe el inodo a disco.

`put_inode`→ Disminuye la cuenta de referencias a este inodo.

`drop_inode`→ Se llama cuando se libera la última referencia a un inodo.

`delete_inode`→ Borra el inodo del disco.

`put_super`→ Se llama al desmontar el sistema de archivos.

`write_super`→ Se llama cuando se requiere escribir en disco el superbloque en memoria.

`sync_fs`→ Sincroniza el superbloque en memoria.

`statfs`→ Llena una estructura `statfs`, la cual contiene estadísticas sobre el sistema de archivos.

`remount_fs`→ Se llama al remontar el sistema de archivos con diferentes opciones de montaje.

`clear_inode`→ Libera el inodo y toda la memoria relacionada con él.

E.4. inode

Cada estructura `inode` contiene meta-información sobre un archivo.

```
struct inode {
    struct hlist_node       i_hash;
    struct list_head        i_list;
    struct list_head        i_dentry;
    unsigned long           i_ino;
    atomic_t                i_count;
    umode_t                 i_mode;
    unsigned int            i_nlink;
    uid_t                   i_uid;
```

```
    gid_t                      i_gid;
    dev_t                      i_rdev;
    loff_t                     i_size;
    struct timespec            i_atime;
    struct timespec            i_mtime;
    struct timespec            i_ctime;
    unsigned int               i_blkbits;
    unsigned long              i_blksize;
    unsigned long              i_version;
    unsigned long              i_blocks;
    unsigned short             i_bytes;
    unsigned char              i_sock;
    spinlock_t                 i_lock;
    struct semaphore           i_sem;
    struct rw_semaphore        i_alloc_sem;
    struct inode_operations    *i_op;
    struct file_operations     *i_fop;
    struct super_block         *i_sb;
    struct file_lock           *i_flock;
    struct address_space       *i_mapping;
    struct address_space       i_data;
    struct list_head           i_devices;
    struct pipe_inode_info     *i_pipe;
    struct block_device        *i_bdev;
    struct cdev                *i_cdev;

    __u32                      i_generation;

    unsigned long              i_state;
    unsigned long              dirtied_when;

    unsigned int               i_flags;

    atomic_t                   i_writecount;
    union {
        void                   *generic_ip;
    } u;
};
```

`i_hash`→ Todos los inodos se mantienen en una tabla hash. Este campo permite ligar todos los inodos con el mismo hash.

`i_list`→ Los inodos son mantenidos en diferntes listas. Este campo liga el inodo a su lista correspondiente.

`i_dentry`→ Lista de todos las estructuras `dentry` cuyo inodo es éste.

`i_ino`→ Número de inodo (único)

`i_count`→ Cuenta de referencias a este inodo.

`i_mode`→ Tipo de archivo y permisos de acceso.

`i_nlink`→ Número de ligas duras.

`i_uid`→ Identificador del propietario.

`i_gid`→ Identificador del grupo.

`i_rdev`→ Identificador del dispositivo donde reside este inodo.

`i_size`→ Tamaño del archivo en bytes.

`i_atime`→ Fecha del último acceso.

`i_mtime`→ Fecha de la última modificación.

`i_ctime`→ Fecha del último cambio a la meta-información del inodo.

`i_blkbits`→ Tamaño de bloque en bits.

`i_blksize`→ Tamaño de bloque en bytes.

`i_version`→ Número de version del inodo. Este número lo pueden usar los distintos sistemas de archivo para detectar cambios en un inodo después de ciertas operaciones. Comúnmente es incrementado en cada modificación al inodo.

`i_blocks`→ Número de bloques en el archivo.

`i_bytes`→ Número de bytes consumidos en el último bloque del archivo.

`i_sock`→ Bandera que indica que este archivo es un socket.

`i_lock`→ Candado (spinlock) para protección de los campos del inodo.

`i_sem`→ Semáforo para protección de los campos del inodo.

`i_alloc_sem`→ Semáforo (tipo lectores-escritores) para protección de los campos del inodo.

`i_op`→ Apuntador a una estructura `inode_operations` con las operaciones implementadas por este sistema de archivos.

`i_fop`→ Apuntador a una estructura `file_operations` con las operaciones implementadas por este sistema de archivos.

`i_sb`→ Superbloque asociado a este inodo.

`i_flock`→ Apuntador a una estructura `file_lock` para implementar candados en archivos.

`i_mapping`→ Apuntador a la estructura `address_space` correspondiente al inodo. Esta estructura contiene las operaciones de bajo nivel para leer y escribir páginas.

`i_data`→ Estructura `address_space` correspondiente al inodo. Este campo sólo es usado por los inodos que representan un dispositivo de bloque.

`i_devices`→ Todos los inodos que representan a un mismo dispositivo se encuentran en una lista ligada a través de este campo.

`i_pipe`→ Campo utilizado para describir información cuando el inodo es una tubería (pipe).

`i_bdev`→ Apuntador al dispositivo de bloque representado por este inodo.

`i_cdev`→ Apuntador al dispositivo de caracter representado por este inodo.

`i_generation`→ Versión del inodo. Algunos sistemas de archivos han cambiado la estructura de sus inodos y utilizan este campo para diferenciar entre las distintas versiones.

`i_state`→ Banderas que indican el estado del inodo.

`dirtied_when`→ Tiempo en que el inodo cambió por primera vez de su estado en disco.

`i_flags`→ Banderas que indican las opciones de montaje del sistema de archivos.

`i_writecount`→ Cuenta de procesos escritores.

`generic_ip`→ Información específica del sistema de archivos.

E.5. inode_operations

La estructura `inode_operations` contiene las operaciones sobre inodos implementadas por el sistema de archivos.

```
struct inode_operations {
    int  (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *,struct dentry *,
                               struct nameidata *);
    int  (*link) (struct dentry *,struct inode *,struct dentry *);
    int  (*unlink) (struct inode *,struct dentry *);
    int  (*symlink) (struct inode *,struct dentry *,const char *);
    int  (*mkdir) (struct inode *,struct dentry *,int);
    int  (*rmdir) (struct inode *,struct dentry *);
    int  (*mknod) (struct inode *,struct dentry *,int,dev_t);
    int  (*rename) (struct inode *, struct dentry *,
                    struct inode *, struct dentry *);
    int  (*readlink) (struct dentry *, char __user *,int);
    int  (*follow_link) (struct dentry *, struct nameidata *);
    void (*truncate) (struct inode *);
    int  (*permission) (struct inode *, int, struct nameidata *);
    int  (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int  (*removexattr) (struct dentry *, const char *);
};
```

`create`→ Crea un nuevo inodo en disco con la estructura `dentry` asociada a él.

`lookup`→ Busca, en el directorio especificado por el inodo, el archivo o directorio descrito en la estructura `dentry`.

`link`→ Crea un enlace duro.

`unlink`→ Borra la entrada descrita en `dentry` del directorio con el inodo `inode`.

`symlink`→ Crea un enlace simbólico.

`mkdir`→ Crea un nuevo directorio.

`rmdir`→ Borra un directorio.

`mknod`→ Crea un archivo especial (archivo de dispositivo, tubería con nombre, socket).

`rename`→ Cambia el nombre de un archivo o directorio.

`readlink`→ Los enlaces simbólicos son archivos cuyo contenido es la ruta (realtiva o absoluta) de otro archivo. Esta función lee el contenido del archivo y lo copia en un búfer.

`follow_link`→ Sigue un enlace simbólico para determinar el inodo final.

`truncate`→ Trunca el tamaño del archivo.

`permission`→ Verifica si el modo de acceso especificado es permitido para el inodo.

`setxattr`→ Agrega un atributo extendido (con nombre arbitrario) al inodo.

`getxattr`→ Devuelve el atributo extendido solicitado del inodo.

`listxattr`→ Devuelve una lista de los atributos extendidos del inodo.

`removexattr`→ Remueve un atributo extendido del inodo.

E.6. `dentry`

Las estructuras de tipo `dentry` describen cada uno de los elementos de una ruta. Por ejemplo, en la ruta `/usr/bin/gcc`, cada uno de los elementos `/`, `usr`, `bin` y `gcc` es descrito por una estructura `dentry`.

```
struct dentry {
    atomic_t                    d_count;
    unsigned int                d_flags;
    spinlock_t                  d_lock;
    struct inode                *d_inode;

    struct dentry               *d_parent;
    struct qstr                 d_name;

    struct list_head            d_lru;
    struct list_head            d_child;
    struct list_head            d_subdirs;
    struct list_head            d_alias;
    struct dentry_operations    *d_op;
    struct super_block          *d_sb;
    void                        *d_fsdata;
```

```
    struct  rcu_head            d_rcu;
    struct  hlist_node          d_hash;
    int                         d_mounted;
};
```

`d_count`→ Cuenta de referencias.

`d_flags`→ Las estructuras `dentry` son mantenidas en una caché de las más recientes. La bandera `DCACHE_REFERENCED` indica si esta estructura ha sido referenciada recientemente y no debe ser eliminada de la caché.

`d_lock`→ Candado para el acceso a esta estructura.

`d_inode`→ Apuntador al inodo asociado.

`d_parent`→ Apuntador a la estructura `dentry` del directorio padre.

`d_name`→ Nombre del archivo o directorio (por ejemplo, `usr`).

`d_lru`→ Todos las estructuras de tipo `dentry` que no están siendo utilizadas (`d_count` es 0) son mantenidas en una lista LRU a través de este campo.

`d_child`→ Todos los hijos de un mismo directorio padre son mantenidos en una lista a través de este campo.

`d_subdirs`→ Las estructuras `dentry` que corresponden a un directorio contienen una lista de sus subdirectorios en este campo.

`d_alias`→ Lista de todas las estructuras `dentry` que son representadas por el mismo inodo que esta estructura.

`d_op`→ Apuntador a una estructura `dentry_operations` que contiene las operaciones implementadas por este sistema de archivos.

`d_sb`→ Superbloque asociado con este archivo.

`d_fsdata`→ Información específica del sistema de archivos.

`d_rcu`→ candado RCU [27].

`d_hash`→ Las estructuras `dentry` son mantenidas en una tablas hash. Las estructuras con el mismo valor hash se encuentran ligadas a través de este campo.

`d_mounted`→ Bandera que indica si esta entrada es un punto de montaje.

E.7. dentry_operations

La estructura `dentry_operations` contiene apuntadores a las funciones implementadas por este sistema de archivos.

```
struct dentry_operations {
    int  (*d_revalidate)(struct dentry *, struct nameidata *);
    int  (*d_hash) (struct dentry *, struct qstr *);
    int  (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
    int  (*d_delete)(struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_iput)(struct dentry *, struct inode *);
};
```

`d_revalidate`→ Esta función es llamada para determinar si una estructura `dentry` en la caché es aún válida.

`d_hash`→ Esta función calcula el valor hash de la estructura `dentry`

`d_compare`→ Esta función compara dos nombres de archivo o directorios para determinar si son iguales.

`d_delete`→ Esta función es llamada cuando `dentry`→`d_count` se hace 0 y debe indicar si debe eliminarse de la caché.

`d_release`→ Se llama cuando la estructura va a ser liberada (en el caso que fuera necesario realizar más operaciones).

`d_iput`→ Es llamada cuando el inodo asociado con esta estructura `dentry` es invalidado. Esto ocurre cuando el inodo asociado ha sido borrado del disco.

E.8. file

La estructura `file` contiene la información referente a un archivo abierto por un proceso. Si varios procesos han abierto el mismo archivo, cada proceso tiene su propia estructura `file`.

```
struct file {
    struct list_head          f_list;
    struct dentry             *f_dentry;
    struct vfsmount           *f_vfsmnt;
    struct file_operations    *f_op;
```

```
    atomic_t                    f_count;
    unsigned int                f_flags;
    mode_t                      f_mode;
    int                         f_error;
    loff_t                      f_pos;
    struct fown_struct          f_owner;
    unsigned int                f_uid, f_gid;
    struct file_ra_state        f_ra;
    unsigned long               f_version;

    void                        *private_data;

    struct list_head            f_ep_links;
    spinlock_t                  f_ep_lock;

    struct address_space        *f_mapping;
};
```

f_list→ Todos los archivos pertenecientes a un mismo superbloque son ligados a través de este campo.

f_dentry→ Apuntador a la estructura `dentry` asociada con este archivo.

f_vfsmnt→ Apuntador a la estructura `vfsmount` (ver mas adelante) asociada con este archivo.

f_op→ Apuntador a la estructura `file_operations` que contiene las funciones implementadas por este sistema de archivo.

f_count→ Cuenta de refencias.

f_flags→ Banderas especificadas al abrir el archivos (solo lectura, escritura, etc.)

f_mode→ Modo de acceso al archivo (lectura/escritura).

f_error→ Error en el archivo (no utilizado).

f_pos→ Posición actual en el archivo.

f_owner→ Proceso propietario del archivo (que recibirá las señales).

f_uid→ Identificador del usuario que abríó el archivo.

`f_gid`→ Identificador del grupo que abrió el archivo.

`f_ra`→ Para hacer mejor uso del disco duro, Linux lee varios bloques de un archivo por adelantado. Esta estructura contiene datos sobre cuántos bloques leer por adelantado y estadísticas sobre la eficiencia obtenida.

`f_version`→ Número de versión. Es incrementada en cada cambio a la estructura. Puede utilizarse para detectar cambios.

`private_data`→ Usado por los archivos de dispositivos que representan una terminal. En este campo se almacena la estructura relacionada con la terminal asociada al proceso dueño de la terminal.

`f_ep_links`→ Lista de todos los archivos monitoreados por la llamada a sistema `epoll()`.

`f_ep_lock`→ Candado para la lista anterior.

`f_mapping`→ Apuntador a la estructura `address_space` correspondiente al archivo. Esta estructura contiene las operaciones de bajo nivel para leer y escribir páginas del archivo.

E.9. file_operations

La estructura `file_operations` contiene las operaciones implementadas por el sistema de archivos. En esta estructura se encuentran las funciones que implementan la mayoría de las llamadas a sistema relacionadas con la manipulación de archivos (`open()`, `read()`, etc.). Los nombres de los apuntadores son idénticos a las llamadas a sistema que implementan.

```
struct file_operations {
    struct module *owner;
    loff_t  (*llseek) (struct file *, loff_t , int);
    ssize_t (*read)   (struct file *, char __user *,
                       size_t , loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *,
                         size_t , loff_t);
    ssize_t (*write)  (struct file *, const char __user *,
                      size_t , loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *,
                          size_t , loff_t);
    int     (*readdir) (struct file *, void *, filldir_t);
```

```
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int      (*ioctl) (struct inode *, struct file *, unsigned int,
                  unsigned long);
    int      (*mmap) (struct file *, struct vm_area_struct *);
    int      (*open) (struct inode *, struct file *);
    int      (*flush) (struct file *);
    int      (*release) (struct inode *, struct file *);
    int      (*fsync) (struct file *, struct dentry *, int datasync);
    int      (*aio_fsync) (struct kiocb *, int datasync);
    int      (*fasync) (int, struct file *, int);
    int      (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *,
                    unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *,
                     unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t,
                       read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *,
                       int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                                       unsigned long, unsigned long,
                                       unsigned long);
    int      (*flock) (struct file *, int, struct file_lock *);
};
```

owner→ Apuntador al módulo propietario de estas operaciones.

llseek→ Modifica la posición actual en el archivo (file→f_pos).

read→ Lee cierta cantidad de bytes del archivo.

aio_read→ Inicia una lectura asíncrona (con la llamada a sistema aio_read()).

write→ Escribe una cierta cantidad de bytes al archivo.

aio_write→ Inicia una escritura asíncrona (con la llamada a sistema aio_write())

readdir→ Lee la siguiente entrada de un directorio.

poll→ Espera (duerme) a que exista actividad (del tipo especificado) en el archivo.

ioctl→ Envía un comando a un dispositivo. Sólo para archivos que representan un dispositivo.

`mmap`→ Mapea a memoria un archivo.

`open`→ Abre el archivo y lo liga al inodo correspondiente.

`flush`→ Es llamada cuando `file→f_count` es decrementado.

`release`→ Es llamada cuando `file→f_count` se hace 0.

`fsync`→ Escribe todo el contenido del archivo que no ha sido escrito (está en caché) a disco.

`aio_fsync`→ Equivalente a la anterior función, pero lo realiza asíncronamente.

`lock`→ Adquiere un candado en el archivo (para implementar la llamada a sistema `fcntl()`).

`readv`→ Lee un vector de peticiones. Cada elemento del vector indica el búfer en donde se almacenarán los datos y la longitud de cada petición.

`writev`→ Escribe un vector de peticiones.

`sendfile`→ Copia datos de un archivo a otro. Esta copia se realiza dentro del núcleo directamente, por lo cual se evita hacer copias innecesarias.

`sendpage`→ Función utilizada por `sendfile()` para cada página copiada.

`get_unmapped_area`→ Función utilizada por archivos mapeados en memoria para localizar zonas no mapeadas.

`flock`→ Adquiere o remueve un candado del archivo.

E.10. namespace

La estructura `namespace` describe una jerarquía de puntos de montaje que permiten que cada proceso pueda tener una "vista" diferente de los sistemas de archivos montados en el sistema.

```
struct namespace {
    atomic_t             count;
    struct vfsmount *    root;
    struct list_head     list;
    struct rw_semaphore sem;
};
```

count→ Cuenta de referencias.

root→ Apuntador a la estructura `vfsmount` del directorio raíz de este `namespace`.

list→ Cabecera de la lista de estructuras `vfsmount` pertencientes a este `namespace`.

sem→ Semáforo para protección de modificaciones a esta estructura.

E.11. vfsmount

La estructura `vfsmount` describe un punto de montaje y su posición en la jerarquía de directorios.

```
struct vfsmount
{
    struct list_head    mnt_hash;
    struct vfsmount     *mnt_parent;
    struct dentry       *mnt_mountpoint;
    struct dentry       *mnt_root;
    struct super_block  *mnt_sb;
    struct list_head    mnt_mounts;
    struct list_head    mnt_child;
    atomic_t            mnt_count;
    int                 mnt_flags;
    int                 mnt_expiry_mark;
    char                *mnt_devname;
    struct list_head    mnt_list;
    struct list_head    mnt_fslink;
    struct namespace    *mnt_namespace;
};
```

mnt_hash→ Las estructuras `vfsmount` se mantienen en una tablas hash. Las estrcuturas `vfsmount` con el mismo valor hash se ligan a través de este campo.

mnt_parent→ Apuntador a la estructura `vfsmount` del directorio padre.

mnt_mountpoint→ Apuntador a la estructura `dentry` que describe el punto de montaje.

mnt_root→ Apuntador a la estructura `dentry` que describe el directorio raíz de este sistema de archivos.

mnt_sb→ Apuntador al superbloque.

`mnt_mounts`→ Lista de las estructuras `vfsmount` que son hijas de esta estructura.

`mnt_child`→ Este campo es el utilizado por las estructuras `vfsmount` “hermanas” de esta estructura.

`mnt_count`→ Cuenta de referencias.

`mnt_flags`→ Banderas de montaje.

`mnt_expiry_mark`→ Bandera que indica que este montaje es expirable (es decir, que se desmonte después de cierto tiempo sin ser usado).

`mnt_devname`→ Nombre del dispositivo que contiene este punto de montaje.

`mnt_list`→ Todas las estructuras `vfsmount` que pertenecen a un mismo `namespace` son mantenidas en una lista a través de este campo.

`mnt_fslink`→ Lista de puntos de montaje expirables.

`mnt_namespace`→ Estructura `namespace` que contiene este punto de montaje.

Bibliografía

[1] *Bochs ia-32 emulator.* `http://bochs.sourceforge.net/`.

[2] *Linux test project.* `http://ltp.sourceforge.net/`.

[3] *Qemu cpu emulator.* `http://fabrice.bellard.free.fr/qemu/`.

[4] Benjamin Atkin and Emin Gün Sirer, *Portos: an educational operating system for the post-pc environment*, Proceedings of the 33rd SIGCSE technical symposium on Computer science education, ACM Press, 2002, pp. 116–120.

[5] Rudolf Bayer, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Informatica **1** (1972), 290–306.

[6] Jeff Bonwick, *The slab allocator: An object-caching kernel memory allocator*, USENIX Summer, 1994, pp. 87–98.

[7] Daniel P. Bovet and Marco Cesati, *A real bottom-up operating systems course*, SIGOPS Oper. Syst. Rev. **35** (2001), no. 1, 48–60.

[8] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson, *The nachos instructional operating system*, Tech. report, 1993.

[9] Jorge Buenabad Chávez, *Xinix time-sharing operating system*, SIGOPS Oper. Syst. Rev. **25** (1991), no. 4, 22–34.

[10] Douglas Comer, *Operating system design, the XINU approach*, Prentice-Hall publishers, 1984.

[11] Peter J. Denning, *Virtual memory*, ACM Comput. Surv. **2** (1970), no. 3, 153–189.

[12] John Fotheringham, *Dynamic storage allocation in the atlas computer, including an automatic use of a backing store*, Commun. ACM **4** (1961), no. 10, 435–436.

[13] Inc Free Software Foundation, *Gnu general public license.* `http://www.gnu.org/copyleft/gpl.html`, 1991.

[14] David A. Holland, Ada T. Lim, and Margo I. Seltzer, *A new instructional operating system*, Proceedings of the 33rd SIGCSE technical symposium on Computer science education, ACM Press, 2002, pp. 111–115.

[15] David Hovemeyer, Jeffrey K. Hollingsworth, and Bobby Bhattacharjee, *Running on the bare metal with geekos*, Proceedings of the 35th SIGCSE technical symposium on Computer science education, ACM Press, 2004, pp. 315–319.

[16] IEEE, *IEEE std 1003.1-2001 standard for information technology — portable operating system interface (POSIX), issue 6*, IEEE, New York, NY, USA, 2001, Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[17] Sitaram Iyer and Peter Druschel, *Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o*, SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, 2001, pp. 117–130.

[18] *The linux kernel source.* *http://www.kernel.org*.

[19] Kernighan and Ritchie, *The C programming language*, Prentice Hall, Englewood Cliffs, 1978.

[20] T. Kilburn, D. Edwards, M. Lanigan, and F. Sumner, *One-level storage system*, IRE Transactions Elect. Computers **37** (1962), 223–235.

[21] Kenneth C. Knowlton, *A fast storage allocator*, Commun. ACM **8** (1965), no. 10, 623–624.

[22] Donald E. Knuth, *The art of computer programming: Fundamental algorithms*, 2 ed., vol. 1, Addison-Wesley Publishing Co., Reading, Massachusetts, 1973.

[23] John Lions, *Lions comentary on unix 6th edition with source code*, Peer-to-Peer Communications, 1996.

[24] Kwan Lowe, *Kernel rebuild guide.* *http://www.digitalhermit.com/linux/Kernel-Build-HOWTO.html*, 2004.

[25] Paul E. McKenney, *Stochastic fairness queuing*, IEEE INFOCOM'90 Proceedings (San Francisco), The Institute of Electrical and Electronics Engineers, Inc., June 1990, pp. 733–740.

[26] ______, *Stochastic fairness queuing*, Internetworking: Theory and Experience **2** (1991), 113–131.

[27] Paul E. McKenney and John D. Slingwine, *Read-copy update: Using execution history to solve concurrency problems*, Parallel and Distributed Computing and Systems (Las Vegas, NV), October 1998, pp. 509–518.

[28] Jake Moilanen, *Genetic library. `http://kernel.jakem.net/`*, 2005.

[29] Jason Nieh and Chris Vaill, *Experiences teaching operating systems using virtual platforms and linux*, SIGCSE Bull. **37** (2005), no. 1, 520–524.

[30] CORPORATE The Joint Task Force on Computing Curricula, *Computing curricula 2001*, J. Educ. Resour. Comput. **1** (2001), no. 3es, 1.

[31] Jerome Pinot, *Supported architectures for linux. `http://ngc891.blogdns.net/kernel/docs/arch.txt`*, 2004.

[32] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language*, Comp. Sci. Tech. Rep. No. 31, Bell Laboratories, Murray Hill, New Jersey, October 1975, 31 Superseded by B. W. Kernighan and D. M. Ritchie, *The C Programming Language* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[33] Dennis M. Ritchie and Ken Thompson, *The unix time-sharing system*, Commun. ACM **17** (1974), no. 7, 365–375.

[34] Abraham Silberschatz and Peter B. Galvin, *Operating System Concepts*, 4. ed., Addison Wesley, 1994.

[35] William Stallings, *Operating systems*, Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1992.

[36] Richard Stallman, *Linux and the gnu project. `http://www.gnu.org/gnu/linux-and-gnu.html`*, 1997.

[37] A. S. Tanenbaum, *Operating systems: Design and implementation*, Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1987.

[38] David Woodhouse, *Jffs: The journalling flash file system*, Ottawa Linux Symposium, RedHat Inc., 2001.

[39] Su Yun-Lin, *On teaching operating systems*, SIGCSE Bull. **21** (1989), no. 3, 11–14.

Índice alfabético

CURRENT_BONUS, 23
CURRENT_TIME, 89
DBNULL, 80
IBNULL, 81
MAX_BONUS, 23
MAX_SLEEP_AVG, 23
PAGE_OFFSET, 41
PF_BORROWED_MM, 120
PF_DEAD, 119
PF_DUMPCORE, 119
PF_EXITING, 119
PF_FLUSHER, 120
PF_FORKNOEXEC, 119
PF_KSWAPD, 120
PF_MEMALLOC, 119
PF_MEMDIE, 120
PF_SIGNALED, 119
PF_STARTING, 119
PF_SUPERPRIV, 119
PF_SYNCWRITE, 120
PGID, 26
PID, 26
SID, 26
SIFTTOU, 18
SIGCONT, 18
SIGSTOP, 18
SIGTSTP, 18
SIGTTIN, 18
TASK_INTERRUPTIBLE, 18, 119
TASK_RUNNING, 17–20, 119
TASK_STOPPED, 18, 119
TASK_UNINTERRUPTIBLE, 18, 119
TASK_ZOMBIE, 18, 119
TGID, 26
XINUFS_BSIZE, 85
XINUFS_MAGIC, 86
XINUFS_ROOT_INO, 88
XINUFS_SB, 85
ZONE_DMA, 36
ZONE_HIGHMEM, 36
ZONE_NORMAL, 36, 38
active_mm, 25
active, 28
alloc_pages(), 42
bio_vec, 58, 59, 132
bio, 56, 58, 59, 131
block_device, 130
block_prepare_write(), 97
block_read_full_page(), 94
buffer_head, 56–58, 95, 129
calibrate_delay(), 30
chroot(), 28
clone(), 32, 34
close(), 53
container_of(), 87
contig_page_data, 36
copy_process(), 34
core_dump, 26
current, 89
dcache_init(), 76

dentry_operations, 144
dentry, 28, 74, 76, 142
do_fork(), 32
effective_prio(), 22
exec(), 27
expired, 19, 28
file_operations, 74, 91, 94, 146
file_system_type, 73, 81, 82, 133
files_cache, 31
files_init(), 76
files_struct, 28, 31, 74
file, 74, 144
fork(), 32, 34
fork_init(), 30
free_area, 41
free_pages(), 42
fs_cache, 31
fs_struct, 27, 31
generic_commit_write(), 96
generic_file_read(), 94
generic_file_write(), 96
generic_read_dir(), 91
getpid(), 26
iget(), 86
init_bootmem(), 28
init_idle(), 28
init_mm, 28
init_xinufs_fs(), 82
inode_init(), 76
inode_operations, 74, 92, 141
inode, 74, 76, 87, 137
kmem_cache_init(), 45
kmem_cache_t, 43, 45
linux_binfmt, 25
load_binary, 26
load_shlib, 26
loops_per_jiffy, 30
lseek(), 53, 55
map_bh(), 95
mem_init(), 28, 41
mm_struct, 25, 28, 31, 47
mm, 25
mnt_init(), 76
namespace, 148
open(), 53
pagetable_init(), 40
page, 59
pg_data_t, 36
pgdat_list, 36, 49
pid_hash, 30
pidhash_init(), 28
pidmap_init(), 30
prio_array_t, 19, 20
prio_array, 117
prio, 22
proc_caches_init(), 30
queue, 28
read(), 53
register_filesystem(), 82, 83
rest_init(), 31
runqueue_t, 19, 28
runqueue, 117
sb_bread(), 85
sched_init(), 28
setup_arch(), 28, 37
setup_memory(), 28, 37
sighand_cache, 30
sighand_struct, 30
signal_cache, 30
signal_struct, 30
start_kernel(), 28, 37, 74
static_prio, 21
struct page, 36
super_block, 73, 83, 134
super_operations, 73, 86, 136
task_struct, 21, 22, 24, 25, 30, 119
task_timeslice(), 23
time_init(), 30

timer_interrupt(), 32
total_forks, 34
vfork(), 32, 34
vfs_caches_init(), 76
vfs_caches_init_early(), 74
vfsmount, 28, 74, 149
vm_area_struct, 31, 46
wait4(), 18
write(), 53
xinufs_alloc_datablock(), 95, 97
xinufs_create(), 93
xinufs_find_datablock(), 95
xinufs_get_block(), 94, 95, 97
xinufs_get_first_iblk(), 89, 90
xinufs_get_iblk(), 90
xinufs_inode_by_name(), 93
xinufs_inode_info, 86
xinufs_link(), 94
xinufs_lookup(), 92
xinufs_mknod(), 93
xinufs_new_inode(), 94
xinufs_prepare_write(), 96, 97
xinufs_read_inode(), 87
xinufs_readdir(), 91
xinufs_readpage(), 94
xinufs_sb_info, 85
xinufs_sbi(), 85
xtime, 30, 89
zone_sizes_init(), 40
zone, 36

kbuild, 10