VNiVERSiDAD
D SALAMANCA

JOS OPERATING SYSTEM

JOSÉ SÁNCHEZ SÁNCHEZ

jose@ssanchez.net

Dedicado a mi mujer, Cecilia, por la paciencia que tuvo en el tiempo que este proyecto me robó. Y también a mis ex-compañeros de France Telecom España, Bernardo, Ferran e Iván, por el tiempo que pasamos arreglando problemas extraños.

Índice general

Capítulo 1

Introducción

1.1. Objetivos

Este proyecto empezó como un experimento (inspirado por [Mol]) de hacer un cargador que imprimiese en la pantalla de un PC.

Ante la imposibilidad de entender el código de la referencia citada, se procedió a investigar sobre la pila de llamadas del lenguaje C, proceso que resulta fundamental para cualquier software complejo que se quiera construir con dicho lenguaje (compilador, sistema operativo u otro código con gestión avanzada de memoria).

Entender el código no es suficiente para generar algo similar, por lo que se procedió a investigar sobre el lenguaje ensamblador (concretamente con la sintaxis AT&T y usando el libro [Blu05]).

Pero la verdadera dificultad del lenguaje ensamblador no es entender su sintaxis o funcionamiento (algo simple dado el bajo nivel de sus instrucciones), sino que lo verdaderamente difícil es conocer el procesador.

Los manuales oficiales de los procesadores x86 son complejos para un principiante por lo que antes de poder entenderlos fue necesario echar mano de libros más explicativos (como [Mes95], [MS96], [MHJM] entre otros).

Con estos conocimientos ya estaban las piezas pero para juntarlas fue necesario investigar sobre el enlazador (o "*linker*") y en menor medida sobre el "*toolchain*" o conjunto de herramientas que nos ayudan a plasmar el código.

El objetivo inicial se consiguió en pocas horas y con poco más de cien lineas de código (entre código ensamblador, C, script del linker y fichero "*make*") y debido pre-

cisamente a la facilidad de este primer paso, se pensó en hacer un controlador de video más avanzado.

De nuevo fue sencillo y como el momento coincidió con el auge de los procesadores de 64 bits, se pensó en poner la CPU en ese modo e investigar sobre las diferencias entre 32 y 64 bits.

Este proceso fue relativamente complejo (como veremos, debido al cambio de tabla de páginas), pero sirvió para seguir investigando, aprendiendo y leyendo sobre las distintas partes del sistema, y una vez superado para pensar que un sistema operativo es, con sus peculiaridades y dificultades añadidas, un programa como otro cualquiera y que con esfuerzo y dedicación puede llevarse a buen puerto, además de para valorar extender el proyecto hasta algo más funcional.

Los objetivos iniciales de este trabajo eran:

- Comprender en profundidad como funciona un sistema operativo y en que momento y como usa el procesador.
- Ampliar los conocimientos de la arquitectura x86-64.
- Conocer como funcionan las herramientas que se nos proporcionan para programar software, para enlazarlo, para cargarlo, formatos binarios, estándares, etcétera.
- Particularizando un poco más, comprender el sistema de memoria de un sistema operativo así como otros subsistemas que pudieran resultar útiles a la hora de optimizar la ejecución de un sistema operativo real.
- Conocer algunos dispositivos reales ya que incluso los más complejos funcionan conceptualmente igual que los dispositivos simples.

Como veremos, estos objetivos se han cumplido e incluso se ha llegado mucho más lejos. En todo caso, desde el principio no se establecieron límites sino que se pensó en un diseño incremental que permitiera ir consiguiendo hitos.

Este documento presenta una descripción temporal de estos hitos así como discusiones y justificaciones de las decisiones tomadas.

1.2. Sistemas operativos

Antes de empezar con ningún detalle es importante tener claro que es o que se entiende por sistema operativo y que límites se establecen y en que consiste este trabajo.

La mayoría de definiciones que encontramos de un sistema operativo nos dicen que es un programa que gestiona el *hardware* y provee servicios a las aplicaciones de usuario. Sistema operativo también puede incluir el software de sistema, que son las aplicaciones necesarias para que este software tenga alguna funcionalidad visible por el usuario: entorno de ventanas, configuraciones, etcétera.

No suele haber problema en distinguir ambos significados, en todo caso podemos referirnos al primero como núcleo o kernel para evitar ambigüedad.

A partir de ahora vamos a centrarnos solamente en el núcleo, ya que el resto del software queda fuera de este trabajo. En ocasiones puntuales será necesario volver a las aplicaciones, ya que una de las funciones del kernel es controlar la ejecución de éstas. Desde nuestro punto de vista será indiferente que una aplicación sea de sistema o de usuario y nuestro núcleo trabajará con un concepto más simple: el proceso.

Hacer un kernel es una tarea mastodóntica para la que se necesitan muchos y buenos programadores trabajando durante años. Evidentemente el tamaño y complejidad dependerá de lo que intentemos hacer; no es lo mismo programar un sistema para un dispositivo especializado cuyas piezas elegimos nosotros (imaginemos un juguete o un electrodoméstico), que para un PC que tendrá dispositivos conectados de lo más variopintos.

En todo caso, es una empresa más compleja que hacer una aplicación cualquiera por las siguientes razones:

- Los sistemas operativos se hacen usando lenguajes de programación poco abstractos y no tan productivos. Mientras que para una aplicación de usuario puede usarse Java o C++, para un sistema operativo se suele usar C o ensamblador. La diferencia desde el punto de vista de la productividad, es que podemos necesitar varias lineas de C (y más aún de ensamblador) para tener la misma funcionalidad que tiene una linea de Java.

- Es difícil reusar código[1], y no en cuanto a que podamos copiarlo de otro sitio, sino en cuanto a que tenemos que hacer absolutamente todo. Por ejemplo, una aplicación de usuario puede imprimir por pantalla usando una única linea de código, mientras, un kernel, puede necesitar como mínimo varios cientos de lineas (en cualquier sistema operativo de los que usamos en nuestro día a día serán muchas más).

- Los problemas pueden llegar a ser extremadamente difíciles de reproducir y encontrar. Una aplicación de usuario es relativamente determinista, es decir, entre

[1] Como veremos, el código es la forma hacer o escribir software.

una invocación y la siguiente, las condiciones son casi las mismas, pero un sistema operativo no se comporta así. En la práctica es imposible que un usuario lance las aplicaciones en el mismo orden, con el mismo lapso de tiempo, con el ordenador a la misma temperatura, etcétera. Si el orden en el que se realizan las acciones cambia el comportamiento del sistema de forma no esperada, se genera una condición de carrera (del inglés *race condition*).

- Es difícil de probar porque cada vez que queramos ver como se comporta un trozo de código nuevo, tendremos que copiar el software de forma que funcione (ya que al encenderlo, el ordenador tiene que ser capaz de encontrar la imagen) y reiniciar el equipo dos veces (una para probarlo y otra para poder volver a copiar de nuevo la siguiente versión que queramos probar). Este problema se mitiga con los emuladores. Un emulador, en este contexto no es más que un programa que se comporta como si fuera un ordenador completo, con la peculiaridad de que no es un entorno 100 % real ya que al ser una aplicación como otra cualquiera, es, como dijimos, determinista (pseudodeterminista en realidad ya que cosas como las acciones de los usuarios, pulsaciones de teclado por ejemplo, son no deterministas).

- Los problemas no se pueden buscar mediante otras herramientas. Al hacer un sistema operativo no habrá aplicaciones corriendo al mismo tiempo así que no podremos usar un depurador para ver el estado de las variables ni ninguna otra herramienta parecida. Lo único que podremos hacer será correr el sistema operativo instrucción a instrucción desde el emulador (si es que el problema se manifiesta en el emulador) y tal vez programar algún mecanismo para inspeccionar el código mediante otro ordenador o terminal conectados, pero esta funcionalidad tendríamos que implementarla nosotros por lo que a su vez el desarrollo sufriría de los mismos problemas.

- Existen sutilezas y defectos en los ordenadores que hacen perder tiempo al programador. A veces hay que poner una instrucción que no haga nada entre dos instrucciones para dar tiempo a que el dispositivo que la recibe se prepare para recibir la siguiente, otras, un componente no funciona como viene documentado o bien existen diferencias entre distintos equipos que hacen extremadamente complicado que un trozo de código funcione en todos.

- Aunque ¡OS carece de multiprogramación, pueden darse otro tipo de problemas como la reordenación de instrucciones por parte del compilador o del procesador, la coherencia de cachés, y otros mecanismos de muy bajo nivel que pueden afectar a la ejecución y generar problemas misteriosos.

En este trabajo vamos a intentar explicar como construir un kernel simple, que use las piezas más comunes de un ordenador estándar y moderno y que funcione en un emulador. Además se intentará que quede preparado para que en el futuro pueda hacerse funcionar en un equipo físico y para que se pueda extender tanto como se quiera.

1.2.1. Funciones

Ya hemos definido kernel, pero ¿para qué se necesita uno? Estas son algunas de las razones.

- Cuando encendemos un equipo o conectamos algún dispositivo, ¿quién habría de hacer su inicialización? Podríamos dejar que cada aplicación inicializara cada dispositivo que usara (y lo dejara en un estado consistente al terminar de usarlo), pero sería costoso (incluso prohibitivo) que los programadores tuvieran que conocer cada dispositivo existente en el mercado, además de que se darían situaciones desagradables como tener que actualizar todo nuestro software al comprar un dispositivo nuevo. Otra opción podría ser, usar código compartido, pero esto nos lleva al siguiente punto.

- Multitarea. Las aplicaciones no usan simultáneamente todos los recursos disponibles en un sistema. Por ejemplo, un procesador de texto podría estar esperando una pulsación del usuario mientras otro programa descarga un archivo de Internet y otro reproduce un archivo multimedia. Parece razonable pensar que unos programas usen los recursos que no están siendo usados, y dado que un ordenador moderno ejecuta miles de millones de instrucciones en un segundo, el usuario tendrá la impresión de que las aplicaciones se ejecutan concurrentemente. De nuevo, sin un sistema operativo, la única forma de controlar esto sería obligando a las aplicaciones a cooperar y es imposible conseguir que algo así funcione bien (como ejemplo tenemos los programas TSR de MSDOS).

- Reparto y optimización de recursos. Si podemos ejecutar varias aplicaciones simultáneamente, antes o después (desafortunadamente en el software se cumple la ley de Murphy) más de una aplicación querrá usar un recurso al mismo tiempo. Por ejemplo si escribimos por teclado, ¿qué aplicación debe recibir la pulsación? Si llega un paquete de red ¿para qué aplicación es? Además, el sistema operativo va un paso más allá y puede optimizar estos recursos. Por ejemplo, si ejecutamos un mismo programa varias veces, el código puede compartirse en las mismas celdas de memoria (mediante la memoria virtual), o si varios programas intentan

escribir a disco al mismo tiempo, las escrituras pueden reordenarse y mejorar por tanto el rendimiento.

- Políticas globales. Sin un sistema operativo cada cambio que quisiéramos hacer habría de aplicarse en cada aplicación. Por ejemplo, tal vez queramos limitar la velocidad de uso de la red para no molestar al resto de usuarios conectados. Es imposible modificar las aplicaciones para adaptarlas a cada situación y para hacer un uso óptimo unas tendrían que saber el uso de red de las otras.

Para hacernos una idea aproximada del lugar que ocupa el sistema operativo, podemos ver la figura 1.1.

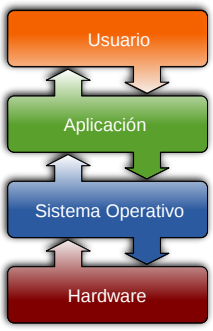

Figura 1.1: Capas del sistema

Las aplicaciones no pueden acceder al hardware directamente sino que solicitan los servicios necesarios al sistema operativo mediante las llamadas al sistema.

Es habitual tener llamadas al sistema para abrir un archivo, para leerlo, para imprimir por pantalla, para leer de teclado, para enviar datos por la red o impresora... Pero todo dependerá de la implementación. Puede haber un diseño con una llamada al sistema para imprimir un carácter y otro que permita imprimir una cadena. En el primer caso, si quisiéramos imprimir una cadena, tendríamos que llamarla repetidamente y en el segundo, con una sola llamada tendríamos la cadena en la pantalla.

Hay que destacar que este proceso es transparente para el programador puesto que este trabaja con las funciones de biblioteca . Estas funciones sirven para tener una funcionalidad reusable que se apoya a su vez en las llamadas al sistema o en otras funciones para ofrecer algo útil al programador. Una biblioteca simplemente es un conjunto de funciones agrupado por funcionalidad. Un sistema operativo moderno, por ejemplo Linux, que es el sistema que más inspira a ¡OS, cuenta con unos pocos cientos de estas llamadas al sistema y en cuanto a funciones de biblioteca, depende de cada instalación pero en una instalación estándar puede haber decenas de miles.

Por el otro lado, el sistema operativo se comunica con el hardware, y como podemos ver en el sentido de las flechas, el hardware con el sistema operativo. El primer caso se dará cada vez que requiramos inicializar un dispositivo o cuando necesitemos un servicio del hardware (escribir en disco, imprimir un archivo). El segundo cuando el hardware quiera avisar de algo, por ejemplo cuando ya haya leído ese bloque o cuando

la impresora se quede sin papel.

Esta separación en capas proporciona la ventaja de que el interfaz es el mismo independientemente del dispositivo con el que trabajemos. Podemos tener varios modelos de tarjetas de red y hacer que las aplicaciones trabajen con un interfaz común que no depende del hardware que estemos usando. Otra ventaja es que el sistema queda dividido en piezas más manejables que pueden ser repartidas entre equipos de programadores.

Para que este sistema de capas funcione es necesario que el hardware ayude de alguna forma. Si un proceso pudiera escribir en cualquier parte de la memoria, éste podría acceder a los datos o instrucciones de cualquier otro proceso, o si pudiera ejecutar cualquier instrucción tendría acceso al hardware sin control alguno y podría leer datos de otros procesos o del núcleo o hacer que el equipo dejara de funcionar o se apagara.

Para terminar, es difícil hablar de un sistema operativo sin mencionar el concepto de proceso, ya que el proceso es la abstracción principal para gestionar las entidades de ejecución. Podemos entender por proceso una instancia de ejecución de un programa, aunque hoy en día no es tan sencillo de definir ya que los distintos procesos pueden compartir más o menos elementos. Podemos ver la ayuda de la llamada al sistema clone(2) en Linux para tener información sobre lo que ese sistema permite compartir entre procesos relacionados. Es fácil comprender este concepto asociando un proceso con un programa, en la mayoría de casos es así, aunque algunos programas usan varios procesos para funcionar.

1.3. CPU

El recurso principal que gestiona un sistema operativo, así como la pieza sobre la que se construye un ordenador, es la CPU o procesador.

Simplificando su definición, un procesador es un dispositivo electrónico que dada una entrada y un estado produce una salida. En realidad podemos extender esta definición para cubrir cualquier programa informático; la "Máquina de Turing" desarrollada por Alan Turing teoriza estos conceptos.

Una entrada, si es válida, consiste en un dato o en cualquiera de las instrucciones soportadas junto con sus parámetros. El estado es el contenido que haya en sus registros y en memoria [2] y la salida es el efecto producido por esas instrucciones.

Los registros son unas pequeñas y rápidas memorias internas a las que cada proce-

[2] Aunque la memoria es ajena al procesador, muchos procesadores guardan estructuras en ella.

sador tiene acceso rápido y directo. La mayoría de estos registros tienen un tamaño que coincide con la capacidad del procesador de mover datos simultáneamente. Este dato se conoce como "tamaño de palabra" o directamente "palabra" (del inglés *word*).

Dado que el procesador no puede leer y escribir a la vez en memoria, muchas instrucciones trabajan sólo con registros, y si queremos hacer algo tan fácil como "sumar uno en dirección de memoria X", necesitaremos tres instrucciones: "trae la dirección de memoria X al registro Y", "suma uno al registro Y", "lleva el registro Y a la dirección de memoria X". Este es un ejemplo, y si se necesitarán más o menos instrucciones dependerá del procesador.

No todos los registros son iguales, hay instrucciones que sólo operan sobre algunos, otros son para controlar el estado de la CPU o para marcar la siguiente instrucción. Hay también registros específicos para números con decimales, para apuntar a la pila (como en seguida veremos, se trata de una estructura de datos) o para uso del sistema operativo.

Los procesadores también tienen registros que apuntan a direcciones de memoria donde se han guardado estructuras de datos que usa el procesador de forma transparente al programador.

El conjunto de registros, de instrucciones soportadas por un procesador, así como sus parámetros se conoce como la arquitectura del procesador. Como los procesadores nuevos añaden registros e instrucciones, cualquier software que haga uso de ellas perderá la compatibilidad. En la práctica esto no supone problema ya que el programador (o el compilador en la mayoría de casos) puede detectar si el procesador soporta cualquier característica extendida antes de usarla.

La arquitectura más famosa y la usada en los PCs, es la arquitectura x86. En los últimos años se ha reemplazado por la x86-64 que es una extensión para soportar 64 bits de memoria. Otro ejemplo de arquitectura muy usada es ARM, que se suele usar para teléfonos, tablets y routers por ejemplo de los que tenemos en casa para conectarnos a Internet.

1.4. Memoria y dispositivos

Es más costoso hacer memorias rápidas que memorias lentas, así que en cada ordenador se incluyen muchos tipos de memorias. Cuánto más rápidas son, menos capacidad tienen y cuanto más capacidad, se tarda más en acceder a ellas.

Además hay otras características que pueden tener: unas son volátiles (es decir, su

contenido se pierde cuando apagamos el sistema), otras están asociadas a memorias más lentas y pueden tener sólo ciertos contenidos de éstas (estas se llaman memorias caché asociativas).

Ordenadas de mayor a menor velocidad tenemos: registros de la CPU, caché del procesador (suele haber varios niveles: primer nivel, segundo y hasta tercero), RAM, memoria flash (como la memoria Robson de Intel[3]), discos duros, unidades ópticas, cintas, una red (aunque la red ha avanzado posiciones con el tiempo). Hay que tener en cuenta que un registro es cientos de veces más rápido que la RAM y esta a su vez miles de veces más lenta que el disco (si hacemos cuentas, podemos aproximadamente leer un registro un millón de veces en lo que leemos un bloque de disco). Estos datos varían en función de la tecnología que se use, pero una diferencia tan grande es significativa a la hora de diseñar el sistema.

Desde el punto de vista del sistema operativo, los registros son más o menos transparentes ya que al usar C, el compilador (programa que traduce nuestro código a código comprensible por el procesador) se encarga de elegirlos; no obstante, algunas operaciones no están disponibles desde el compilador y otras necesitan registros específicos, por lo que en el caso de un núcleo tendremos que operarlos manualmente.

La memoria RAM existente se puede seleccionar mediante un número que representa su dirección, y que estará comprendido entre cero y el tamaño de memoria que tengamos instalada; a esto se le conoce como memoria física. Además este número, en algunas arquitecturas (y en las que no lo requieren, como x86, conviene hacerlo así para mejorar el rendimiento), ha de ser un múltiplo del tamaño de palabra. La razón es que los datos se leen de memoria en palabras y si leemos por ejemplo 32 bits a partir de la dirección 1 (en un procesador de 32 bits), se necesitan dos lecturas, una para los tres primeros bytes (lee bytes 0-3) y otra para el último (lee bytes 4-7).

Pero un sistema operativo moderno no trabaja directamente con esta numeración física (salvo cuando quiera manipular precisamente la memoria física pero estas operaciones suelen abstraerse en funciones de bajo nivel) sino que lo hace con la memoria virtual. Conceptualmente es algo simple: cuando seleccionamos una dirección de memoria ésta no llega tal cual al dispositivo sino que se hace una traducción por el camino.

Esta traducción la hace otro dispositivo: MMU (del inglés: *Memory Management Unit*, ver imagen 1.2), por lo general mediante unas tablas.

Las cachés también usan direcciones virtuales porque se encuentran entre la CPU y la MMU. Comparando con la memoria RAM, las cachés son pequeñas (unos pocos KB para las de nivel 1 y unos pocos MB para las de nivel 2 y 3) y funcionan, casi siempre,

[3] `http://en.wikipedia.org/wiki/Intel_Turbo_Memory`

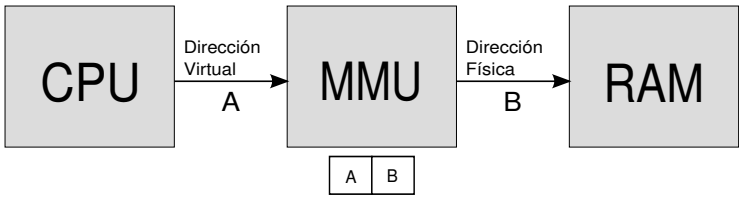

Figura 1.2: Direcciones de memoria

de forma transparente al programador. La razón de que esta transparencia no ocurra siempre, es que el sistema operativo puede cambiar en cualquier momento la tabla de traducción de direcciones en uso, y cuando la cambia, ha de invalidar las cachés puesto que una tabla nueva significa que las direcciones virtuales pasan a traducirse de forma diferente.

Normalmente cada proceso tiene una tabla distinta y por tanto una visión de que tiene a su disposición tanta memoria como le permite la arquitectura (2 elevado al tamaño de palabra, es decir 2^{32} en un equipo de 32 bits y 2^{64} en un equipo de 64 bits). Pero en realidad no es así porque esa dirección lógica se traduce a una dirección física y la memoria instalada en el equipo puede ser (y normalmente es) menor que el máximo admitido por la arquitectura.

Si que suele darse el caso de que la suma de las cantidades de memoria usadas por todas las aplicaciones superen a la memoria instalada en el equipo, al menos en algún momento. Lo que se hace es llevar las partes de la memoria menos usadas a disco. Antiguamente se llevaban a disco los procesos enteros (ver [Bac86]), pero este mecanismo de llevar solamente las páginas es más moderno y se conoce como "paginación bajo demanda" (del inglés: *"on demand paging"*, aunque hay que tener en cuenta que este concepto se refiere a algo más amplio ya que incluye la carga de páginas al leer procesos de disco).

Este disco duro si que puede ser mayor que el espacio virtual de un proceso. Por tanto podríamos usar, en teoría, un programa que necesitara más memoria de la que tenemos. En la práctica, como vimos, un disco es mucho más lento que una memoria RAM, y como el procesador no puede trabajar directamente con el disco sino que tiene que llevar lo que quiera leer a memoria, es posible que el sistema operativo pase más tiempo llevando datos de memoria a disco (para hacer hueco) y trayendo datos de disco a memoria (para trabajar con ellos), que ejecutando el programa[4].

Ahora imaginemos como es esa tabla que contiene las traducciones. En la informática, una tabla es una estructura que se accede por desplazamiento (del inglés:

[4]Este fenómeno se conoce como *"trashing"* lo que se puede traducir como "hiperpaginación".

"offset"), es decir, una colección de elementos del mismo tamaño situados uno tras otro y un número que indica la posición. De esta forma sabemos que el quinto elemento en una tabla cuyos elementos ocupan cuatro bytes, estará en el byte 16.

Si tradujéramos las direcciones por bytes esta tabla tendría que ser incluso más grande que la memoria (por cada byte tendríamos que tener 4 bytes para especificar la dirección física de ese byte). Por palabras necesitaríamos tanta memoria como quisiéramos traducir y recordemos que podremos tener varias tablas, por lo que este proceso no sería viable.

La solución es traducir por trozos llamados páginas. El tamaño de estos trozos es dependiente del procesador, incluso un mismo procesador puede tener varios tamaños de páginas en uso simultáneamente.

Pero echemos cuentas. Un tamaño común para estas páginas es 4096 bytes (2^{12}), o dicho de otra forma, los últimos 12 bits de la dirección a cero y el resto para representar el número de página. ¿cuánto ocuparía una tabla para un proceso en un equipo de 32 bits? La respuesta es: $32-12=20$ bits que se pueden combinar para formar un número de página, y a 4 bytes por traducción, serían 4MB.

A día de hoy parece asumible, pero este sistema estaba en uso cuando los PCs tenían en torno a esa memoria, además con el cambio a 64 bits el espacio virtual es enorme y este proceso se vuelve inviable. ¿Cómo podríamos reducir la memoria usada para las tablas de páginas?

Los procesos que corren en un equipo moderno usan una cantidad de memoria que se mide generalmente en megabytes. Además esa memoria suele ser relativamente contigua. Pensemos por un momento en que la usan; la respuesta es en código y datos y ambas cosas pueden colocarse juntas entre sí. La formalización de este concepto se conoce como principio de localidad[5].

Lo que se hace es dividir jerárquicamente esas direcciones, y dejar en blanco las partes no usadas.

Imaginemos que tenemos (y vamos a trabajar en decimal por comodidad), una tabla con mil elementos, del 0 al 999, pero estamos usando nada más 3 de ellos. Ver figura 1.3.

En la parte izquierda se muestran algunos elementos, pero de 3 que hay en uso, en memoria tendríamos 1000 ocupando espacio.

En la parte de la derecha se muestra el ejemplo con tablas jerarquizadas. Todos los que tenemos empiezan con 1, luego si reservamos una tabla de 10 elementos, podemos

[5] `http://en.wikipedia.org/wiki/Locality_of_reference`

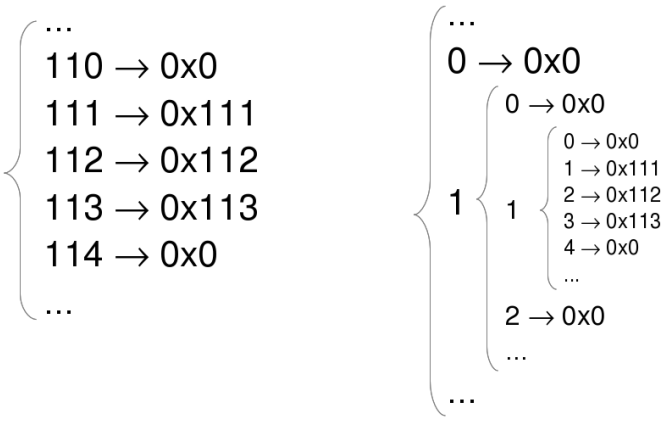

Figura 1.3: Tablas de páginas conceptuales

dejar todos vacíos menos el segundo (el primero sería para los que empezaran por cero). En esa casilla pondríamos una referencia a otra tabla con 10 elementos, la cual a su vez describiría los elementos 100 a 199, es decir, una decena por cada elemento mediante una referencia a un tercer nivel. La tabla de este tercer y último nivel, tendría, ahora sí, las direcciones físicas en los elementos 1, 2 y 3.

Por tanto hemos pasado de necesitar una tabla con 1000 elementos, a tener 3 tablas de 10 elementos cada una o el equivalente, 30 elementos en memoria. Si aplicamos estas cuentas a datos reales el ahorro es mucho mayor.

Si por ejemplo ahora quisiéramos añadir el elemento 117, no haría falta reservar más memoria puesto que ya tenemos todas las tablas por las que habría que pasar para encontrar su traducción. Si quisiéramos añadir traducción para la dirección 121, si que necesitaríamos dos tablas más (20 elementos). Veremos más adelante como funciona este proceso en una CPU x86-64.

Con esto tenemos una idea general del sistema: un programa cede el procesador a otros programas de forma segura, retomando el control cuando éstos terminan de usarlo (colaborativa o forzosamente), quieren acceder a algún servicio del sistema operativo, o algún dispositivo necesita atención.

Ese programa se ejecuta como si no hubiera ningún otro en el sistema pero varias veces por segundo (números comunes son 100, 250, 1000), el sistema operativo retoma el control y si hay otro proceso con mayor prioridad, guarda en memoria el estado del proceso anterior y cambia los registros de la CPU (esto incluye cambiar la tabla de páginas, posiblemente asignando la dirección de memoria donde está la nueva tabla a un registro) para que siga ejecutando el nuevo. Este proceso se conoce como cambio de contexto, del inglés: *“context switch”*.

Para acceder a los dispositivos hay diversas formas. Lo más común es tener direcciones de memoria especiales y no usables por ningún programa que mapean (del inglés "mapping"[6]) directamente el dispositivo. Un ejemplo es el dispositivo de video; escribir unos bytes en una dirección de memoria concreta, significa escribir dicho carácter en una posición determinada de la pantalla.

Otro método de acceder a dispositivos, es usando un espacio de direcciones distinto al de la memoria, esto es lo que se conoce como: "espacio IO" o más comunmente "puertos". Tal como pasa con la memoria mapeada, "escribir un dato por un puerto" equivale a enviar ese dato por la red o "leer de un puerto" puede ser leer la hora actual. En el PC se definen puertos estándar, y leer por ejemplo del puerto 0x71, equivale a leer el dato que hayamos pedido del dispositivo de reloj RTC.

1.5. Toolchain

Ahora que sabemos más o menos lo que vamos a hacer, tenemos que ver más o menos como.

Por "toolchain" se entiende el conjunto de herramientas usadas para construir un producto.

En el caso de un sistema operativo, es necesario conocer en profundidad las herramientas ya que se hace uso extensivo de ellas y como no tenemos algunas de las facilidades que se nos proporcionan cuando intentamos hacer una aplicación normal, hemos de usar otras herramientas menos sofisticadas (por ejemplo un "desensamblador" en lugar de un depurador).

Primero necesitamos un editor de texto sobre el que podemos escribir el código. Este código tiene que ser traducido a instrucciones del procesador, cargado en memoria cuando enciende el sistema, e indicar al procesador que tiene que empezar a ejecutarlo.

Vamos a explicar un poco más este proceso. Originalmente los programas se escribían con tarjetas perforadas, podemos ver una en la imagen 1.4. Un ordenador era un dispositivo que recogía estas tarjetas (tal como la impresora recoge el papel) y que mediante unos contactos podía saber que perforaciones había y por tanto que instrucciones tenía que ejecutar. Como podemos ver, todas las columnas tienen el mismo número de elementos así que podemos asociar un bit a cada uno de ellos e identificar la instrucción con un número compuesto por dichos bits. Este número se llama "código de instrucción" (del inglés: *"opcode"*).

[6] Aunque podría traducirse como proyectar, el verbo mapear existe en español y resulta más común en la jerga técncia

Figura 1.4: Imagen de tarjeta perforada

Por ejemplo, la instrucción *nop* de la arquitectura Intel, es una instrucción que no hace nada, se usa cuando tenemos que alinear datos en memoria o simplemente dejar el procesador parado por alguna razón (como esperar un acceso a dispositivo, o cuando no hay nada que ejecutar). Esta instrucción tiene el opcode 0x90. Este número en binario es: 10010000 por lo que para codificarlo en una tarjeta perforada, tendríamos que perforar los agujeros 7 y 4 (empezamos a contar en cero).

Aunque en la imagen sale nada más la zona de los números, la tarjeta podía incluir un bit más de paridad, una cabecera con alguna información, etcétera.

No todas las instrucciones son tan simples, algunas de ellas llevan parámetros. Por ejemplo, para mover una dirección de memoria a un registro, necesitamos especificar la dirección de memoria y el registro. En tal caso se coloca un byte con la instrucción y otro u otros con la dirección. Para hacer un salto (lo que equivale a mover cierta dirección al registro de instrucción) habrá que especificar la dirección a la que queremos saltar.

La forma de especificar la dirección de memoria puede variar. Podemos poner una dirección absoluta pero en tal caso necesitaríamos 4 bytes para una arquitectura de 32 bits y 8 para una de 64, esto hace que aumente la necesidad de memoria y de procesamiento. Por contra, si usamos una relativa, sólo podríamos acceder a una pequeña parte del espacio de direcciones o en el caso de un salto, sólo podría ser tan grande como permita el tamaño del operando.

Además el procesador tiene que saber en todo momento cuantos bytes componen cada instrucción porque sino perdería el sincronismo de saber donde empiezan las instrucciones y donde los parámetros. Para esto hay muchas soluciones: hacer que todas las instrucciones tengan el mismo número de bytes, usar prefijos o sufijos distintos en función del número de bytes incluidos en la instrucción, etcétera. Ver [HP06] para más información.

La forma de especificar direcciones de memoria y registros se llama: "modos de direccionamiento" y cada arquitectura tiene unos propios (cabe decir que x86 es una de las arquitecturas con mayor número de modos de direccionamiento).

Podemos especificar "100 bytes más adelante" o "50 bytes relativos al registro A", incluso tenemos modos más complejos: "registro A más el número de veces indicado el registro B". La documentación del procesador especifica que modos están disponibles (según que instrucciones) y el ensamblador que usemos (e incluso es habitual que el compilador tenga flags para ello) proporcionará métodos para especificar cada uno de los distintos tipos de salto.

En la práctica la mayoría de saltos se realizan a memorias cercanas (de nuevo el principio de localidad) ya que se trata de construcciones como condicionales, bucles, etcétera. Y es común que los movimientos de datos sean relativos a un registro que indique el inicio de algún segmento (físico o lógico) de datos.

Volviendo a las tarjetas perforadas, un problema que presentaban era añadir código entre medias de otro código. Imaginemos que usamos una dirección relativa a una instrucción, algo como: "10 bytes tras la instrucción actual". ¿qué pasaría si quisiéramos añadir instrucciones entre medias? Habría que ajustar todos los símbolos para apuntar a la nueva localización. Los programas modernos no se componen de unas pocas tarjetas sino de hasta miles de millones de instrucciones por lo que sería imposible actualizarlos todos en cada modificación. La solución es usar símbolos y resolverlos más adelante. El programa que hace este trabajo es el enlazador (del inglés *linker*).

Cuando tenemos un linker, podemos dejar las direcciones sin rellenar, y apuntar en otra parte que tenemos una "reubicación" (del inglés *"relocation"*). Después de compilar, el linker recorre la lista de reubicaciones y es capaz de rellenar las direcciones finales que han sido asignadas a cada uno de los símbolos. Para más información ver: [Lev99].

Como la correspondencia entre instrucciones y códigos de instrucción es tan simple como mirar una tabla, este proceso se automatizó y se inventaron los ensambladores (del inglés *"assembler"*). Ya no es necesario programar con los números sino con mnemónicos. Cada ensamblador tiene su sintaxis, ¡OS usa GNU Assembler con sintaxis AT&T que es la clásica de sistemas Unix.

Vamos a ver un ejemplo:

Listado 1.1: Sintaxis at&t

```
movq %rbp, %rsi
movq %rbx, %rax
```

La primera instrucción mueve el contenido del registro *rbp* al registro *rsi* y la segunda mueve el contenido del registro *rbx* al registro *rax*. Se usa *movq* en lugar de *mov* porque en esta sintaxis se incluye el tamaño del dato que movemos como parte del mnemónico y *q* significa *quad* (cuatro palabras del procesador original de 16 bits). Estas instrucciones se traducen como: "`48 89 ee`" y "`48 89 d8`" respectivamente.[7]

El ensamblador produce como salida un binario que se conoce como "código objeto". Se llama así porque a diferencia de los binarios ejecutables, no tiene ninguna información de carga, solamente se trata de una traducción en binario de las instrucciones que le hemos indicado desde el código que hemos pasado al ensamblador.

Con experiencia en este proceso nos daríamos cuenta de que las construcciones son siempre del mismo estilo: "repite este código hasta que se cumpla la condición", "imprime por pantalla un carácter", etcétera. Los programadores de la época se percataron y con los años fueron saliendo ideas con mayor grado de abstracción hasta que a mediados y finales de los 50 se inventaron FORTRAN, LISP y COBOL. Las abstracciones se asentaron: condicionales, bucles, procedimientos, programación orientada a objetos (empezó con Simula en los años 60), hasta que en 1972 apareció C.

Este es el lenguaje usado para construir ¡OS ya que a día de hoy sigue siendo el lenguaje favorito de los programadores de sistemas operativos. Esto se debe a la portabilidad del código y eficiencia, a la capacidad simple de acceso al hardware, así como al poco grado de abstracción. De hecho, este lenguaje es usado habitualmente para programar los intérpretes del resto de lenguajes de más alto nivel.

La función de un compilador de C es conceptualmente sencilla: abstraer el lenguaje ensamblador mediante construcciones más sencillas para los programadores. No es el propósito de este trabajo explicar el lenguaje C (para ello se recomienda el libro de sus autores: [KR88]), pero podemos tener algo como:

Listado 1.2: Ejemplo de código C

```
if (a < b)
        mayor = b;
else
        mayor = a;
```

Que es mucho más legible que su equivalente en ensamblador, y ya no digamos el equivalente en binario, para cuyo descifrado necesitaríamos un manual de la CPU con los mnemónicos e instrucciones.

Existe una diferencia importante entre un ensamblador y un compilador. A partir

[7] Para los curiosos, el comando "objdump" de UNIX permite ver las instrucciones para cualquier binario.

de las instrucciones de la CPU se pueden obtener las instrucciones de ensamblador que se usaron para codificarlo [8], pero no es posible obtener el código C original a partir de un binario[9].

El programa que realiza este paso se llama "desensamblador" y es útil en el desarrollo de sistemas operativos ya que a veces necesitamos instrucciones precisas, y mediante esta herramienta podemos comprobar si las instrucciones que generó el compilador fueron las que nosotros queríamos. Este problema no suele darse en programas de usuario, los cuales no necesitan código ensamblador y pueden ser programados con C en su totalidad.

El compilador usado será GCC[10], es un compilador versátil, moderno y gratuito. Este compilador no genera código binario sino código ensamblador por lo que se necesita también un ensamblador, este será: GNU Assembler[11]. Dado que un kernel es un programa especial, necesitaremos también enlazarlo en una dirección física predeterminada y para ello usaremos el linker de GNU, LD (parte de las binutils y por tanto desarrollado y distribuido junto con el ensamblador). En ningún momento podremos usar la biblioteca de C porque se apoya en las llamadas al sistema y precisamente es el sistema operativo el que las provee. Usaremos además otras herramientas:

- Make para unir las piezas.
- GDB. Aunque como hemos dicho usar un depurador no es posible, sí es posible hacer una implementación de un algoritmo que funcione como una aplicación de usuario y mover las funciones al kernel.
- Qemu y Bochs. Estos emuladores son más que útiles a la hora de hacer un sistema ya que emulan un ordenador completo. Bochs va aún más allá porque está precisamente diseñado para esta labor y aunque simple, incluye un depurador integrado, y nos proporciona un entorno sobre el que poder buscar problemas.
- Grub. Aunque aún no hemos entrado en la arquitectura Intel, Grub nos proporciona ayuda para arrancar el sistema operativo. La situación es la siguiente: el procesador sólo puede cargar programas desde memoria, pero el sistema operativo se guarda en disco, ¿cómo cargamos el sistema operativo si no tenemos sistema operativo?

[8] Los ensambladores modernos incluyen macros y otras construcciones de alto nivel que imposibilitan parcialmente esta afirmación.

[9] Hay programas que intentan hacer esta labor. Se conocen como "descompiladores" o "decompiladores". Pero aunque pueden conseguir código C a partir de código binario, es imposible asegurar que ese código se usó para generar el ensamblador.

[10] `http://gcc.gnu.org/`

[11] `http://www.gnu.org/software/binutils/`

Grub lo hace por nosotros y sin una herramienta así, habría que programar un cargador en 512 bits (tamaño del sector de arranque), el cual se tendría que apoyar en las funciones que nos proporciona una memoria ROM incluida con los PCs que se conoce como BIOS. Además este cargador necesitaría encontrar el bloque (o bloques) de disco donde se encuentra la imagen, el cual puede ser distinto en cada compilación.

Capítulo 2

x86, x86-64

2.1. Características generales

La arquitectura x86-64 es una extensión de 64 bits de la arquitectura Intel de 32 bits x86.

Esta arquitectura presenta más extensiones que ninguna otra ya que soporta compatibilidad con los procesadores antiguos, los cuales a su vez tienen modos de compatibilidad con procesadores aún más antiguos. Es capaz de funcionar en modo de 16 bits, en modo de 32 bits, en modo de 32 bits protegido, en modo de 64 bits, y también en modos de 16 y 32 bits mientras está en modo de 64 bits.

La principal razón para necesitar una arquitectura de 64 bits fue la necesidad de instalar más de 4GB de memoria en los equipos. Como vimos, la cantidad máxima de memoria que puede direccionar un equipo de 32 bits son 2^{32} bytes (4GB) y aunque Intel inventó un sistema para instalar hasta 64 GB por equipo (llamado PAE), los procesos estaban aún limitados a 4GB y el diseño e implementación de la característica por parte de los sistemas operativos era complejo.

Las extensiones principales de esta arquitectura son, además del modo de 64 bits: los registros pasan a tener un tamaño de 64 bits, añade 8 registros de propósito general, 8 registros multimedia y elimina la segmentación de direcciones.

Otra característica de las CPUs x86, es que poseen 4 niveles de privilegios. ¡OS, al igual que otros sistemas, usará sólo 2 (el primero y el último): uno para las aplicaciones y otro para el kernel. La diferencia es que el modo privilegiado tiene acceso a las instrucciones reservadas para el sistema operativo y el otro está limitado a instrucciones

más inofensivas.

2.2. Registros

Como curiosidad podemos ver los registros de la CPU x86-64 disponibles para los programadores de aplicaciones en la figura 2.1.

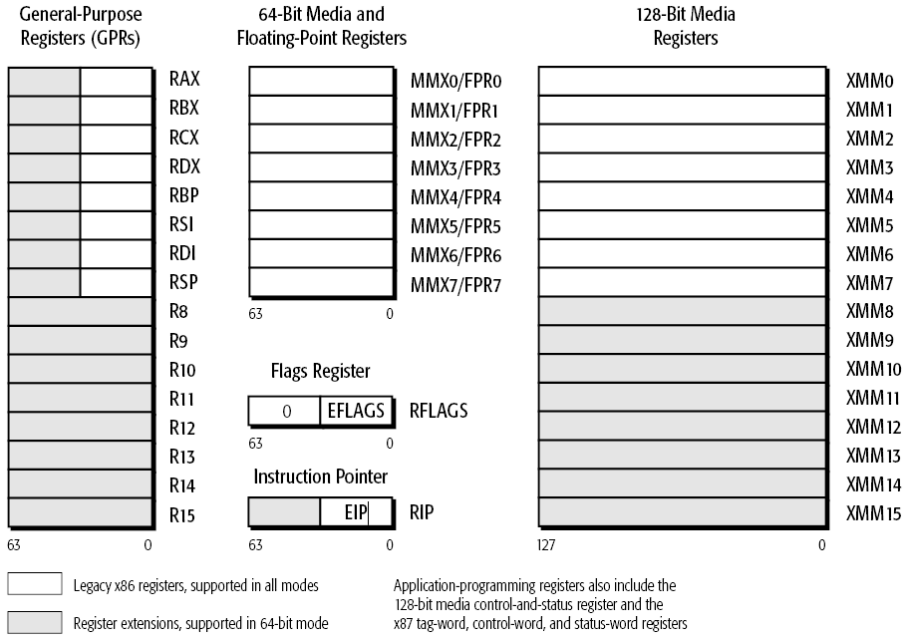

Figura 2.1: Registros de la CPU x86-64

El registro *rip*, se usa para apuntar a la siguiente instrucción que se ejecutará. No se puede manipular directamente sino que existen instrucciones para introducir saltos en el código. Como se mencionó, estos saltos se dividen en distintos tipos según la distancia que queramos saltar. De esta forma se ahorra memoria y ciclos de procesador al no ser necesarios los 8 bytes que especifican una dirección absoluta.

Los registros de propósito general que ya estaban en modo de 32 bits, pueden accederse en modo de 16, 32 ó 64 bits. Por ej. el registro *rax* tiene un tamaño de 64 bits, si accedemos al registro *eax* estaremos accediendo a sus 32 bits inferiores, y si accedemos a *ax*, accederemos a los 16 bits más bajos de *rax* y de *eax*. Además los registros de 16 bits se dividen en dos partes, el byte alto y el bajo o: *ah* y *al*. Esta es parte de la complejidad que mencionamos que el procesador arrastra por mantener la compatibilidad.

El registro *rsp* se usa para gestión de la pila y para los *"stack frames"*. Estos elementos se usan para las llamadas a funciones y se verán más adelante.

El registro *rflags* (*eflags* en modo de 32 bits y *flags* en modo de 16) se accede o modifica bit a bit y no se suele hacerse directamente sino que las instrucciones lo usan colateralmente. Contiene bits (o flags) con resultados o configuraciones tales como si la última operación tuvo acarreo, desbordó, o fue cero. De esta forma podemos tener este pseudocódigo ensamblador:

Listado 2.1: Pseudocódigo comparador

```
compara a y b
if el flag zero es 1
        son iguales
else
        son distintos
```

Que se traduce al siguiente código ensamblador (sintaxis AT&T):

Listado 2.2: Ensamblador comparador

```
; mueve el valor a los registros indicados:
mov $5, %ecx
mov $5, %edx
; resta uno de otro (sin modificarlos)
; y activa eflags.ZF (Zero Flag) si son iguales.
cmp %edx, %ecx
; salta a la etiqueta "distintos" si no son iguales (eflags.ZF == 0)
jne $distintos
; esta parte se ejecuta si fueron iguales.
;
; jmp salta incondicionalmente a la etiqueta,
; necesario para no ejecutar la parte en que fueron distintos.
jmp $salida
distintos:
; esta parte se ejecuta si fueron distintos.
salida:
```

Para completar, en un lenguaje de alto nivel como C, el código quedaría mucho más sencillo:

Listado 2.3: Comparador en C

```
if (a == b)
        // ... iguales
else
        // ... distintos
```

Como hemos dicho, un compilador de C traduciría este código en algo parecido a lo que podemos ver en el listado 2.2.

2.3. Instruction Set

Las instrucciones se dividen en varios grupos. Será necesario conocer las instrucciones básicas ya que en este grupo se incluyen las que manejan el flujo del programa, movimientos de datos, etc. También hemos de conocer las de sistema ya que permiten cambiar los modos del procesador, activar la protección de memoria o gestionar la memoria virtual.

Las instrucciones multimedia son un poco especiales ya que para usarlas hay que inicializar la FPU ("unidad de coma flotante" o *"floating point unit"*) y en el momento que usamos una instrucción multimedia el estado de los registros cambia (otra consecuencia que deriva del hecho de que la FPU era un añadido en procesadores antiguos).

Desde el punto de vista del kernel, es computacionalmente caro usar números reales, por tanto el uso de la FPU se reserva para tareas como criptografía, CRC, etcétera. No obstante, tendrá que salvar los registros de la FPU al cambiar de aplicación, y como son muchos suele usar una técnica conocida como *"lazy FPU context switch"* (que podemos traducir como "cambio de contexto con FPU perezosa"). Lo que hace es no salvar los registros de la FPU cuando cambia de aplicación sino que desactiva la FPU y si la nueva tarea intenta usar números racionales se lanza una excepción; el kernel entonces detecta el caso y salva los registros permitiendo a la nueva tarea usarlos. Este mecanismo es más rápido que guardarlos siempre ya que no todos los procesos usan números racionales y es posible que la tarea que corrió por última vez vuelva a correr en un futuro cercano y no sea necesario guardar y restaurar los registros sino que los tenga ahí de la última vez.

Este mecanismo se menciona porque es similar al que permite tener memoria virtual mediante la paginación a disco.

Las referencias oficiales para las instrucciones son: [AMDc], [cori] y [corj] y conviene tenerlas a mano a la hora de usar ensamblador..

2.4. Modos de operación

Cuando un procesador x86-64 arranca se encuentra en modo real de 16 bits. La MMU no está activa y por tanto todas las direcciones que se envían por el bus serán

físicas. Tanto el procesador como el sistema operativo ven la memoria como un array de 1MB.

Dado que en ¡OS se optó por la solución más complicada de usar protección de memoria, tenemos que pasar el procesador ese modo. Para ello será necesario instalar tablas de páginas (recordemos el capítulo 1) y modificar unos flags de un registro, además de crear y saltar a un segmento de 32 bits (en seguida veremos la segmentación).

Como ¡OS necesita el modo de 64 bits para funcionar, aún será necesario un cambio más. Dado que el cargador que usaremos nos deja el procesador en modo de 32 bits, vamos a poner aquí parte del código necesario para hacer el cambio ya que a pesar de que aún no se han discutido algunos conceptos, será útil hacernos una idea rápida de los comienzos:

Listado 2.4: Como habilitar el modo de 64 bits

```
; Las directivas que empiezan por punto no es código
; sino sentencias para el ensamblador.
; .section indica en que sección del binario colocarlo.
; En este caso se trata de una sección propia creada para colocar
; el código de arranque.
.section .mboot

; El código que sigue es de 32 bits.
.code32

; Leemos cr4, vamos a activar los flags que se indican.
movl %cr4, %eax
; bts activa bits, de 0 a 31.
bts $4, %eax    ; PSE, Page size extensions.
bts $5, %eax    ; PAE, Tablas de páginas de 64 bits.
bts $7, %eax    ; PGE, Page Global enable.
; Activamos los cambios.
; Nótese que el registro cr4 no se puede manipular directamente.
; Hay que tener cuidado con no escribir cualquier dato
; en el registro ya que podríamos modificar otros flags.
movl %eax, %cr4

; Usa registro EFER para habilitar Long Mode.
mov $0x0c0000080, %ecx
; Esta instrucción que sigue lee del registro MSR (ver manuales
; de la CPU, son unos registros especiales) indicado en ecx
; y coloca el dato en eax (el cual se indica implícitamente):
rdmsr
; Activa el bit 8 (noveno bit) que es Long Mode Enable
bts $8, %eax
```

```
; Escribe eax en el registro MSR indicado por ecx (ecx sigue
; teniendo el mismo valor ya que no lo hemos modificado).
wrmsr

; Aquí mapearíamos las tablas de páginas necesarias,
; algo que veremos después como se hace.

; Pega un salto largo al segmento 1 (cada uno ocupa 8 bytes,
; por eso se pasa 0x8).
; Esta instrucción mira la tabla global de segmentos, le suma
; los bytes indicados y lee el descriptor que esté ahí colocado.
; Copia el descriptor al registro CS (code segment register)
; y salta a la dirección de memoria indicada por el símbolo
; (la cual se descubrirá al enlazar).
ljmp $0x8, $start64

; Indica que el código que sigue es de 64 bits.
.code64
; En linker asignará a este símbolo la dirección que le
; corresponda a este trozo de código.
start64:
; Cuando salte aquí, ya estamos en modo de 64 bits.
```

A pesar de que no sería necesario para el ejemplo, he incluido la parte del salto ya que así se van introduciendo algunos conceptos como segmentación. La idea es que definimos en memoria una tabla (GDT o *"global descriptor table"*) con descriptores de segmentos, y toda instrucción que se ejecute en el procesador lo hace con un descriptor cargado en un registro, el cual incluye permisos y privilegios. El salto se hace para cambiar a un segmento de 64 bits.

2.5. Modelo de Memoria

En este punto ya deberíamos tener una idea básica de lo que es la memoria virtual, pero vamos a aplicar el concepto a un procesador x86-64 funcionando en modo de 64 bits, y a nuestro kernel.

Justo antes del salto que indicamos en la sección 2.4, se mapeó la memoria aunque sin indicar los mapeos concretos. La duda que surge es ¿qué mapeamos exactamente y como?

Pensemos que elementos tenemos y que posibilidades nos ofrecen. Tenemos un espacio virtual enorme y una cantidad de memoria RAM que comparativamente será pequeña pero de tamaño desconocido a la hora de programar el kernel (cada equipo puede

tener una cantidad de memoria instalada diferente). Tenemos también un código que ocupará unos pocos MB y que ha sido cargado en el primer MB de memoria física (las razones para esta decisión se expondrán más adelante ya que no son relevantes para esta discusión).

Podemos usar por tanto un mapeo lineal, es decir, el bit 1MB lógico equivaldría a la dirección física 2^{20}. Pensemos más en futuro, ¿en qué dirección virtual cargaremos después las aplicaciones? Dado que sabemos cuanto ocupará el kernel una vez compilado (el linker nos podría rellenar una variable con ese dato), podremos colocar las aplicaciones justo después de la imagen del kernel.

Este enfoque podría funcionar en un kernel monotarea (obviando que el kernel también necesita poder reservar memoria), pero si hubiera varias tareas tendríamos varios impedimentos técnicos. Por ejemplo los binarios tienen una dirección de carga, ¿que ocurriría si el mismo binario se ejecutase dos veces? Además las tareas cambian de tamaño durante su ejecución (el uso de memoria no es constante), y si todas usaran la misma tabla de páginas unas tareas podrían acceder a la memoria de las otras sin ningún control.

La primera idea sería usar una tabla de páginas para el kernel y otra para los procesos. Vamos a suponer que escogemos esta opción, que tenemos el kernel inicializado y que vamos a ejecutar la primera tarea. El kernel lee de disco el binario y lo mapea en memoria en una nueva tabla. Una vez mapeado el código surge otro problema, ¿cómo hacemos el cambio de tablas? En x86-64 no es posible hacer en una sola operación el cambio de tabla y el salto a otra dirección de memoria, es decir, si cambiamos la tabla, el registro *rip* (el cual usa direcciones virtuales), apuntará a las instrucciones del kernel que hacen el cambio, pero una vez hecho, la dirección no estaría mapeada y no se podría ejecutar la siguiente instrucción que salta al código de la tarea.

Se podría mapear una página (o varias) del kernel en cada proceso para realizar el salto, algo como un trampolín de código para entradas y salidas entre modo kernel y modo usuario. Pero con esta solución la tabla de páginas habría de cambiar y por tanto las cachés del procesador se invalidarían por cada cambio (recordamos que hay un cambio en cada llamada al sistema y en cada cambio de contexto), e invalidar las cachés de forma tan continua supone una pérdida de rendimiento bastante importante. Hay que destacar que un kernel es un software que necesariamente ha de estar optimizado ya que sino la lentitud en el sistema se percibe rápidamente.

La solución tomada por ¡OS y por la mayoría de sistemas operativos, es mapear el kernel completo y tanta memoria como se pueda en la memoria virtual de todos los procesos. La cantidad concreta variará en función de la arquitectura y del siste-

ma operativo; en arquitecturas con una cantidad de memoria virtual pequeña, como por ejemplo x86, es común dejar 1GB para mapeos del kernel y 3GB para uso de las aplicaciones. Es necesario tener esto en cuenta para entender por que una aplicación compilada para 32 bits solamente puede hacer uso de 3GB de memoria en un sistema operativo que use memoria virtual. En un sistema operativo de 64 bits suele reservarse la mitad para el kernel (puede considerarse como el espacio virtual negativo si tomamos el primer bit como el signo) ya que 2^{63} sigue siendo una cantidad enorme comparado con la memoria física que podemos encontrar en ningún equipo.

Volviendo a nuestra discusión, como sería un poco extraño mapear distintas porciones repartidas por el espacio virtual, mapeamos en cada proceso al principio (ya que el kernel está precisamente al principio y estamos mapeando linealmente), una cantidad de memoria, y para el resto de nuevo tenemos varias opciones.

Podríamos usar la misma tabla para todos los procesos pero habría que marcar las páginas de los procesos que no se están ejecutando para que no fueran accesibles desde el que se ejecuta, esto sería costoso y en un entorno de varios procesadores no funcionaría ya que podría haber dos procesos en ejecución simultáneamente (aunque se podrían marcar las páginas con el número de procesador en el que serían válidas). Por otro lado, una página entregada a un proceso haría que otro proceso no pudiera hacer uso de esa dirección virtual (algo de lo que se podría abusar fácilmente), por lo que la memoria total usable por cada proceso sería muy inferior a la total instalada en el sistema (y recordemos que una de las características de la memoria virtual es que permite precisamente usar más memoria de la que haya instalada).

Si usamos una tabla para cada proceso ya no tendríamos que marcar ninguna página por estar usada por otro proceso y cada proceso podría hacer uso de todo el espacio de direcciones (salvo el mapeado para el kernel). Al tener tablas de páginas por cada proceso tampoco es necesario que las direcciones se traduzcan linealmente y podemos por tanto buscar una página libre cualquiera y mapearla donde se necesite (como este proceso se hace por hardware no penalizará el rendimiento). Además, cuando no haya hueco en memoria física para colocar una página, podemos mover cualquier otra que no se use a disco (o descartarla si ya estuviera en disco como por ejemplo cualquiera que provenga de un archivo y no haya sido modificada) y hacer que el proceso use como decimos más memoria que la instalada en el equipo ya que un disco es mucho más grande.

Básicamente este es el funcionamiento resumido de la memoria virtual. Pero ¿qué ocurriría si un proceso está ejecutando una tarea y llega un paquete de red o una pulsación de teclado para otra tarea?

El kernel podría guardar el dato y entregarlo a la tarea correspondiente cuando se ejecutara pero supondría hacer 2 copias (una a la memoria del kernel y otra a la del proceso). También podríamos esperar a que se ejecutara esa aplicación pero si el flujo de paquetes fuera alto, el buffer del hardware correspondiente podría desbordarse. No es posible guardarlo en la memoria de ese proceso para el que se recibe porque se está ejecutando otro proceso diferente con otra tabla de páginas y esa dirección no es accesible.

Esta memoria que no es directamente accesible se llama "memoria alta" (del inglés "*high memory*") y el kernel reserva una cantidad de su espacio virtual para poder mapearla de forma temporal.

Para una explicación más completa sobre la memoria virtual podemos usar un recurso como [BP05] o casi cualquier otro libro del kernel de Linux indicado en la bibliografía. Todo este sistema es conceptualmente simple pero tiene muchas sutilezas que complican la programación y el desarrollo.

Ahora pasamos a algo más práctico. En Linux podemos ver cuanta memoria alta y baja tenemos, por ejemplo:

Listado 2.5: Estado de la memoria sin usar memoria alta

```
~$ free -lm
             total       used       free     shared    buffers     cached
Mem:           496         52        443          0          4         16
Low:           496         52        443
```

Este equipo no tiene memoria alta porque el kernel puede mapear los 496MB de RAM presentes en el espacio virtual que ha reservado para su uso propio (direcciones que estarán mapeados en espacio kernel para todos los procesos).

Listado 2.6: Estado de la memoria con memoria alta.

```
~$ free -lm
             total       used       free     shared    buffers     cached
Mem:          2013         40       1972          0          4         16
Low:           857         18        839
High:         1155         22       1133
```

En este caso 1155MB no son direccionables por el kernel y si quisiera acceder a la memoria de un proceso cuando tiene mapeado otro, habría de mapear alguna cantidad de memoria aunque fuera de forma temporal.

Las estructuras en espacio de kernel no pueden hacer referencia a una dirección en espacio de usuario ya que la tabla de páginas cambia y con ello la dirección física

(que es la que al final se accede ya que la virtual sólo es como decimos, un traductor intermedio).

En un equipo de 64 bits este problema no existe ya que el espacio de direcciones es enorme (miles de millones de veces la memoria física presente en los PCs de hoy en día), por tanto no hace falta tanta complicación: la memoria RAM se mapea como dijimos dos veces, una linealmente en modo kernel y otra en modo de usuario de forma distinta para cada aplicación. Por tanto la memoria máxima que podría usar un proceso sería la mitad, es decir, 2^{63} bytes (lo cual ha de ser más que suficiente para muchísimos años).

La ventaja de tener toda la memoria mapeada en modo kernel es que los mapeos entre la memoria del kernel y la memoria física serán lineales (podrían ser incluso inmediatos, todo dependerá de la dirección virtual en la que se coloque el cero físico). Y por tanto ¡OS, que se mapea en la segunda mitad del espacio virtual de direcciones, para convertir unas direcciones en otras usa estas macros:

Listado 2.7: Macros de conversión entre memoria virtual y física

```
#define __pa(_x) ((void *)(((u64)_x) - K_PAGE_OFFSET))
#define __va(_x) ((void *)(((u64)_x) + K_PAGE_OFFSET))
```

Cuando queramos saber la dirección física de la variable "var", haremos:

Listado 2.8: Dirección física de la variable

```
__pa(&var);
```

Y cuando queramos acceder a una dirección física concreta haremos:

Listado 2.9: Dirección lógica de la variable

```
__va(variable_que_contiene_dirección_física);
```

Estas transformaciones quedarán aisladas en la capa que maneja la memoria física y para servicios de bajo nivel que traten con el hardware, así que a la hora de programar el código no tendremos que preocuparnos la mayoría del tiempo.

Como hemos dicho, ¡OS, al igual que otros kernels, mapea el kernel en la segunda mitad del espacio (conocida como *"higher half"*) y deja libre la mitad baja (*"lower half"*). No hay ninguna razón por la que una opción sea mejor que la otra, salvo porque quizás es más fácil desmapear la mitad inferior para buscar errores y porque los punteros de las aplicaciones son más "naturales" ya que pueden contener cualquier valor entre 0 y 2^{63} (esto no es del todo cierto porque ninguna CPU x86-64 actual usa el

espacio de direcciones entero sino que están limitadas a 48 bits, por lo que será menos).

Como punto extra vamos a explicar como hace un kernel para pegar el salto a su *higher half*. Inicialmente (en modo real), no hay mapeos, luego la dirección usada para acceder a las variables es la física. Cuando activamos protección de memoria, estamos en un caso parecido al discutido anteriormente: no podemos saltar a la dirección lógica *higher half* donde colocaremos el kernel hasta que activemos protección de memoria, pero no podemos saltar después de activarla porque la instrucción que hace el salto no podría ser leída por el procesador al no estar mapeada.

Vamos a verlo con un ejemplo: la instrucción que activa protección está en la dirección de memoria 5 y la que salta está detrás (podría no ser la dirección 6 si la instrucción que activa protección ocupara digamos 2 bytes y podría no ser la 7 si el procesador obligara a alinear las instrucciones).

Lo que hacemos es preparar unas tablas para que la dirección 5 y siguientes sean accesibles también desde la dirección 5 virtual y siguientes, e igualmente ponemos otra traducción para que desde la 105 también se llegue a la dirección 5. Una vez preparadas las tablas podemos activar la MMU y funcionará puesto que tenemos una traducción lineal.

En este momento es posible acceder a esas instrucciones tanto con la dirección 5 como con la 105, por lo que saltamos desde la instrucción de salto (6, 7, etcétera) a la dirección de la instrucción que la sigue + 100 y el programa sigue ejecutándose desde ahí y será posible desmapear el mapeo lineal original.

A partir de este momento todas las direcciones referenciarán a las direcciones altas y ya podemos desmapear las bajas. Por supesto, ese código ya no será accesible con las direcciones bajas.

La única peculiaridad que hay que comentar, es que cuando compilamos el código, normalmente no hay que indicarle la dirección de carga pero como en este caso será distinta de la de ejecución, tendremos que pasar esos parámetros en el script del linker que construyamos.

El lector extremadamente agudo puede preguntarse que si hay direcciones de carga y de ejecución, ¿sería necesario que la de antes del salto se coloque en una dirección de carga distinta de la de después? La respuesta es afirmativa. ¡OS coloca como última instrucción ejecutable (detrás podría haber datos) de la sección `.mboot` la instrucción de salto y la dirección a la que se salta ha de estar en cualquier otra sección de código (concretamente en la sección `.code`).

2.6. Interrupciones y Excepciones

En todo sistema hay eventos asíncronos, es decir, pueden llegar en cualquier momento. Por ejemplo, un bloque se ha leído de disco, un usuario pulsa una tecla, etcétera. Si el sistema está ocupado ejecutando una aplicación, ¿cómo sabe que un evento de este tipo ha ocurrido?

Algunos sistemas sondean (del inglés: *polling*) periódicamente cada dispositivo por si necesitara atención. Esta opción es buena cuando el número de interrupciones es alto y de hecho hay sistemas operativos que cambian de modo de funcionamiento cuando un dispositivo tiene una tasa alta de interrupciones, pero lo normal es usar una interrupción del procesador.

El concepto es simple, se trata de un mecanismo para interrumpir a la CPU. El procesador se carga con una tabla con direcciones de procedimientos y cuando hay una interrupción, dado que ésta está numerada, se salta a la entrada cuyo número ha ocurrido y por tanto a un procedimiento de código que lo maneja.

Una CPU x86-64 permite hasta 256 interrupciones (ya que el número que las identifica ocupa un byte). Las interrupciones además pueden ser de tres tipos: excepciones (errores software), interrupciones software (generadas por la instrucción de interrupción) e interrupciones externas (respuesta del procesador a un evento externo, es decir, un dispositivo).

Tenemos interrupciones para cuando hay división por cero, para poder depurar programas (así es como funcionan los depuradores, sin un apoyo hardware sería difícil hacerlos funcionar), para cuando ejecutamos instrucciones o accesos inválidos, etcétera.

La interrupción 14 se lanza cuando algún proceso intenta acceder a una página que no está presente en memoria (bit "*present*" con valor cero en tablas de páginas de procesadores x86). Cuando esto ocurre, el sistema operativo instala un manejador que mueve la página necesaria a memoria ram (tal vez mueva alguna página de más para no tener que manejar una interrupción poco después) y marca la página en la tabla de páginas como presente. Es posible que la excepción no se haya lanzado porque el proceso necesita leer una parte del binario que aún no ha sido cargada en memoria (o de un archivo de texto) sino porque el proceso intentó acceder a una dirección de memoria para la que no tenía permisos (por ejemplo escribir en una página de código o reservada para el kernel), en tal caso el kernel matará el proceso.

El concepto de las excepciones no es muy complejo. La implementación tiene más detalles que se verán en su momento.

2.7. Registros del sistema

Estos registros del procesador sólo están accesibles en modo privilegiado.

Se incluyen: registros de control, flags de sistema, registros que trabajan con descriptores de tablas (describen las estructuras en memoria que usa el procesador), registros de estado (para multitarea por hardware, que puede ser usada opcionalmente) y registros de depuración. Además hay otros registros "específicos del modelo", como el registro EFER usado cuando activamos el modo de 64 bits.

La documentación de estos registros está en los manuales de programación de sistema de Intel ([cork]) y AMD ([AMDb]).

2.8. Segmentación

La segmentación trata de dividir la memoria en segmentos o secciones. Es un concepto parecido a la paginación en cuanto a que ambos sistemas se pueden usar para conseguir protección de memoria (alternativa o simultáneamente).

Mediante una dirección base y un límite, las direcciones se traducen. Si accedemos al byte 10 de un segmento que empieza en el byte 20, accederemos a la dirección 30 de memoria.

En modo de 64 bit la segmentación prácticamente desaparece. Los registros (salvo *fs* y *gs*) tienen como base 0 y como límite 2^{64} y sólo se usan para control de permisos, bien en cuanto a usuario y sistema o en cuanto a código y datos (los segmentos de código se marcan de sólo lectura ya que no es común y mucho menos necesario escribir en el código de un programa).

Además, la segmentación tiene otra función de uso obligado para el sistema operativo; las estructuras que usa el procesador en memoria se definen como pequeños segmentos.

Por ejemplo, la tabla principal que gestiona los segmentos disponibles en el sistema se llama GDT (de "global descriptor table"). ¡OS define algo así:

Listado 2.10: Ejemplo de GDT

```
u64 gdt[GDT_NENTRIES] __attribute__ ((aligned (16))) = {
        0x0000000000000000,
        0x00af98000000ffff, /* k_cs */
        0x008f92000000ffff, /* k_ds */
        0x008f92000000ffff, /* k_ss */
        ... etc.
```

```
};
```

Un segmento vacío, después los segmentos para el kernel en este orden: código, datos y pila (stack) y el resto que no se muestran aquí. Para decirle al sistema que use estos segmentos, se construye un descriptor de segmento y lo que se carga para activar la tabla de segmentos es el descriptor:

Listado 2.11: Carga de GDT

```
struct gdt64_ptr {
        u16 len;
        void *base;
} __attribute__ ((__packed__, aligned (8)));

...

struct gdt64_ptr gdt64_ptr;

gdt64_ptr.base = gdt; /* Dirección del gdt */
gdt64_ptr.len = len; /* Bytes que ocupa la tabla - 1 */

asm volatile ("lgdtq %0\n"
                : : "m" (gdt64_ptr));
```

Es decir, construye un descriptor formado por la longitud y la dirección base y se lo pasa a la instrucción "*lgdtq*" para cargarlo (de nuevo la instrucción termina en "q" porque la dirección que se le pasa como argumento es de 64 bits).

Los atributos que se ponen al final de la estructura son requisitos impuestos por la instrucción sobre el descriptor que se le pasa como parámetro. El primer argumento ("__packed__") se usará habitualmente ya que hace que los campos de la estructura se coloquen seguidos y sin alinear. Si lo omitiéramos podría ser que el compilador, para optimizar no colocara "base" justo después de "len" sino alineada a la siguiente dirección múltiplo del tamaño de palabra. El otro argumento ("aligned (8)") hace que la propia estructura se coloque en una dirección de memoria múltiplo de 8 bytes. Es probable que el compilador lo hiciera de todas formas pero es necesario para asegurarnos ya que si no se cumplen estas condiciones el procesador no aceptaría la estructura.

La CPU carga en un registro interno (lo que quiere decir que es invisible para el programador) el descriptor de la tabla de segmentos, y cada vez que haya un acceso a un segmento nuevo, busca en la tabla el descriptor, lo lee y lo precarga en el registro apropiado. Por ejemplo, imaginemos el salto a 64 bits del código 2.4; como se indica, queremos saltar a un segmento de código de 64 bits en modo kernel (esos datos están codificados en el descriptor). El procesador accede a la dirección de memoria indicada

respecto del registro GDT (un ejemplo de acceso con segmentación), lee 64 bits y los valida como hemos dicho (si es código, permisos, etc), en caso de ser válido carga el descriptor en el registro *CS*, y a partir de esta carga todos los accesos a memoria se realizan respecto de dicho segmento (como es un segmento de código, se refiere a accesos de código).

Dado que este proceso lo presentamos de forma simplificada, podemos ver el funcionamiento completo de la instrucción en los manuales de instrucciones del procesador.

Se definen segmentos también en otra tabla llamada LDT que es similar a la GDT pero por proceso (algo que imponen los sistemas operativos y no la CPU), en la tabla de interrupciones (IDT o *interrupt descriptor table*) y para describir tareas si se usa TSS o *"task state segment"* (opcional ya que podemos hacer cambio de tareas o *task switching* por software).

2.9. Paginación

Como vimos, la parte superior se mapea linealmente sobre la memoria ram, es decir, un acceso de memoria a la dirección `0xffff800000000000` accede a la dirección física `0`. También vimos que esta traducción no se hace individualmente para cada dirección sino en bloques llamados páginas.

No es necesario que esta traducción se vuelva a hacer cada vez que cambiemos de página. Vamos a suponer que tenemos un bucle en un programa y que el código que forma el cuerpo del bucle está dividido en dos páginas. Sería mucho trabajo tener que traducir cada vez que se alcance el código en la segunda página y cada vez que se regrese al principio del bucle. Lo que se hace es dividir las direcciones en dirección base de página más desplazamiento u *offset* (proceso que hace el hardware), y almacenar las direcciones base de página en una caché conocida como TLB (del inglés *"translation lookaside buffer"*). De esta forma las últimas páginas referenciadas en un proceso se traducen inmediatamente ya que estas cachés son casi tan rápidas como el procesador (mucho más que la memoria RAM en todo caso).

Una CPU x86 en modo de 64 bits permite tres tamaños de páginas: 4KB, 2MB y 1GB. Todos ellos pueden usarse simultáneamente.

El tamaño más comúnmente usado como tamaño de página es 4KB. Este tamaño tiene ventajas e inconvenientes. 4KB no es mucha memoria y para cubrir la memoria usada por los procesos se necesitan muchas páginas, esto hace que sea computacionalmente pesado manejarlas, además después de que el sistema haya estado funcionando

un tiempo puede ser difícil encontrar dos páginas físicas contiguas. Otra desventaja es que la presión sobre el TLB crece.

Entre las ventajas tenemos que los sistemas ya escritos usan páginas de 4KB y reduce la fragmentación interna; un término que se refiere al espacio malgastado de las páginas. Si por ejemplo quisiéramos ejecutar un binario de 4097 bytes, necesitaríamos dos páginas de 4096 a pesar de que en la segunda sólo se usara un byte, 4095 bytes se desperdician por la fragmentación interna ya que los datos han de ir en otra página. Cuanto más grande es el tamaño de página, más aumenta el espacio malgastado por la fragmentación interna.

Las páginas de 2MB parece excesivamente grandes. La fragmentación interna sería importante. Imaginemos un proceso de unos pocos KB de tamaño y que no use muchos datos, al menos necesitaría 4MB de memoria para funcionar (una página para código y otra para datos). ¡OS usa este tamaño de página de forma experimental. Mediante este enfoque las estructuras del kernel se simplifican y hoy en día es normal tener varios GB de memoria en cualquier equipo (además de que esta fragmentación sólo se produce en los subsistemas que usen páginas enteras, el kernel podrá reservar sus estructuras en porciones de páginas).

1GB dedicado a cada página es excesivo salvo para aplicaciones que lo requieran (como bases de datos grandes por ejemplo). El sistema operativo puede proporcionar un interfaz para proveer páginas de 1GB pero hay que tener en cuenta que será casi imposible encontrar 1GB libre y contiguo en memoria física.

Vamos a ver por tanto como funciona la paginación con un tamaño de página de 2MB.

En la figura 2.2 vemos un diagrama de como serían las estructuras de tablas de páginas. Los bits 63 al 48 (ambos inclusive) se reservan como extensión de signo (lo cual quiere decir que no se usan por procesadores actuales), los bits del 47 al 39 (es decir, 9 bits, o 512 posibilidades), dividen la memoria (virtual) en 512 partes, por tanto, cada una mapeará un tamaño de $2^{47}/512$ ó $2^{47} >> 9$ bytes (es decir 256 GB).

La primera tabla: PML4E, va a contener en cada entrada, punteros a tablas PDPE o cero si ese rango de direcciones no está mapeado en el espacio virtual. Cada entrada de la tabla PDPE contendrá un puntero a una tabla de elementos PDE, y cada entrada de la tabla PDE contendrá una dirección de memoria física o cero si la página no está mapeada.

Con operaciones de bits es rápido seguir las tablas. Si buscamos por ejemplo la dirección física para la virtual `0xffff800000000000`, sólo tenemos que contar respecto de los bits que "cubre" la entrada de la tabla que manejamos y seguir los punteros.

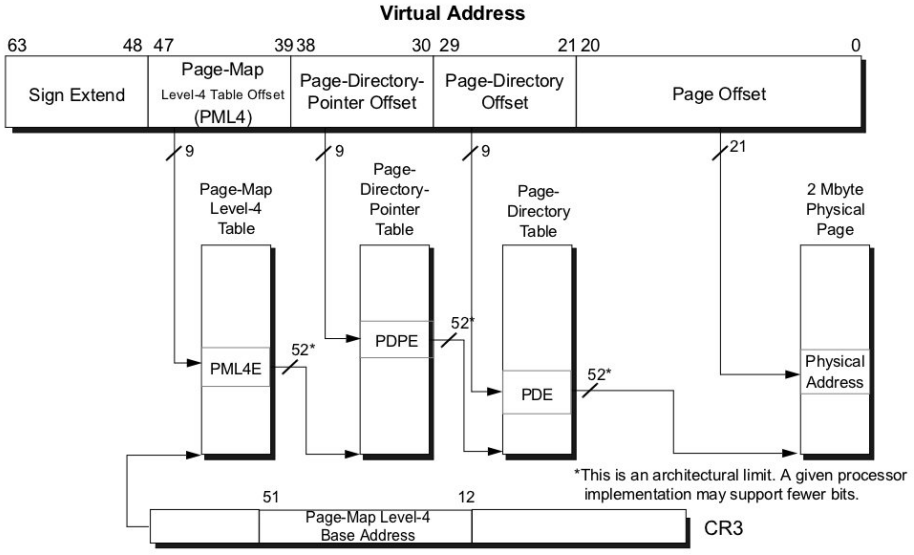

Figura 2.2: Paginación con páginas de 2MB

Es decir, `(0xffff800000000000 >> 39) & 0x1ff`, lo cual vale 256, y a 8 bytes por entrada, resulta en un offset de tabla de 2048 bytes. El resultado es de esperar porque `0xffff800000000000` es justo la página que mapea el kernel y la mitad del espacio virtual disponible.

Listado 2.12: Recorriendo las tablas de páginas.

```
/* Esto devuelve el número de entrada de la tabla PML4
 * con la dirección de la tabla PDPE que tenemos que mirar.
 * En este caso será la número 256 o byte 2048. */
numero_entrada = (direccion_virtual >> 39) & 0x1ff;
/* Cada entrada ocupa 8 bits */
pdpeptr = *(pml4ptr + ( 8 * numero_entrada ));

/* Hacemos lo mismo para calcular el offset sobre el PDPE:
 * En este caso será 0, luego saltamos a otra tabla (PDE)
 * cuya dirección está en el primer elemento de pdpeptr. */
pdeptr = *(pdpeptr + 8 * ((direccion_virtual >> 30) & 0x1ff));

physical_address = *(pdeptr + (direccion_virtual >> 21) & ~0x1fffff);
```

Como hemos dicho, este proceso se repite cada vez que se accede a memoria y la traducción no se encuentra en la tabla TLB.

El primer detalle que tiene que llamarnos la atención es que los 21 bits más bajos de la dirección virtual no se usan. Si nos damos cuenta, todas las direcciones de las páginas son múltiplos del tamaño de página, luego para una página de 2MB, los 21 bits menos

significativos son cero. El PDE en lugar de dejar esos bits a cero o ahorrar dejando que trabajáramos con menos bits, los usa para flags de página. Los más importantes son:

- u/s: para indicar el privilegio de CPU necesario para acceder a las direcciones, *user* o *supervisor*.
- r/w: los permisos de acceso a las direcciones. Los mapeos que se hagan para código se marcarán como "r" (lectura) y los que se hagan para datos como "r/w" (lectura/escritura).
- p: marca si la página (o tabla si se refiere a un nivel anterior) está presente o no en memoria. Así podemos llevar los datos a disco y cuando el proceso acceda a dicho segmento, dará el fallo de protección que comentábamos y se lanzará una interrupción.
- Bits que indican el tipo de tabla. Si es PML4, PDPE, PDE, o en el caso de páginas de 4KB, PTE.

La memoria virtual es algo a veces confuso, se recomienda mirar los manuales del sistema de AMD ([AMDb]) y de Intel ([cork]).

2.10. Llamadas al sistema

Las llamadas al sistema (o *"syscalls"*), como vimos, son los servicios que ofrece el sistema operativo para que las aplicaciones no tengan que acceder directamente al hardware.

Tendremos llamadas al sistema para controlar procesos (cargarlos, ejecutarlos, crearlos y terminarlos), para manejo de archivos (abrirlos, leerlos, escribirlos y cerrarlos), para manejo de dispositivos (usarlos, leer de ellos o configurarlos), para información (fechas, datos de sistema, atributos) y para comunicación (abrir conexión de red, enviar y recibir, conectar dispositivos).

El funcionamiento de las llamadas al sistema tampoco es complejo (al igual que otros mecanismos de la CPU, la implementación si que es mucho más tediosa que el concepto), el software lanza una interrupción, la cual provoca una llamada a un procedimiento manejador del kernel, ese procedimiento tiene una tabla y en función del número colocado en un registro despacha la petición a la función que se ha solicitado.

Por tanto las llamadas al sistema tienen que ser estables, no podemos cambiar los números ni los parámetros ya que el software que haga uso de ellas dejaría de funcionar.

Las CPUs x86-64 traen un mecanismo nuevo para las interrupciones que se conoce como: SYSCALL/SYSRET o SYSENTER/SYSEXIT (hay dos nombres porque se crearon dos mecanismos distintos, el primero por AMD y el segundo por Intel). Dado que en las llamadas al sistema el procesador ha de cambiar de nivel de privilegio, mediante este mecanismo el cambio es más rápido. La idea es que se preconfiguran unos datos en unos registros de la CPU y para generar una llamada al sistema se usan las nuevas instrucciones en lugar de usar una interrupción.

2.11. Stack y call stack

La pila es una estructura que se llama así porque nos recuerda a una pila de platos: no es posible quitar elementos inferiores sin quitar los superiores. Por tanto tiene básicamente dos operaciones (si obviamos, crear y destruir): *push* y *pop*. La primera añade un elemento y la otra lo quita. Además en este caso se añade, como optimización, otra operación que podemos llamar "mover tope". Si quisiéramos quitar 5 elementos es más fácil indicar que el tope está 5 elementos más abajo a ejecutar 5 operaciones "pop"; por supuesto todos los elementos han de ser del mismo tamaño.

Si imaginamos como se añaden y eliminan elementos, veremos que el último elemento que entra es el primero que sale. Si metemos por ejemplo los valores 1, 2, 3, 4, 5, al sacarlos obtendremos 5, 4, 3, 2, 1.

Además en equipos x86 la pila tiene otra peculiaridad: crece hacia abajo. Esto hace que sea más confusa, si creamos una pila para elementos de 1 byte en la dirección de memoria `0xff`, cuando añadamos un elemento el tope bajará hasta la dirección `0xfe` y cuando lo quitemos el tope volverá a `0xff`.

Las CPUs x86 tienen registros específicos para trabajar con pilas y hasta un segmento dedicado (*SS* o "stack segment"). Por ahora es suficiente decir que el registro *rsp* (*esp* para 32 bits) siempre apunta al tope de la pila (la dirección más baja). Además este registro puede usarse como punto de referencia y podremos leer datos relativos como por ejemplo, 3 elementos respecto de *rsp*.

Esta sección parece sacada de un libro de algoritmos y parece extraño que un procesador tenga que saber de pilas. La razón de esta necesidad es lo que se conoce como *"call stack"* (podríamos traducirlo como "pila de llamadas").

Los lenguajes de programación tienen llamadas a funciones que es algo parecido a las operaciones matemáticas pero con nombre. Para sumar dos números hacemos $a+b$, esto traducido a una función sería: `suma(a, b)`. En realidad las operaciones que realizan las funciones son más complejas: mapea esta página en esa dirección, lee de

un archivo, etcétera, además es habitual construir unas funciones sobre otras. Por tanto habrá funciones con menor grado de abstracción que harán tareas de nivel más bajo y funciones construidas a partir de ellas que tendrán un nivel de abstracción mayor.

Supongamos que el procesador está ejecutando una función y que se llama a otra, ¿cómo se gestiona algo así? Si lo pensamos es una situación que encaja perfectamente con una pila. Lo que se hace es poner en la pila los parámetros que le pasamos a la función[1] (los operandos) antes de llamarla y programar la función de forma que cuando termine deje la pila como estaba. Esto funciona bien si esa función llama a otra o a otras y esas a su vez llaman a más y así sucesivamente (siempre que no se supere el espacio asignado a la pila).

Desde el punto de vista del sistema, ¿en que dirección de memoria hemos de colocar esta pila? En principio cualquier dirección del proceso es válida, lo único que hemos de tener en cuenta es que crece hacia abajo, y si la ponemos detrás del código, hemos de dejar suficiente espacio para que quepa, ya que si creciera demasiado podría intentar escribir en el segmento del código, y dado que este segmento se marca de sólo lectura, se causaría un fallo de protección (en un caso así el kernel termina el proceso pero si esto nos ocurre en el espacio de kernel, el resultado sería impredecible). Este problema se complica aún más con el uso de hilos ya que cada hilo tiene un stack de ejecución distinto.

Un recurso muy bueno relacionado con esta sección y la siguiente es [x86] ya que aunque está orientado a desensamblar binarios, explica la pila en profundidad.

2.12. Convenciones de llamada y "stack frames"

"Stack frame" se podría traducir como "marco de pila" pero es complicado buscar una traducción totalmente satisfactoria y "convenciones de llamada" viene del inglés *"calling conventions"*.

En la sección anterior se vio la necesidad de usar una pila para las llamadas entre funciones; ahora vamos a ver como se colocan en memoria los distintos elementos. Dado que las funciones reciben y devuelven parámetros es un poco confuso.

Un *"stack frame"* es la parte de la pila asociada a una función así que por tanto, la pila de llamadas se compone de *"stack frames"*.

Conocer como funciona la pila es de vital importancia para implementar las interrupciones. El procesador salta a una dirección de memoria y hemos de ser capaces de

[1] Esto no siempre es cierto ya que como optimización algunos parámetros se pasan en registros.

regresar al punto donde estábamos antes de que esa interrupción llegase y de dejar los registros como estaban. También es importante saber cómo funciona para poder usar el lenguaje C. En una aplicación de usuario el stack lo inicializa el kernel, en este caso hemos de ser nosotros los que elijamos donde ponerlo y llamemos después a lo que será nuestra función "main"[2].

También encontraremos *stack frames* cuando un sistema operativo multitarea cambia de proceso ya que ha de cambiar también el *stack* en uso, y cuando se pasa de modo usuario a modo kernel y viceversa ya que el kernel y los procesos tienen *stacks* en uso diferentes.

Antes de examinarlo, hay que decir que la forma de colocar los elementos en el *stack* es una convención que depende del sistema y compilador. Varias de estas convenciones son: *cdecl*, *stdcall*, *fastcall*, *Optlink*, etcétera.

Cuando AMD sacó los primeros procesadores con soporte para la arquitectura x86-64, también publicó un documento para intentar estandarizar la forma de llamar a las funciones, este documento es [MHJM].

En lugar de explicar directamente como funciona una pila de un sistema operativo, resulta más instructivo usar el compilador, descompilador y depurador y examinar el código que genera.

Listado 2.13: Programa de ejemplo para examinar el stack.

```
/* Compilar con:
 * gcc -O2 -Wall -pedantic -o stack stack.c */

#include <stdio.h>

int
f (char *s)
{
        asm ("nop;");
        puts(s);
        asm ("nop;");
        return 0;
}

int
main (int argc, char **argv)
{
        f("Hola mundo.");

        return 0;
```

[2] "main" es la función de C donde comienza la ejecución.

```
}
```

En el listado 2.13 tenemos el clásico programa "hola mundo" que suele ser el primer programa que se hace al aprender un lenguaje de programación. Se ha modificado para que en lugar de llamar directamente a la función que imprime, llame a `f()` con la cadena para que a su vez llame a la función que imprime.

Si desensamblamos el código y vemos la parte donde está la función `main()`, tenemos:

Listado 2.14: Main desensamblado

```
objdump -D stack
...
400415:       bf 1c 06 40 00          mov    $0x40061c,%edi
400419:       e8 f1 00 00 00          callq  400510 <f>
...
```

Como vemos hay tres columnas, la primera es la dirección de memoria virtual (en hexadecimal) donde se cargará el programa (podemos ver qué el kernel usa *higher half*, si no fuera así, dado que esta memoria es virtual y el binario se ha compilado para 64 bits, sería una dirección mucho más larga). La segunda son los números u *opcodes* (también en hexadecimal) de las instrucciones que se indican en la tercera columna. Este desensamblador tiene como característica que puede poner los mnemónicos y además resuelve símbolos, por eso tenemos respectivamente las instrucciones de ensamblador y etiqutas como en este caso "f".

La primera instrucción mueve una dirección de memoria a un registro de 32 bits. "Mover dirección de memoria a registro *edi*", es la instrucción *bf* seguida de los 32 bits de la dirección[3]. Lo primero que sorprende es que la dirección está especificada al revés con dígitos tomados de dos en dos, es decir: `1c064000` en lugar de `0040061c`, veremos después el motivo.

Si examinamos esa dirección en el binario, vemos que contiene unos bytes:

```
48 6f 6c 61 20 6d 75 6e 64 6f 2e 00 00 00 .....etc.
```

Y como muestra el descompilador, los bytes están en la sección *".rodata"*. Esa sección se usa, como su nombre indica, para datos de sólo lectura. Mediante un comando podemos intentar convertirlos a texto:

[3] A pesar de ser un binario de 64 bits, el puntero es de 32 bits porque el compilador ha generado código para un modelo de memoria "small", ver `http://en.wikipedia.org/wiki/Intel_Memory_Model`. Un análisis de esta característica resulta excesivo.

Listado 2.15: Conversión de hexadecimal a cadena.

```
~$ echo 48 6f 6c 61 20 6d 75 6e 64 6f 2e 00 |\
       perl -ne 's/\s//g; print pack "H*", $_'
Hola mundo.
~$
```

Luego, mueve la dirección de memoria donde se encuentra la cadena "Hola mundo." a un registro.

Si examinamos el documento de AMD indicado ([MHJM]), vemos que el primer parámetro de las funciones se pasa por el registro *rdi*. Luego ya sabemos de que se trata esa instrucción: está preparando el primer parámetro[4].

En cuanto a la segunda instrucción si vemos el manual de la CPU, lo que hace `call` es: push de la dirección de la instrucción posterior en la pila y salta a la dirección indicada que como se ve, es la dirección de la función[5].

Ahora ya sabemos como llamar a las funciones. Si tuvieran más parámetros, se pasarían en este orden: *rdi*, *rsi*, *rdx*, *rcx*, *r8*, *r9* y el resto por la pila en orden inverso (nótese que pasarlos por la pila es más lento que por un registro ya que por un registro se evita acceder a memoria).

El código generado por el compilador para la función contendrá tres partes: el prólogo, que es el código que genera el compilador antes de nuestra primera instrucción, el código ensamblador generado para nuestro código C y el epílogo que es el código que emite el compilador para dejar la pila como estaba y regresar a la función que nos llamó. Para poder distinguir las distintas partes he añadido dos instrucciones *nop* justo al principio y al final de la función. El código ensamblador generado queda por tanto:

Listado 2.16: Una función desensamblada

```
...
0000000000400510 <f>:
  400510:       48 83 ec 08             sub    $0x8,%rsp
  400514:       90                      nop
  400515:       e8 d6 fe ff ff          callq  4003f0 <puts@plt>
  40051a:       90                      nop
  40051b:       31 c0                   xor    %eax,%eax
  40051d:       48 83 c4 08             add    $0x8,%rsp
```

[4] Aunque no es totalmente relevante para la discusión, en este momento resulta sumamente fácil explicar la diferencia entre paso por valor y por referencia. En este caso está pasando el parámetro por referencia y la función podría modificar el dato, para pasarlo por valor habría de sacar una copia de la cadena en la pila

[5] De nuevo, con esta discusión es fácil comprender que es un puntero a función en C, un concepto importante en el desarrollo de un sistema operativo y que se hace difícil a los programadores novatos.

```
  400521:       c3                      retq
...
```

Lo primero que hace es sumar 8 a la pila (*sub* significa restar pero recordemos una vez más que la pila crece hacia abajo). Esta parte es confusa por lo que necesitamos mirar el estándar. En la sección en la que explica los "*stack frames*", dice (traducción libre) "el final de los parámetros de la función estará alineado a un múltiplo de 16 bytes. En otras palabras, el valor `rsp + 8` siempre es un múltiplo de 16 cuando el control se transfiere a la función. El puntero de pila `rsp` siempre apunta al final del último stack frame reservado.".

Vamos a usar el depurador GDB para ver si es el caso.

Listado 2.17: Depurando la pila con GDB

```
(gdb) b f
Breakpoint 1 at 0x400510
(gdb) r
Starting program: stack

Breakpoint 1, 0x0000000000400510 in f ()
(gdb) info registers rsp
rsp            0x7fffffffe5a8   0x7fffffffe5a8
```

La dirección `0x7fffffffe5a8` no es múltiplo de 16 bits por lo que tal vez esté reste 8, para que lo sea. El manual de optimización de Intel ([corh]) nos dice que por razones de rendimiento, las instrucciones SIMD ("*Single Instruction Multiple Data*", se refiere a MMX, SSE, SSE2, SSE3 y a cualquier otra instrucción que use varios datos simultáneamente), necesitan sus operandos alineados en múltiplos de 16 bytes, y además en el apéndice añade que es importante que el stack frame se alinee en una dirección múltiplo de 16 bits.

Si recompilamos el programa con el flag: `-mpreferred-stack-boundary=5`, podemos comprobar que la pila se alinea a una dirección de 32 bytes:

Listado 2.18: Prólogo alineado de una llamada a función

```
  400510:       48 83 ec 18             sub    $0x18,%rsp
```

Es decir, resta 24 a la dirección y como podemos comprobar en el depurador, efectivamente el stack queda alineado a una dirección múltiplo de 32 bytes.

Ahora pasamos al código generado para gestionar nuestra llamada a la función de biblioteca. En este caso se trata de otra función que recibe una cadena por lo que

no hace nada dado que ya tiene la dirección de la cadena en el registro adecuado, simplemente ejecuta la llamada a la función.

Para saber la dirección de la función, ya veremos más adelante, lo que hace es dejar la referencia vacía, y en el momento de ejecución el programa que carga nuestro binario, pone en memoria la biblioteca de C y resuelve la dirección del símbolo.

Si por curiosidad queremos saber por que GCC añade `@plt` detrás del nombre de la función, la razón es lo que se conoce como *"procedure linkage table"*, que es una optimización para poder compartir los mapeos del código de los binarios entre distintos procesos con la biblioteca de C mapeada en direcciones de memoria distintas (las bibliotecas se pueden compartir aún cargadas en direcciones distintas ya que el código generado se llama PIC o "código independiente de la posición", del inglés "*position independent code*"). Si un proceso carga la biblioteca de C en una dirección y el mismo proceso se ejecuta simultáneamente y la carga en otra, el código no se podría compartir puesto que las llamadas a las funciones de biblioteca se resolverían a direcciones diferentes.

Para que esto funcione se añade una sección privada a cada proceso de forma que los binarios, en lugar de llamar a la función de biblioteca, llamen a una función con el mismo nombre definida en la sección privada, y que esa función resuelva la dirección de memoria adecuada para ese proceso.

Para terminar pasamos al epílogo. La instrucción `xor` hace un "o exclusivo de bits" entre sus parámetros. Al tener el mismo parámetro dos veces, lo que hace es poner a cero el registro dado que cero "*y exclusivo*" cero es siempre cero y uno "*y exclusivo*" uno, también es siempre cero. Usa este sistema en lugar de usar: `movl $0,%eax` porque es una forma más optimizada de hacerlo (requiere menos ciclos de procesador). La razón por la que pone este registro a cero es porque es la forma usada para devolver un dato desde la función.

La siguiente instrucción restaura la pila como estaba cuando se entró en la función, y la última instrucción está específicamente hecha para regresar de las funciones. Es una función que según el manual equivale a `pop%rip`, es decir, recordando las operaciones de pila, saca el último elemento de la pila y lo coloca en el registro *rip*. Como vimos, la instrucción *call* hizo un *push* de la dirección de la instrucción que iba detrás de ella, luego al hacer *pop* en el registro de instrucción, saltamos a esa dirección de memoria y la ejecución sigue por el camino que llevaba antes de llamar a la función, pero probablemente con resultados colaterales como un registro o una dirección de memoria modificados o algún cambio de estado en el procesador.

2.13. Endianness

Como vimos en la sección 2.12, la dirección que se pasaba como parámetro tenía los bytes invertidos.

Sin entrar en muchos detalles, existen dos tipos de arquitecturas: *"big endian"* y *"little endian"*. x86-64 es una arquitectura *little endian* y los datos multibyte se escribirán con el byte menos significativo primero.

Por ejemplo, si tenemos el dato de 16 bits: `0x01ff`, en memoria se colocaría así:

```
[_______16_______]
[___8___][___8___]
   ff       01
```

La mayoría de las veces nos interesa sólo el dato completo, pero en ocasiones habrá que tenerlo en cuenta.

2.14. BIOS

Aunque la BIOS es un sistema ajeno al procesador, es un buen momento para ver que papel juega.

Cuando encendemos un procesador, este salta a una dirección de memoria. Los PCs tienen esa dirección concreta conectada a una memoria que contiene un programa que se conoce como BIOS.

Este programa identifica que dispositivos hay conectados, los inicializa, localiza cuales permiten arrancar un sistema operativo y según el orden que tenga configurado, intenta arrancar desde ese dispositivo. La BIOS también configurará el reloj, permitirá desconectar (como si se hubiera hecho físicamente) dispositivos y permitirá opciones de seguridad como preguntar por una contraseña en el arranque.

Además de estas funciones, también proporciona unas rutinas a los sistemas operativos para hacer cosas básicas, tales como informar sobre la cantidad de memoria conectada, leer una pulsación del teclado o escribir un bloque en un diskette.

En principio parece que podríamos usar esos servicios en lugar de implementarlos nosotros, pero en la práctica no es muy buena solución (los sistemas operativos DOS tales como MS-DOS funcionaban así). Las distintas BIOS presentan incompatibilidades y sólo ofrecen acceso a los dispositivos que conocen, además se trata de un

acceso básico (por ejemplo, no podríamos acceder a un disco usando la última versión de DMA, lo cual es mucho más rápido).

En la práctica la BIOS se usará para pocas cosas que no se pueden hacer de otra forma.

2.15. Dispositivos varios

Para completar la idea de funcionamiento de un PC vamos a comentar muy por encima los distintos dispositivos que lo componían originalmente.

Cuando se habla de un PC todo el mundo sabe lo que es pero en realidad ese nombre viene del ordenador personal que de IBM sacó a la venta en 1981. Este ordenador y sus siguientes versiones (cabe destacar XT del año 1983, AT de 1984 y Personal System/2 de 1987) tenían unas características básicas que aún poseen los ordenadores modernos.

Por tanto cuando el sistema operativo arranca, ha de buscar unos dispositivos básicos y mínimos, y a partir de ahí ha de buscar dispositivos más potentes que los reemplacen.

Un ejemplo muy típico es el dispositivo de video. Cuando el ordenador se enciende el video suele estar en un modo básico (algunas BIOS modernas cambian a modos de video más potentes pero cuando carguen el sistema operativo, este ha de encontrarse los dispositivos como espera). El sistema operativo ha de probar distintos modos hasta encontrar el que quiera usar según la configuración de usuario o si no la hay, lo normal es que escoja el más potente.

Este papel corresponde a los drivers, y si no disponemos de un driver para un dispositivo, no podremos usarlo. En el caso de dispositivos que venían en los PCs (video, teclado, disco duro), se usará una versión más básica con menos funcionalidad, en el caso de un dispositivo más nuevo, no podremos usarlo.

2.16. CPUID

En todo momento podemos descubrir las características de la CPU en la que trabajamos mediante la instrucción: *cpuid*. Ahora que tenemos una visión completa del sistema, vamos a hacer un programa que lea información de la CPU mediante esta instrucción y así introducimos algún concepto más.

La instrucción recibe un parámetro en el registro *eax* con la información que que-

remos, y devuelve el resultado en los registros: *eax*, *ebx*, *edx* y *ecx*.

Lo que vamos a leer del procesador es el nombre del fabricante. Para ello le pasamos en *eax* el valor "0" y recogemos en los registros mencionados los caracteres que forman la cadena.

Además vamos a introducir otra característica del compilador GCC: *"inline assembly"* (aunque se vio una forma básica con la instrucción *nop*, no intercambiaba parámetros y la instrucción no hace nada). Se trata de decirle al compilador que inserte en un punto dado instrucciones directamente en ensamblador. La ventaja de esta aproximación es que C puede insertar ensamblador en cualquier punto sin tener que enlazar con un archivo externo, además se podrán intercambiar variables entre uno y otro lenguaje de programación y se podrá permitir que el compilador escoja los registros del procesador para una operación (siempre que cumplan los requisitos que impongamos).

Lo primero va a ser poner las variables en una estructura para que se nos coloquen como queremos en memoria. Después hacemos una función que copie las variables en los campos de la estructura. Invocamos a la función con el parámetro cero y recogemos los resultados directamente de los registros. Comento las lineas más importantes:

Listado 2.19: CPUID

```
/* Para tener el prototipo de printf: */
#include <stdio.h>

/* Podremos tener los mismos datos como 4 enteros o como cadena.
 * El caracter c de la segunda estructura es para el caracter
 * terminador de cadena ya que sino habría que copiar el dato.
 * Nótese que las estructuras anónimas es una extensión de GCC
 * que será adoptada por el próximo estándar de C. */
union _cpuregs32 {
        struct {
                unsigned int eax;
                unsigned int ebx;
                unsigned int edx;
                unsigned int ecx;
        };
        struct {
                unsigned int seax;
                char s[12];
                char c;
        };
};

/* Esta es la parte mas complicada.
```

```
 * inline quiere decir que en lugar de generar una llamada,
 * inserta su código 'en linea'. Se suele hacer para funciones
 * pequeñas cuyo prólogo y epílogo pesarían más que el código
 * que contienen. Hay que tener en cuenta que cada vez que se
 * llame, se hará una copia en memoria de la función. */
static inline void
cpuid (union _cpuregs32 *r)
{
        /* Con "asm", el compilador copia la cadena tal cual al
         * generar el ensamblador. Es necesario poner \n porque
         * podría haber otra instrucción en la misma linea y no
         * quedarían separadas. Para mas información ver:
         * http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html */
        asm ("cpuid\n\t"
                /* Parametros de salida. Es posible que no se
                 * modifiquen algunos. Dependerá de la función
                 * de CPUID con que nos llamen. */
                : "=a" (r->eax), "=b" (r->ebx),
                  "=d" (r->edx), "=c" (r->ecx)
                /* Unico parametro de entrada: */
                : "0" (r->eax));
}


int
main (int argc, char **argv)
{
        union _cpuregs32 c;

        /* Indicamos que queremos la operación 0 (eax = 0). */
        c.seax = 0;
        cpuid (&c);
        /* Terminador de cadena. */
        c.c = '\0';

        printf ("Nuestro fabricante es: %s\n", c.s);

        return 0;
}
```

Capítulo 3

Primeros pasos

3.1. Introducción a ELF

ELF es uno de los formatos existentes para ejecutables, código objeto, bibliotecas dinámicas y volcados de memoria. Se trata de un formato muy usado y los binarios de casi todos los UNIX se generan en formato ELF.

Además, aunque podría haberse usado cualquier formato, el archivo que contiene el código de ¡OS es a su vez un archivo ELF.

Los archivos ELF se componen de secciones. Para no entrar en más detalles que los necesarios, vamos a examinar el binario generado para ¡OS mediante la herramienta *"readelf"*, estas son las lineas relevantes:

```
$ readelf -a jOS
...
  Entry point address:               0x100024
...
Section Headers:
[Nr] Name              Type             Address           Offset
     Size              EntSize          Flags  Link  Info  Align
...
[ 2] .text             PROGBITS         ffff800000105000  00006000
     0000000000008000  0000000000000000 WAX       0     0     32
...
[ 4] .data             PROGBITS         ffff80000010e500  0000f500
     0000000000000b00  0000000000000000  WA       0     0     32
```

```
[ 5] .bss              NOBITS           ffff80000010f000  00010000
     0000000000004f08  0000000000000000  WA       0     0     32
...
```

En un ejecutable, el "*entry point*" es la dirección de memoria a la que tendrá que saltar la CPU que lo ejecute una vez cargado. Para simplificarlo, será la dirección donde esté `main()` (aunque en realidad el punto de entrada suele ser `_start`). En el caso del jOS, es fácil comprobar mediante *objdump* que hay en la dirección del *entry point*:

```
0000000000100024 <_start>:
  100024:    fa                      cli
  100025:    89 1d 20 00 10 00       mov    %ebx,0x100020(%rip)
  10002b:    0f 01 15 08 01 10 00    lgdt   0x100108(%rip)
...
```

No sorprendentemente es el mismo código ensamblador que carga el sistema operativo.

Lo que le sigue son algunas de las secciones del binario. Las secciones son como compartimentos en los que se colocan los distintos elementos que lo componen. Estas son las más importantes:

- `.text`: en esta sección va el código ejecutable del programa, es decir, los cuerpos de las funciones.
- `.data`: en esta sección se colocan los datos globales inicializados y los definidos como static dentro de una función. Por ejemplo cuando fuera de cualquier función hacemos: `int var = 1;`, el compilador coloca la información en la sección `.data`.
- `.bss`: los símbolos que se definan en esta función se inicializan a cero en el momento de cargar el archivo ELF. La ventaja es que si reservamos 4KB en la sección `.data`, el binario ocupará 4KB más, pero si lo reservamos en `.bss`, sólo ocupará tanto más como ocupen las directivas que indican al cargador que cree la variable. El compilador de C colocará en esta sección las variables globales no inicializadas, por ejemplo cuando hacemos: `int var;` fuera de cualquier función (en caso de colocar esta directiva en una función se usaría la pila ya que la función sería local y sólo existiría en tanto en que la función se estuviera ejecutando).

Además hay secciones para depuración (gracias a ellas podemos usar gdb y ver a que código se corresponden las instrucciones), para enlace dinámico, para distintos usos del compilador (por ejemplo constructores y destructores de C++), para la tabla de símbolos, etcétera, y otras que podemos crear nosotros. Podemos ver información sobre ELF en las especificaciones ([is] para 32 bits y [HP/98] para 64 bits) y también en [Lev99].

3.2. GNU Assembler

Como hemos dicho, el ensamblador que usaremos es GNU Assembler (GAS). En general ¡OS no usará muchos archivos en ensamblador porque GCC permite incluir instrucciones "*inline*" cuando éstas hagan falta.

GAS es un ensamblador potente, que provee todas las características que necesitaremos, está bien documentado y probado y es software libre, por lo que está disponible sin coste alguno y en código fuente.

Aunque permite usar sintaxis Intel y AT&T, ¡OS escogerá esta última opción ya que es el formato nativo del ensamblador y de UNIX. Ambos tipos de sintaxis son iguales en cuanto a características por lo que la elección de uno u otro es una cuestión personal o del software que se use (aunque lo normal es usar AT&T si se usa este ensamblador). Podemos ver un artículo interesante que compara ambos tipos de sintaxis (para la sintaxis Intel no usa GAS sino NASM: `http://www.nasm.us/`), disponible en en `http://www.ibm.com/developerworks/linux/library/l-gas-nasm/index.html`. Quizás la mayor desventaja es que es menos común que la sintaxis de Intel y por tanto los ejemplos y documentación son menos habituales.

Vamos a comentar algunas características del ensamblador de las que haremos uso para arrancar el sistema operativo, es la forma más sencilla ya que seguir el código comentado es mucho más fácil que escribirlo.

La primera es la extensión de los archivos, cuando el ensamblador encuentra un archivo con extensión `.S` (mayúscula), pasa el archivo por CPP, el preprocesador de C[1] antes de ensamblarlo para generar el código objeto.

Pero las macros del preprocesador no son las únicas que podemos usar porque GAS también nos proporciona las suyas. Vemos un ejemplo de macro GAS:

Listado 3.1: Ejemplo de macro GAS que mapea un PDE

[1] Aunque no hemos hablado del preprocesador, es un programa que acompaña a todos los compiladores de C, que aplica macros (sustituciones) sobre el código, proceso que hace buscando y sustituyendo. Reconoceremos las lineas del preprocesador porque empiezan por un símbolo `#`.

```
.macro map_pde pde_ptr, base_and_flags
        movl \pde_ptr, %edi
        movl \base_and_flags, %eax
        movl $512, %ecx
1:
        movl %eax, (%edi)
        addl $0x200000, %eax # cada página mapea 2MB
        addl $8, %edi        # siguiente entrada, 8 bytes detrás
        loop 1b
.endm
```

En el listado 3.1 podemos ver una macro que mapea un PDE con páginas de 2MB (proceso que se vió en la sección 2.9).

La directiva `.macro` indica que empieza una macro y `.endm` indica el final. `map_pde` es el nombre de la para que podamos referirnos a ella y en la misma linea están los parámetros. `pde_ptr` referencia una dirección de memoria con espacio libre (necesitará 4KB) donde se rellenará una tabla con los mapeos y `base_and_flags` será la dirección base a partir de la cual se mapearán las 512 páginas de 2MB (1GB de memoria).

Para mapear la dirección virtual 0 en la física 0 y así hasta el primer GB de memoria, usaríamos la macro así: `map_pde $pde, $0x87`[2].

Después se inicializa la dirección física en *eax*, y la dirección del PTE en *edi*, además usa *ecx* como contador. Ahora vemos a que pseudocódigo equivale la macro, pero destacamos antes un par de cosas más.

Primero el paréntesis alrededor del registro *edi*, que quiere decir: "dirección apuntada por el registro" (es decir, no copia al registro sino a la dirección a la que apunta). El otro detalle es el "1:", que es simplemente una etiqueta a la que nos podemos referir. Cuando se compile se sustituirá por la dirección de memoria que se asigne a la instrucción que la sigue. La instrucción "*loop*" salta a la etiqueta indicada pero en este caso es especial; cuando usamos un número seguido de "b", el ensamblador busca una etiqueta con ese número hacia atrás, y cuando el número va seguido de una "f", el compilador busca una etiqueta con ese número hacia delante en el código. Dado que no nos referiremos a esa etiqueta desde ninguna otra parte del código (y mucho menos desde fuera del archivo), tiene sentido usar una etiqueta así. Es recomendable evitar exportar símbolos ya que éstos son visibles para el linker y además de que se ralentizaría mínimamente el programa, aumentarían las posibilidades de conflictos no deseados (dos etiquetas con el mismo nombre) y de que se usaran interfaces que no fueron diseñados

[2]Aunque se verá a continuación, se recomienda como ejercicio investigar donde se pasa la dirección física.

para ser exportados a otros módulos[3].

Podemos ver el pseudocódigo de la macro 3.1 en el listado 3.2.

Listado 3.2: Pseudocódigo de la macro

```
edi = dirección de tabla que queremos rellenar
eax = dirección física
ecx = 512
mientras ecx > 0
        # sin paréntesis, la siguiente linea sería: edi = eax
        *edi = eax
        eax += 2MB
        edi += 8 # pasa al siguiente elemento de la tabla
        ecx--
fin mientras
```

Comparando con la macro en ensamblador, vemos como los lenguajes de programación nos facilitan mucho las cosas.

Además hay otro tipo de macros para valores constantes, similares a los *define* del preprocesador de C:

Listado 3.3: Constantes en GAS

```
; una constante normal:
.set VAR, 0x12345678
; constante con flags:
.set VAR2, 1<<2 | 1<<5
```

GAS también tiene algunas directivas para controlar el código (este código se muestra como ejemplo y en su conjunto no tiene sentido):

Listado 3.4: Directivas en GAS

```
        .code32
        .section mboot
        .align 4
        .data
        .global _label
_label:
        .long var

        .lcomm var, 0x1000
```

[3]Aunque C y mucho menos ensamblador no impiden que el programador pueda saltarse cualquier tipo de interfaz, es aconsejable mantener alguna convención similar a la que imponen sobre todo los lenguajes orientados a objetos.

La directiva `.code32` indica al ensamblador que genere código de 32 bits, igualmente puede usarse: `.code64`, `.code16`, etcétera.

`.section mboot` indica que lo que sigue ha de colocarse en una sección ELF con el nombre que se señala. Poco más abajo, `data` es lo mismo pero como se refiere a una sección estándar, tiene un nombre reservado.

El ensamblador, cuando genera el binario recorre el archivo y va asignando direcciones a las directivas y al código. `.align 4` indica que se han de saltar posiblemente bytes hasta que la dirección de lo que sigue sea múltiplo de 32 bits (4 bytes).

`_label:` es una etiqueta, la linea anterior contiene la directiva `.global`, que hace que la etiqueta sea visible fuera de la unidad en la que estamos (el archivo). Se usa para exportar variables, las cuales serán accesibles desde código C o desde otro archivo de código ensamblador ya que el linker será capaz de verlas y resolverlas. Cabe destacar que el nombre empieza por un guión bajo ("_"). Algunos compiladores modifican el nombre de las funciones o variables al compilar; GCC añade guiones bajos por lo que la variable será visible desde código C como: `label`[4].

Desde C además es posible controlar el nombre de las variables generadas en ensamblador, por ejemplo: `int var asm ("mivariable") = 123;`, o para funciones poniendo una declaración: `extern f() asm ("funcion123");` antes del cuerpo. También con la opción de GCC: `-fno-leading-underscore` podemos decirle a GCC que no añada ese guión bajo. Empíricamente se comprueba de todas formas que las versiones actuales de GCC no imponen el uso de un registro concreto por lo que estas directrices son más consejos que órdenes; esto es especialmente cierto cuando aumentamos el número de registros impuestos al compilador.

La linea siguiente es la forma de reservar espacio para una variable. La linea: `.long var`, reserva espacio para un long (4 bytes) y crea el nombre var para que podamos referenciarlo. Que reserve espacio significa que crea (físicamente) el número de bytes indicados en la sección actual del binario donde pongamos la directiva.

Por último, `.lcomm var, 0x1000`, reserva 4KB de memoria. Según la documentación, ese espacio se reserva en la sección `.bss` por lo que no se hace reserva física. Tal y como se mencionó, la diferencia entre que haya reserva física o no, digamos en el caso de reservar 4KB, es que el binario ocupará 4KB más o no respectivamente (nótese que en realidad es menos ya que en alguna parte se ha de indicar que se reserven esos 4KB al cargar el binario).

Para el resto de características, tenemos disponible el completo manual de GAS:

[4]No es que esta variable se exporte como "label", es que a la variable de C se le añade el guión bajo delante.

[EFf].

3.3. Gestor de arranque

En la sección 1.5 se comentó el trabajo necesario para carga un sistema operativo. La BIOS lee el primer sector del disco que esté configurado como dispositivo de arranque y lo copia en una dirección estándar de memoria. Después "salta" (es decir, hace que el registro *rip* apunte) a esa dirección y por tanto se ejecuta el programa que haya cargado ahí previamente.

En el caso de un PC y un disco duro, el primer sector se conoce como MBR (de inglés *"master boot record"*) y el sistema operativo habría de poner en ese sector código suficiente para leer en memoria la imagen del kernel y para saltar a ella.

Ahora pensemos, ¿en qué dispositivo instalamos la imagen de nuestro kernel?, ¿qué sistema de archivos tendrá?

El primer dispositivo ha sido tradicionalmente un diskette o un disco duro aunque también puede ser un disco compacto o una memoria USB. En la mayoría de los casos, un sistema arrancará desde una partición del disco duro por lo que para encontrar nuestra imagen habrá que calcular la posición relativa al inicio de la partición.

La elección del sistema de archivos es más compleja, pero sea el sistema de archivos que sea, el sector de arranque ha de encontrar la localización en disco del kernel y leerlo. Por tanto ha de tener conocimientos del sistema de archivos y del dispositivo que lo contenga (un mini-driver para diskette, disco duro IDE, disco duro SCSI o similar, y para el sistema de archivos). Esto ha de hacerse en 512 bytes de código ensamblador (podría usarse C pero 512 bytes es muy limitado y C crece más de lo que controlamos, además la mayoría de instrucciones serán de bajo nivel y C no las provee por lo que tiene sentido usar ensamblador).

También hay que tener en cuenta otro detalle: el sector de arranque tiene que ser actualizado cada vez que tengamos una nueva versión del kernel (o bien el kernel tiene que ser puesto en el mismo lugar que el anterior, pero es complicado que el sistema de archivos nos proporcione esta funcionalidad).

El cargador es inevitable pero hay una opción para que el programador del sistema operativo se lo pueda ahorrar: usar una herramienta ya hecha.

Un programa muy usado es GRUB, el cual además de todo este trabajo es capaz de cargar desde disco (soporta una docena de sistemas de archivos) código para mostrarnos un menú con los sistemas operativos instalados. Este programa también cambia el

procesador a modo de 32 bits, detecta la memoria instalada en el equipo, pone modos de video, etcétera. GRUB viene a ser un pequeño sistema operativo.

El interfaz que proporciona a la imagen que carga de memoria está bien documentado en una especificación que se llama Multiboot, la cual puede encontrarse en [ea] (este interfaz se proporciona por conveniencia ya que aunque no es necesario que la imagen coopere, la funcionalidad que proporciona resulta útil).

Grub puede cargar cualquier formato binario, pero viene especialmente preparado para archivos ELF de 32 bits.

En el desarrollo de ¡OS, al ser de 64 bits, supuso un problema y se probaron varias posibilidades, se detallan a continuación:

- Generar un archivo ELF32. No funcionó porque las direcciones de memoria se truncaban a 32 bytes.
- Generar un ELF de 64 bits. Esta es en principio la mejor opción pero GRUB no soportaba en ese momento archivos ELF de 64 bits.
- Generar un archivo ELF de 32 bits a partir del código de 64 bits. Para ello se usó la herramienta "*objdump*" que permite convertir archivos objeto entre formatos.
- Usar otra especificación como Multiboot2. Una especificación muy completa hecha para GRUB2 pero no implementada en código, o mejor dicho, parcialmente implementada (esto se comprobó examinando el código de GRUB2)[5].
- Usar especificación Multiboot en GRUB2, que soporta binarios de 64 bits sin ningún problema.
- Otras muchas posibilidades que no funcionaron, como por ejemplo usar otro cargador distinto de GRUB.

Probar todas estas posibilidades llevó probablemente más tiempo del que se habría invertido en hacer un cargador propio.

Trabajar con "*objdump*" funcionó, pero al desensamblarlo teníamos código de 64 bits metido en un binario de 32 bits por lo que el código no se podía ver. Es una situación incómoda porque a veces sirve de ayuda ver el código generado (podrían haberse dejado los dos binarios, uno para desensamblar y otro para ser copiado pero es confuso puesto que el archivo ELF contiene simultáneamente código de 32 y de 64 bits

[5]La situación puede haber cambiado y tal vez ya exista una especificación Multiboot2 completa.

(el necesario para pasar la CPU a 64 bits tiene que ser forzosamente de 32 bits) así que nunca podríamos listar ambos tipos de código simultáneamente).

GRUB2 si nos solucionaba el problema pero dado que ¡OS se desarrolla para una plataforma virtual, había que instalarlo en una imagen de disco, algo para lo que GRUB2, entonces en desarrollo, no estaba preparado. Durante las primeras versiones se encadenó (proceso que se conoce como "*chainloading*") con GRUB, es decir, se instala GRUB en el master boot record y se le dice que cargue el primer sector de una partición concreta, en esa partición se pone GRUB2 y el sistema arrancó.

Trabajar con buenas herramientas es fácil pero en el momento de empezar a desarrollar ¡OS, los equipos x86-64 no eran excesivamente comunes por lo que no existía tanta documentación y adaptación de software para esta plataforma. Ningún otro cargador proporcionaba arranque para binarios ELF de 64 bits y los que lo hacían no eran tampoco triviales de instalar en una imagen ni estaban bien documentados (hay que aclarar que los cargadores no son excesivamente populares por lo que no hay muchas opciones).

Esta solución mixta se usó al principio pero la solución final elegida fue generar un ELF de 64 bits (con secciones de 32 bits para código de 32 bits) y usar lo que se conoce como "*a.out kludge*". Esto nos dio lo mejor de los dos mundos: tenemos un binario de 64 bits (*objdump* nos muestra tanto el código de 32 como de 64 bits perfectamente) y es posible cargarlo desde GRUB y desde GRUB2.

"a.out kludge" se podría traducir como: "el parche a.out". Si vemos la especificación, Multiboot nos proporciona una forma de indicar los datos de arranque que necesita GRUB cuando no reconoce el formato de la imagen. Esto se hace poniendo un número mágico (del inglés: "magic number"[6]) en los primeros bytes del archivo (8192 primeros bytes), seguido de los datos necesarios.

Esta solución funcionó al principio, pero en cuanto el binario creció, el linker colocó el archivo objeto donde se define el número mágico detrás del límite de bytes impuesto por la especificación y el binario dejó de arrancar (como anécdota, esto ocurrió al cambiar el orden de los argumentos pasados al linker).

Para asegurarnos de que el número mágico está dentro de ese límite, toda la información Multiboot se colocó en una sección ELF y en el script usado para enlazar el código (dado que nuestros requisitos de enlace son especiales hay que usar necesariamente un script del linker) se colocó esa sección al principio.

En el listado 3.5 se muestra parte del código necesario para que GRUB pueda arran-

[6]En este contexto se conoce como números mágicos a las combinaciones de bits que son matemáticamente improbables de darse en otro lado de binario.

car el ¡OS.

Listado 3.5: Multiboot

```
        .set MB_MAGIC, 0x1BADB002
        .set MB_FLAGS, \
                1<<1  /* info de memoria */ |\
                1<<16 /* a.out kludge */

        .section .mboot

        .code32

        .align 4

mb_header:
        /* Grub magic */
        .long MB_MAGIC
        .long MB_FLAGS
        .long -(MB_MAGIC + MB_FLAGS)
        /* Para el kludge a.out: */
        .long mb_header
        .long mboot /* Inicio sección .text */
        .long bss - K_PAGE_OFFSET /* Fin de .data */
        .long eok - K_PAGE_OFFSET /* Fin de .bss */
        .long _start /* Entry point */


        /* Para el puntero con información Multiboot */
        .global mbi32
mbi32:
        .long 0x0
```

En los flags hay que especificar que queremos el "*kludge*" de a.out (bit 16) y como hemos visto `.section .mboot` crea una sección. La cabecera tiene que alinearse a 4 bytes (GRUB busca el número mágico en direcciones múltiplo de 4 bytes) y las variables que se definen después es la información que necesita: donde empieza el código (*text* se refiere a "código" en este contexto), donde terminan los datos y bss y el *entry point*, que es la dirección de memoria a donde saltará GRUB. Como vemos, se indican variables y no valores, algunas serán etiquetas ensamblador y otras las rellena el script del linker.

Al final se reservan 4 bytes de memoria para guardar una dirección que nos deja GRUB en un registro de la CPU, que es un puntero a una estructura con información del sistema (los detalles de esta estructura vienen en la especificación Multiboot).

3.4. Linker

Ya se comentó brevemente la función de un linker en la sección 1.5, en este contexto el linker es necesario para juntar las piezas que componen el código fuente del ¡OS. Para esta tarea de enlace se usará el linker GNU LD, software de nuevo gratuito, potente y bien documentado.

Enlazar un programa normal no tiene ninguna dificultad, compilamos el código fuente y el compilador llama al linker con los parámetros adecuados. Compilar una biblioteca es algo más difícil ya que los parámetros necesarios los pasamos nosotros al compilador. Enlazar un sistema operativo es un proceso tan especializado que necesitaremos un script con las instrucciones.

Este es el script que enlaza ¡OS:

Listado 3.6: linker.ld

```
OUTPUT_ARCH(i386:x86-64)

ENTRY(_start)

/* Dirección (física) donde se cargará el kernel */
sok = 0x100000;


SECTIONS
{
    . = sok;

    .mboot :
    {
        mboot = .;
        boot*.o(.mboot)
        boot*.o(.text)
        boot*.o(.data)
        boot*.o(.bss)
    }

    . += page_offset;
    . = ALIGN(4096);

    .text : AT(ADDR(.text) - page_offset)
    {
        text = .;
        *(.text)
        *(.rodata*)
        . = ALIGN(4096);
```

```
    }

    .data : AT(ADDR(.data) - page_offset)
    {
        data = .;
        *(.data)
        . = ALIGN(4096);
    }

    .bss : AT(ADDR(.bss) - page_offset)
    {
        bss = .;
        *(.bss)
        *(COMMON)
    }


    /* Representa el final del kernel (dirección lógica). */
    eok = .;

    /*. = ASSERT(eok <= 0x400000,
          "linker.ld: Kernel demasiado grande!!!");*/
}
```

Para seguir el código es necesario entender que el carácter "`.`" se refiere a la dirección de memoria actual. Por tanto cuando hace: `. = sok` está diciendo: "la dirección de memoria actual será la indicada por la variable", que en este caso será la dirección en memoria física de carga del kernel e irá especificada en el propio script de enlace.

Después vemos que se indican varios elementos dentro de llaves. Cada uno de ellos representa a una sección ELF.

La primera es la sección `mboot`, que como vimos es para que el número mágico del cargador quede al principio de la imagen. La parte derecha de los dos puntos indica la dirección de memoria física si esta fuera distinta que la actual. Como en este caso sería la misma, no se pone nada. En el cuerpo se crea un símbolo: "mboot"; si nos fijamos en la cabecera hecha para GRUB en el listado 3.5, tenemos: `.long mboot`, que significa: "reserva espacio para un long cuyo valor está en la variable mboot", esta variable la rellena el linker y representará el principio del segmento de código. Las lineas que siguen copian secciones del archivo que coincidan con el patrón indicado, las secciones son las indicadas entre paréntesis.

Al terminar con la sección mboot, suma a la posición actual un valor, ese valor se define desde fuera del script y representará el desplazamiento que queremos dejar entre memoria física y memoria virtual, si es que el kernel es reubicable (si no lo fuera, el

desplazamiento sería cero). Justo una linea después, fuerza un alineamiento del código en una dirección múltiplo de 4KB.

Después copia las secciones `.text`, `.data` y `.bss`, estas tendrán una dirección de carga distinta de la dirección virtual, por eso tienen al final algo así: `AT(ADDR(.data) - page_offset)`.

Ya para terminar crea un símbolo: *"eok"* que referencia la parte final de nuestro binario y el principio de la zona de memoria usable por el kernel (necesario para que el kernel sepa donde termina y también lo necesita GRUB). La linea final que se encuentra comentada sería para abortar la compilación en caso de que el kernel superara el tamaño indicado. Aunque no se hace uso de esta característica, es posible que hayamos hecho suposiciones que no sean ciertas, como por ejemplo haber mapeado una cantidad de memoria inicial y que resulte insuficiente.

En este punto hay que decir que si examinamos el binario final podremos comprobar que todo se generó como queríamos. Veamos:

Listado 3.7: Comprobando las secciones con objdump.

```
0000000000100000 <mboot>:
...
Disassembly of section .text:

ffff800000105000 <kmain>:
```

Como vemos, la sección mbot se carga en una dirección virtual baja y el resto del kernel en una alta. Una vez desmapeada la *lower half*, este código ya no será accesible puesto que aunque lo sería con la dirección virtual, no se compiló para ser ejecutado ahí y lo más probable es que contenga instrucciones que suponen que están en cierta dirección de carga. Sería posible (el kernel de Linux lo hace así), liberar esa memoria o usarla para algo más, pero dado el tamaño de nuestro kernel, es muy poca cantidad y el código que implementara esa liberación también ocuparía espacio (tal vez más que el que usamos para pasar a la mitad alta del espacio de direcciones virtual).

A partir de ahora no será tan necesario conocer el linker GNU LD como las funciones generales de un linker. Para más información de esta sección tenemos el manual de GNU LD ([CT]).

3.5. boot.S

Aunque lo hemos ido poniendo poco a poco, a continuación reproducimos el archivo boot.S que es el que hace las tareas de arranque para el ¡OS. Casi todo ha de ser entendible con las explicaciones que se han ido dando y con los comentarios que contiene. El punto de entrada al kernel es la etiqueta: `_start`.

Listado 3.8: boot.S

```
#define K_CS 0x8
#define K_DS 0x10
#define K_SS 0x18

#define IA32_EFER 0x0c0000080


/* Mapea un PDE (512 entradas) con páginas de 2MB,
 * a partir de (base << 21 & flags) */
.macro map_pde pde_ptr, base_and_flags
        movl \pde_ptr, %edi
        movl \base_and_flags, %eax
        movl $512, %ecx
1:
        movl %eax, (%edi)
        addl $0x200000, %eax    # cada página mapea 2MB
        addl $8, %edi           # sgte. entrada, 8 bytes detrás
        loop 1b
.endm


        .set MB_MAGIC, 0x1BADB002
        .set MB_FLAGS, \
                1<<1 /* info de memoria */ |\
                1<<16 /* a.out kludge */

        .section .mboot

        .code32

        .align 4


mb_header:
        /* Grub magic */
        .long MB_MAGIC
        .long MB_FLAGS
```

```
        .long -(MB_MAGIC + MB_FLAGS)
        /* Para el kludge a.out: */
        .long mb_header
        .long mboot /* Inicio sección .text */
        .long bss - K_PAGE_OFFSET /* Fin de data */
        .long eok - K_PAGE_OFFSET /* Fin de bss */
        .long _start /* Entry point */


        /* Para el puntero con información multiboot */
        .global mbi32
mbi32:
        .long 0x0



        .global _start
_start:
        /* Deshabilita interrupciones globalmente */
        cli

        /* Guarda información multiboot */
        movl %ebx, mbi32

        /* Segmentos */
        lgdt gdt_ptr

        movw $K_DS, %ax
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %fs
        movw %ax, %gs

        /* El código que sigue es para poner la cpu en 64 bits. */
        movl %cr4, %eax
        bts $4, %eax /* PSE, Page Size Extensions */
        bts $5, %eax /* PAE, Tablas de páginas de 64-bits */
        bts $7, %eax /* PGE, Page Global enable */
        movl %eax, %cr4

        mov $IA32_EFER, %ecx
        rdmsr
        bts $8, %eax /* LME = 1. Enable long mode. */
        wrmsr

        /* Instala las tablas de páginas */
        map_pde $pml2, $0x87
        movl $pml2 + 0x7, pml3
```

```
        map_pde $pml2d, $0x9f + 0xc0000000
        movl $pml2d + 0x7, pml3 + 0x18
        movl $pml3 + 0x7, pml4
        movl $pml3 + 0x7, pml4 + 0x800

        movl $pml4, %eax
        movl %eax, %cr3


        movl %cr0, %eax
        bts $1, %eax  /* PE: activa modo protegido */
        bts $31, %eax /* PG: habilita paginación */
        movl %eax, %cr0

        /* Salta a un segmento de 64 bits. */
        ljmp $K_CS, $start64


        .code64

start64:
        /* Este código ya es de 64 bits */
        movw $K_SS, %ax
        movw %ax, %ss
        movabsq $(stack + STACKSIZE), %rsp

        /* movabsq fuerza que el operando sea de 8 bytes */
        movabsq $kmain, %rax
        callq *%rax

1:
        jmp 1b



        .data

        .align 8

gdt_ptr:
        .word 48 - 1
        .long gdt - K_PAGE_OFFSET

        .bss

        .align 4096
```

```
.lcomm pml4, 0x1000
.lcomm pml3, 0x1000
.lcomm pml2, 0x1000
.lcomm pml2d, 0x1000
```

3.6. kmain

El archivo boot.S en un momento dado salta a un símbolo: "`kstart`". Este símbolo, junto con otros elementos que usa boot.S, los definirá un archivo en código C. El archivo es este:

Listado 3.9: main.c

```
__attribute__((aligned (8))) char stack[STACKSIZE];


unsigned long int gdt[4] __attribute__ ((aligned (16))) = {
        0x0000000000000000,
        0x00af98000000ffff, /* k_cs */
        0x008f92000000ffff, /* k_ds */
        0x008f92000000ffff, /* k_ss */
};


#define VGA_BASE        (void *)0xb8000

int
kmain (void)
{
        const char msg[] = "Arrancando el kernel...";
        unsigned int i = 0;

        do {
                *((unsigned short int *)VGA_BASE + i)
                        = (0xf << 8) | *(msg + i);
        } while (++i < sizeof (msg) - 1);

        do {
                *((unsigned short int *)VGA_BASE + i)
                        = (0xf << 8) | ' ';
        } while (++i < 80*25);

        return 0;
}
```

La primera linea crea la pila del kernel. La estructura que va después crea los segmentos y la función es un código sencillo que imprime una cadena usando el dispositivo de video VGA.

El código se compone de dos bucles. Ambos escriben dos caracteres en 16 bits de memoria en cada vuelta. El menos significativo será el carácter que queremos imprimir y el más significativo será el código de color (blanco sobre negro en este caso, pero pueden especificarse otros colores, cada uno en un valor de 4 bits).

3.7. Makefile

Hasta ahora tenemos el código ensamblador que arranca ¡OS (boot.S), el script del linker que dice como han de juntarse las piezas (linker.ld) y una función main en C que imprime una cadena por pantalla. Estos archivos componen un "hola mundo" de un kernel completo, y si comparamos con el trabajo que costaría imprimir una cadena con un programa en C normal, el esfuerzo es considerable. Estos conocimientos sirven para darnos cuenta de todo lo que se necesita en un sistema operativo para hacer algo tan simple como imprimir un carácter en pantalla así como de todas las tecnologías que implicadas ya que nuestro ejemplo es una implementación relativamente simple.

En esta sección vamos a ver los pasos necesarios para juntar las piezas.

Un "Makefile" es un archivo que se interpreta al cargar el comando "make" en el directorio que lo contenga. La diferencia entre usar un simple script y usar make, es que este último comprueba las fechas de los archivos de código y objeto antes de ejecutar un comando. Si el objeto es más nuevo que el código, no necesita ejecutar ese comando puesto que el objeto está actualizado.

Si por ejemplo a partir del archivo de código fuente prueba.c se generase el archivo con el código objeto prueba.o, cuando fuésemos a reconstruir el software, make comprobaría las fechas de ambos archivos y sólo reconstruiría prueba.o si prueba.c fuera más moderno.

Cuando tenemos pocos archivos de código no hay mucha diferencia pero cuando un software crece, make nos puede ahorrar mucho tiempo.

El archivo Makefile usado por ¡OS es simple ya que no comprueba si hay cambios en archivos de cabecera .h para reconstruir los .c que los incluyan, pero los .h no se modifican tan habitualmente y el tamaño total del ¡OS no es tan grande. En la práctica el tiempo total de compilación en mi equipo es de unos pocos segundos (en un arranque en frío sería mayor pero eso sólo ocurre la primera vez).

Este es el contenido del archivo Makefile:

Listado 3.10: Makefile

```
K_PAGE_OFFSET = 0xffff800000000000

SOURCES = boot.o main.o

CFLAGS = -ffreestanding -nostdlib -nostdinc -mno-abm -msoft-float \
         -Wall -std=gnu99 -m64 -O2 \
         -DK_PAGE_OFFSET=$(K_PAGE_OFFSET) -DSTACKSIZE=8192
LDFLAGS = -N -dT linker.ld --defsym page_offset=$(K_PAGE_OFFSET) \
          --oformat=elf64-x86-64

ASFLAGS = $(CFLAGS)


all: $(SOURCES)
        ld $(LDFLAGS) -o jOS $(SOURCES)

clean:
        rm -f *.o jOS */*.o */*/*.o
```

La constante: `K_PAGE_OFFSET` representa el desplazamiento entre direcciones físicas y lógicas. La dirección física `0x12345` siempre será accesible desde la lógica `0xffff800000012345`.

Después indicamos los archivos objeto que queremos que nos genere, y los flags que ha de pasar a los distintos programas para generarlos.

Los flags de C son necesarios para que el compilador no enlace el binario final con la biblioteca de C ni con ninguna otra, para que no incluya sus archivos de cabecera, para que de avisos sobre posibles errores en el código (`-Wall`), para que nos permita usar extensiones GNU (`-std=gnu99`), para generar código de 64 bits y para especificar el nivel de optimización del código[7]. Por último se definen dos macros (con `-D`), la primera para que el código pueda acceder a K_PAGE_OFFSET y la segunda define el tamaño reservado para el stack del kernel.

Los flags del linker (LDFLAGS), indican que script usar para generar el binario final y otras opciones sobre la forma de generar el binario.

ASFLAGS son las opciones que se pasan al ensamblador y se han dejado las mismas que para el código C.

[7]Un análisis de esta característica parece excesivo, pero si comparamos las diferencias entre un trozo de código optimizado y otro no optimizado, los resultados sorprenden. Además en el desarrollo del sistema operativo ha habido veces que el código funcionaba con optimizaciones y no lo hacía sin ellas o al revés.

Después incluye dos reglas, la primera para generar el binario del kernel y la segunda para limpiar los archivos generados como fruto de esa compilación.

3.8. Instalación y carga del binario

Los emuladores cargan los sistemas operativos emulados desde imágenes (archivos al fin y al cabo) y aunque también se pueden cargar desde particiones, no es un uso común. Por tanto tendremos que construir una imagen de un disco duro, crear las particiones, formatearla, instalar y configurar GRUB2 y copiar la imagen del sistema operativo.

Vamos a seguir este proceso.

Grub tiene comandos para generar imágenes: *grub-mkimage*, *grub-install*, *grub-setup* y otros. Linux también nos proporciona herramientas útiles para este proceso: *fdisk* permite crear particiones sobre archivos, *mount* con el flag *loop* nos permite montar particiones contenidas en archivos de disco y otras que ahora veremos.

Para usar estas herramientas y crear la imagen tendríamos que hacer algo así:

- Crear una imagen de disco duro de cualquier tamaño (con *dd* o cualquier otra herramienta similar).
- *fdisk* sobre la imagen para crear las particiones.
- Ejecutar *losetup* con *-o offset* para crear un dispositivo de bloque a partir de la primera partición. La forma de saber el offset correcto es examinando la imagen con *fdisk* y convirtiendo manualmente entre sectores y bytes. Este dato será distinto en función de la herramienta usada para crear las particiones (algunas herramientas alinean el comienzo de las particiones a múltiplos del tamaño de palabra, de página o incluso de megabyte).
- Usar *grub-install*, copiar los archivos adecuados, probablemente falle por la configuración de los dispositivos, algo que podemos arreglar editando el archivo *device.map* y volver a ejecutar el comando o usar *grub-setup*...

Este proceso funcionó con GRUB pero con GRUB2 actualmente no funciona. Es mucho más sencillo crear una imagen, cargarla desde otra máquina virtual con otro sistema operativo y particionarla e instalarle GRUB desde ahí.

Lo primero es crear la imagen, con unos pocos megabytes nos es suficiente:

```
# Podemos especificar el tamaño deseado en count:
dd if=/dev/zero of=hd.img bs=1024 count=65535
```

Después añadimos la imagen a una máquina virtual, en las opciones de arranque por ejemplo de Qemu, habrá que añadir:

```
-hdb /ruta/hd.img
```

Ahora el proceso es más sencillo que de cualquier otra forma. Desde el sistema operativo arrancado:

```
fdisk /dev/sdb
# creamos la parción con 'n', 'p', '1' y dos veces intro.
# la activamos con 'a', '1'
# escribimos los cambios con 'w'
# Comprobamos:
fdisk -l /dev/sdb
...
   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *        2048      131069       64511   83  Linux
# Creamos el sistema de archivos que soportará nuestro kernel:
mkfs.ext2 /dev/sdb1
# Montamos la partición:
mount /dev/sdb1 /mnt/
# Creamos un sistema que se parezca al nuestro:
cd /mnt
for a in bin dev etc lib lib64 mnt proc sbin sys usr var ;\
do mkdir $a ; mount -o bind /$a $a ; done
# Hacemos como si hubiéramos arrancado desde ahí:
chroot /mnt/
# Instalamos GRUB2:
grub-install /dev/sdb
```

Ahora en "*/boot/grub/grub.cfg*" ponemos la entrada con la configuración para el jOS:

```
set timeout=5

menuentry "jOS - (J)ose's (O)perating (S)ystem" {
```

```
set root=(hd0,1)
multiboot /jOS
}
```

Y podemos apagar la máquina virtual.

Desde el sistema operativo copiamos la imagen generada al compilar el ¡OS, para ello:

```
fdik -l imagen.img
# Vemos donde empieza la partición, en la columna Start
# viene número de sector que hemos de multiplicar por 512.
# En mi caso: 2048 * 512 = 1048576 (¡fdisk dejó el primer
# MB vacío!). Con este número:
mount hd.img /mnt/ -o loop,offset=1048576
# Copiamos la imagen:
cp jOS /mnt/
# Desmontamos:
umount /mnt/
```

Y cargamos cualquier emulador con una linea como esta:

```
qemu-system-x86_64 -drive if=ide,index=0,media=disk,file=hd.img -m 512
```

Si todo ha ido bien el sistema operativo debería mostrar por pantalla un mensaje de que se ha cargado correctamente.

A partir de ahora automatizamos la copia de la nueva imagen para que se haga desde el comando “make” antes de ejecutar el emulador y ya no nos volveremos a preocupar durante todo el desarrollo de copiar el binario a la imagen.

3.9. Conclusión

Con estas piezas tenemos una base sobre la que empezar a programar el sistema operativo. Además, aunque trabajar a tan bajo nivel es tedioso porque existen muchas sutilezas y porque hay que tener un conocimiento profundo de muchos dispositivos y herramientas, es más sencillo que los conceptos más abstractos que se manejan en las capas superiores.

Capítulo 4

Piezas básicas

4.1. Primeras abstracciones

Antes de tan siquiera entrar en el diseño de código, conviene pensar un poco que cosas serán necesarias en las primeras fases del desarrollo.

A continuación comento algunas lecciones básicas y prácticas aprendidas en el proceso y que creo resultarán de utilidad para alguien que quiera desarrollar un proyecto similar, especialmente para quien empiece desde cero.

- Usar punteros a void para las direcciones y no enteros del tamaño de palabra. En el código de nivel más bajo habrá muchas referencias a memoria física, puede ser tentador usar un entero puesto que en realidad el dato que buscamos es una dirección de memoria que es igual a un número, pero un puntero a void está precisamente pensado para esa función y como muchas veces no sabemos de antemano el ancho del dato al que queremos acceder, tendremos que hacer conversiones (*casts*) que pueden esconder errores.

- Usar datos físicos para cosas físicas y datos lógicos para cosas lógicas, además definirlos con typedefs. Por datos físicos quiero decir: "`u32`", "`u64`", etcétera y por cosas lógicas: "`size_t`", "`dev_t`" o similares. Evitar los tipos de C porque cambian con la arquitectura (o incluso con el compilador), también hay que tener cuidado con los enteros con signo ya que aunque no se usan mucho, pueden desbordar si asignamos un dato sin signo. También conviene añadir macros para NULL, tamaño de palabra y atributos especiales del compilador (cosas como "*packed*", "*aligned*" y similar). Es poco probable que algún día haya que cambiar

de compilador pero si hubiera que hacerlo o la sintaxis cambiara en una versión concreta, el código sería mucho más mantenible.

- Preparar biblioteca estándar de funciones básicas: *strcpy*, *strcmp*, *isdigit*, *islower*, *va_list*, *snprintf* y similares (*printf* no es posible hasta que no haya un dispositivo donde imprimir, pero podemos pensar que interfaz usaría por debajo e implementarlo con la biblioteca de C para empezar), *div*, *atoi*, funciones de tiempo... Pensar también funciones para cosas auxiliares: árboles, listas, manejo de bits, funciones para la CPU (*cpuid*, *msr*, etcétera), operaciones con números enteros (*bcd2int*, potencias, alineamiento de memoria...).

  ¡OS coloca las funciones estándar bajo "`include/`" en los mismos archivos en que se encuentran las definiciones de la biblioteca estándar de C y el resto bajo "`/lib`". Además, muchas de estas funciones están disponibles por Internet como código "*copyleft*", que quiere decir que podemos reusarlo sin restricciones.

  Lo que no tengamos, podemos desarrollarlo como programas de usuario y una vez que estén probados y funcionen, añadirlos al código del sistema operativo. Mediante este procedimiento ganaremos mucho en estabilidad ya que el código de usuario es más fácil de probar.

  Estos puntos fueron descubiertos demasiado tarde por lo que todo el código del ¡OS es propio (aunque el API de los árboles y las listas se inspira en el kernel de Linux, pero sólo el API puesto que Linux usa árboles balanceados y ¡OS árboles normales. Nótese que las estructuras con almacenamiento interno o "*internal storage*" no se implementaron en el kernel de Linux por vez primera a pesar de que fue el proyecto que las lanzó a la fama.

- Este punto hace referencia al anterior: si desarrollamos cosas en espacio de usuario y las integramos con el kernel cuando sean estables, mediante un pequeño esfuerzo más, podemos añadir tests para esas funciones. Los tests comparan la salida real de un código con la salida esperada y si cambiamos algo que altere la salida real, se detectará ya que el test no se pasaría.

  Podemos ver como se usa esta técnica mirando el directorio `test/`, el cual se parece a un directorio de tests de Perl, tal como si usáramos el módulo Test::More[1].

- Una de las primeras cosas que tendremos que hacer, será un driver de video, aunque sea incompleto, para escribir en modo texto. Al menos ha de proveer la función "escribir carácter". Se necesita con urgencia puesto que será muy útil imprimir por pantalla para poder depurar. Otros sistemas operativos usan otras

[1] `http://search.cpan.org/perldoc?Test::More`

técnicas más avanzadas como un puerto de serie, un depurador integrado o incluso las luces del teclado para emitir los errores en código Morse, pero escribir una cadena de caracteres por pantalla es sencillo de hacer y suficientemente útil como para que sea de lo primero. Si después necesitamos un driver mejor podemos modificarlo sin problemas debido a que el API no cambia.

No obstante, imprimir no siempre sirve de ayuda (sobre todo en el principio del desarrollo) puesto que a veces el equipo se reinicia inmediatamente o hay algún problema antes de poder imprimir en pantalla.

- La primera cosa de mayor nivel que hemos de diseñar y programar, será la memoria dinámica. Si no tenemos memoria dinámica, hemos de saber en tiempo de compilación cuantos elementos de cada cosa vamos a tener. Si lo hacemos así terminaremos por meter números fijos en el código donde en realidad debería ir una estructura dinámica. Además el primer problema sin solución fácil será poner en funcionamiento un sistema de memoria dinámica sin tener memoria dinámica.

Ninguno de estos consejos se siguió en el desarrollo del ¡OS y aunque se intentó corregir, supuso mayor esfuerzo que si se hubieran seguido desde el principio.

4.2. Video y stdio

Sobre la biblioteca de funciones habituales que usemos no vamos a decir nada puesto que son funciones de ayuda para otras cosas y similares a las que se pueden encontrar en una biblioteca estándar de C.

Para hacernos una idea del esfuerzo de esta parte, estas funciones representan un 20 % del código del kernel (aunque el número cambiará probablemente en el futuro).

Por tanto vamos a empezar por el siguiente elemento de mayor prioridad: tener texto en la pantalla.

El sistema gráfico del los PCs es muy confuso. Tenemos dispositivos: CGA, EGA, VGA y otros más modernos; ¡OS usará el modo texto "*VGA compatible*". Podemos encontrar una pequeña introducción en este artículo de Wikipedia: `http://en.wikipedia.org/wiki/VGA_compatible_text_mode` y un poco más de información en `http://wiki.osdev.org/VGA_Hardware`.

La memoria de video se mapea entre las direcciones `0xb8000` y `0xbffff`, y si hacemos cuentas, entre esas direcciones hay 32KB de memoria. Por otro lado en modo básico este adaptador tiene 80 columnas y 25 filas, es decir, 2000 cuadros de texto o

caracteres, cada uno representado con 2 bytes, uno para los colores de texto y de fondo (4 bits, o 16 colores para cada uno) y otro para el carácter que queremos imprimir (256 posibilidades). Si hacemos cuentas, 2000 celdas con 2 bytes cada una suman 4KB, por lo que los requisitos mínimos para hacer funcionar nuestro sistema operativo, son pequeños: adaptador con 16 colores, 4KB de memoria, y con una resolución en modo texto de 80x25. Cualquier ordenador personal con menos de 30 años de antigüedad cumple estos requisitos.

Una duda que surge al desarrollarlo es ¿qué interfaz de programación proporcionará nuestro driver? ¡OS toma una solución sencilla: proporciona una función para escribir un carácter. Por tanto las funciones *printf* o *puts* harán una llamada a esa función para cada carácter que quieran imprimir. Además se proporciona otra función para limpiar la pantalla.

Imprimir un carácter no es tan fácil como parece. ¿en qué parte de la memoria de video lo colocamos? Podemos poner el primero en la primera y cada uno que vaya llegando lo colocamos detrás pero antes o después llegaremos al final; en ese caso habrá que descartar la primera linea (luego tenemos que tener un control de lineas) y desplazar el resto una posición hacia arriba. Otra complicación que se presenta serán los caracteres no imprimibles; si el usuario pulsara un "*intro*", habría que interpretar el carácter de forma que el siguiente irá al principio de la siguiente linea.

Como la memoria es un array unidimensional, cada fila empezará en múltiplos del número de columnas y mover una linea hacia arriba significa copiar cada posición de memoria a partir de la segunda fila, tantas direcciones atrás como columnas tenga una fila, rellenar de espacios la última fila y mover el cursor a la primera posición de la última fila. De forma similar, pasar a la siguiente fila significa avanzar el cursor hasta el siguiente múltiplo del número de columnas y tal vez hacer *scroll* (es decir, mover cada linea a la anterior).

No hay mucho más que comentar ya que la implementación es muy sencilla. Tan sólo una pequeña excepción: el cursor. Se trata de un pequeño subrayado que ayuda a recordar en que posición se escribirá el siguiente caracter. Para moverlo, el dispositivo nos proporciona unos puertos en el espacio de IO (vimos por encima que era el espacio IO en la sección 1.4).

Dado que el espacio IO es útil para varias cosas, pasamos a ver la función que mueve el cursor y así introducimos el método usado para los accesos a este espacio.

Listado 4.1: Colocando el cursor en la pantalla.

```
static inline void
outb (u16 port, u8 val)
```

```
{
        asm volatile ("outb %1, %0" : : "dN" (port), "a" (val));
}

static inline void
move_cursor (void)
{
        u16 loc = line * 80 + col;

        outb (0x3D4, 14); /* byte alto */
        outb (0x3D5, loc >> 8);

        outb (0x3D4, 15); /* byte bajo */
        outb (0x3D5, loc);
}
```

La función "*outb*" escribe por el puerto de 16 bits indicado un dato de 8 bits. Tendremos también "*inb*" para leer un byte de un puerto y también "*inw*" y "*outw*" para trabajar con palabras (16 bits en este contexto).

Sobre la función que mueve el cursor, el driver guarda en variables globales las posiciones de la fila y de la columna. Esto es así porque pantallas sólo hay una. Si pensamos por ejemplo en las terminales de UNIX, podría haber varias y tendríamos que usar memoria dinámica o saber de antemano (al compilar) el número de terminales.

Esta función mueve por tanto el cursor a la posición actual en donde estemos apuntando con nuestro driver.

4.3. ACPI, PIC y APIC

Aunque la memoria virtual es importante, podemos tratar en este capítulo las interrupciones ya que son de más bajo nivel que el manejo de la memoria.

Cualquier usuario con la suficiente experiencia recordará como antiguamente al terminar de usar un ordenador teníamos que pulsar un botón para quitarle la corriente eléctrica. En 1992 Intel y Microsoft desarrollaron la tecnología APM ("*advanced power management*") y los PCs ya podían desconectarse solos, bajar de frecuencia el procesador para ahorrar energía (como el "turbo" de procesadores antiguos, salvo que ese lo hacíamos nosotros pulsando un botón), y suspenderse en memoria o disco ("*suspend*" o "*standby*", cada fabricante usa términos distintos).

APM se quedó pequeño y con la intención de reemplazarlo en 1996 varias compañías publicaron el primer estándar de ACPI (además este estándar integró las especi-

ficaciones multiprocesador y PnP o "*Plug and Play*" BIOS). La principal novedad de cara al usuario de este nuevo estándar fue la capacidad de un sistema de suspenderse y de hibernar apropiadamente (APM era limitado), así como de variar la frecuencia del procesador de forma dinámica ahorrando energía y por tanto dinero.

Esta parte de la historia de los PCs se comenta para completar la explicación ya que es una especificación importante de cara a programar un sistema operativo, pero lo que realmente nos interesa es que es imprescindible para implementar correctamente las interrupciones.

La especificación ACPI usada en este trabajo ha sido [ea09] y ocupa 727 páginas[2], pero aunque a veces sea inevitable, hay que intentar no abrumarse ya que sólo necesitamos una pequeña funcionalidad.

Al arrancar el sistema operativo, una BIOS que soporte ACPI nos deja en memoria unas estructuras con información del sistema. La especificación nos da unos rangos de RAM donde podemos buscar unas tablas, y unos números mágicos[3]. Esas tablas tienen checksum por lo que es prácticamente imposible que pensemos que es una tabla cuando no lo es.

Como la especificación es grande, incluye mucha información sobre el sistema. De entre todas las que nos podríamos encontrar destacamos la tabla MADT; mediante ella podremos detectar los APIC y el IOAPIC, además podremos ver si existen "*overrides*" (podemos traducirlo como "invalidaciones") de las interrupciones en el sistema. Esto quiere decir que ya no usaremos el clásico PIC (del inglés: "*programmable interrupt controller*", modelo "Intel 8259") sino la más moderna arquitectura APIC y que algunos de los dispositivos que tradicionalmente han usado un número fijo de interrupción, pueden estar en otro (de aquí la necesidad de usar ACPI).

Este diseño supone un cambio de arquitectura total de cara al manejo de interrupciones. Cada CPU tiene conectado un LAPIC (de: "*local APIC*") y todos ellos se comunican mediante un bus privado (algunos modelos usan el bus del sistema pero puede verse como privado a efectos prácticos). En dicho bus puede existir un PIC para mantener compatibilidad (o un circuito que lo emule) y se añaden otras características que

[2]Este problema de necesitar demasiada información ocurrirá siempre. Los manuales de AMD suman más de 2500 páginas, los de Intel, incluyendo el de optimización, casi 5 mil, el de los discos ATA versión 7, casi 1000 (otros 300 si queremos serial ata y esto no incluye DMA), 350 de la especificación PCI, 100 de ELF y aún nos quedarían los manuales de los programas (ensamblador, compilador, emulador, depurador, etcétera), los de los dispositivos (PIT, APIC, IOAPIC, RTC, AIP, HPET...), manuales técnicos de tecnologías (Ext2, Multiboot, linkers, hardware...), libros sobre sistemas operativos en general o sobre algún kernel en particular (y de sus subsistemas ya que algunos tienen libros dedicados). Además si quisiéramos hardware moderno la cosa se escapa de las posibilidades de un sólo programador o incluso de un grupo pequeño (USB, drivers y pilas de red, sonido, almacenamiento, etcétera).

[3]En el capítulo anterior explicamos que un número mágico es un número cualquiera, por ejemplo ACPI usa `0x2052545020445352` para detectar las tablas.

podemos ver en [AMDb] y en [cork].

Como el PIC 8259 se necesita para mantener la compatibilidad, no podemos obviar su funcionamiento. En la informática a veces es difícil entender una situación actual sin remontarse a la situación histórica que la generó.

Dado que es un dispositivo antiguo el manual (ver [cord]) contiene toda la especificación incluyendo las patillas de las que dispone el dispositivo. Tanto nivel de detalle no nos será necesario puesto que de cara a programarlo nos es más útil la configuración de registros y puertos de entrada y salida, algo que podemos encontrar resumido en este otro recurso: `http://wiki.osdev.org/PIC`.

Originalmente el PIC 8259 se conectaba a la patilla de interrupciones del procesador y a él se le conectaban los distintos periféricos. Si alguno necesitaba atención avisaba al PIC y este a su vez al procesador. De esta forma las interrupciones se priorizan ya que si llegan dos simultáneamente, el PIC puede elegir sobre cual avisar al procesador; esta prioridad se determina con el número de patilla del PIC, un dispositivo conectado a la patilla 0 tendrá mayor prioridad que otro conectado a la patilla 1 y así sucesivamente hasta la patilla 7.

Con el tiempo las 8 patillas disponibles fueron insuficientes. La solución que se tomó fue conectar otro "en cascada" a la interrupción número dos de éste. En este caso, el segundo PIC avisaría al primero cuando uno de sus dispositivos necesitara atención y dado que este PIC en cascada se conectó a la patilla dos, la prioridad de cualquiera de sus dispositivos conectados es menor que la de los dispositivos conectados a las patillas 0 y 1 y mayor que la de los conectados a las patillas 3-7 del primer PIC.

Las interrupciones se estandarizaron y se numeraron de 0 a 15. La 0 se conectó al temporizador del sistema, la 1 al teclado, las 3 y 4 a los puertos de serie, la 5 a la controladora de disco en el PC XT o al puerto paralelo en el PC AT, la 6 a la disquetera, la 7 al otro puerto paralelo, la 8 al reloj del sistema, la 12 al ratón, la 13 al coprocesador (si había) y las 14 y 15 a los discos duros.

Para conectar el procesador a los distintos dispositivos se usaba un circuito impreso que se conoció como placa base (del inglés: "*motherboard*"). En ese circuito el fabricante del PC instalaba unos dispositivos mínimos que conectaba al PIC. La forma de que un sistema operativo pudiera saber de que dispositivo se trataba, era mediante el uso de números de interrupción estándar (podemos recordar como en MS-DOS había que especificar de forma manual los números de interrupción de las tarjetas de sonido así como de modems y otros dispositivos no estándar).

Con el tiempo las placas incluyeron buses de expansión por lo que los números de interrupción no podían ser estandarizados a priori. Habría sido posible reservar una

para cada tipo de dispositivo pero si descontamos los dispositivos estándar quedaban muy pocas patillas libres.

Cuando se conectaba un dispositivo a un bus de expansión había que modificar su configuración (mediante un puente en el circuito) para asignar un número de interrupción libre al dispositivo, y desde el sistema operativo teníamos que indicar el número de interrupción como hemos dicho. Algo tedioso para el usuario, especialmente para la persona que tuviera que dar soporte a un número grande de equipos o para quien no estuviera muy versado pero necesitara usar un ordenador.

La primera solución fue compartir las patillas libres. Esto causó problemas ya que algunos dispositivos no estaban preparados para ello [4].

El nuevo APIC proporciona 256 interrupciones por lo que ya no hay que compartir. La única peculiaridad es que como los 32 primeros vectores se reservan para las interrupciones del procesador, es necesario mapear las 16 interrupciones del PIC en vectores libres. Además de mapearlas podemos configurar si la interrupción se detecta por flanco o por nivel y esta información la sacaremos precisamente de las tablas que un equipo que implemente la especificación ACPI deja en memoria.

Por ejemplo en la versión de mi emulador Qemu la interrupción 0 se ha movido a la 2 (se deduce por tanto que no hay PIC en cascada en la patilla 2), la tabla también incluye una redirección de la 5 a la 5 (algo que también ocurre con las interrupciones 9, 10 y 11). Estas redirecciones no modifican el número de interrupción pero sí la forma en que se lanza. Por defecto el 8259 genera interrupciones que se activan por flanco alto y en mi caso las interrupciones se han modificado para activarse por nivel alto. Según la arquitectura APIC cuando programamos los vectores hemos de indicar de que tipo queremos la interrupción, este tipo tendrá que coincidir con el tipo que necesita cada dispositivo.

Sin embargo la versión del emulador Bochs que uso no tiene redirecciones. Es necesario por tanto leer las tablas ACPI si queremos usar APIC, si no queremos pasar por el trabajo de interpretarlas podemos usar PIC pero el sistema operativo no tendrá administración avanzada de energía y sólo soportará una CPU ya que las tablas ACPI también contienen información sobre el número de CPUs que tiene el sistema (como se dijo, la especificación ACPI incorporó la especificación multiprocesador).

La diferencia entre nivel y flanco, es que las que se activan por nivel se reconocen cuando la señal se encuentra en un nivel determinado (alto o bajo), las que se activan

[4] En función de si las interrupciones son por flanco o por nivel puede ser más o menos fácil compartir interrupciones. Por ejemplo en la arquitectura ISA los dispositivos solicitaban las interrupciones mediante activaciones por nivel y por tanto éstas no se podían compartir, aunque después se incluyeron resistencias *pull-up* pero en todo caso no era la mejor solución.

por flanco lo que se reconoce es un cambio en el estado (de alto a bajo o de bajo a alto).

Además de ACPI, otro estándar relacionado es SMBIOS (ver [DMTF08]). Contiene información sobre el equipo y se menciona para completar. La forma de colocar las tablas en memoria es similar a la usada por ACPI [5].

Para terminar y como en esta sección no hay mucho código fuente, vamos a poner un extracto del código usado para inicializar el LAPIC y el IOAPIC con el fin de hacernos una idea general de como se programan estos dispositivos. Como siempre, el proceso completo se encuentra en el código.

Listado 4.2: Código lapic

```
static inline void
lapic_write (u32 reg, u32 val)
{
        /* La localización del LAPIC se lee de las tablas ACPI. */
        volatile u32 *addr = (volatile u32 *)(lapic[0].base + reg);

        *addr = val;
}

static u32
ioapic_read (u8 reg)
{
        *(volatile u32 *)(ioapic[0].base + IOREGSEL) = reg;

        return *(volatile u32 *)(ioapic[0].base + IOWIN);
}

static inline void
lapic_eoi (void)
{
        lapic_write (APIC_EOI, 0);
}

...

/* Lee la versión del APIC disponible */
lapic[0].version = (u8)(lapic_read (APIC_VERSION) & 0xff);
/* TPR a 0 desbloquea las interrupciones */
lapic_write (APIC_TPR, 0);
/* Instala un manejador que se llamará en caso de error */
intr_install_handler (APIC_E_INTR, do_lapic_err);

/* Lee la versión del IOAPIC */
```

[5] En el desarrollo de ¡OS se programó una especificación parcial de SMBIOS de unas 100 lineas de código y nunca se aprovechó ya que se decidió usar las más completas tablas ACPI.

```
ioapic[0].version = (u8)(ioapic_read (IOAPICVER) & 0xff);
/* Envía el fin de interrupción */
lapic_eoi ();
```

Se muestra una función para escribir en un registro del LAPIC, y como vemos, es muy sencillo puesto que los registros se representan con un desplazamiento respecto a una dirección de memoria base que leemos de las tablas ACPI.

A continuación se muestra otra función para leer del IOAPIC que funciona de forma parecida pero no igual porque el registro del que queremos leer se pasa escribiendo en un registro concreto y el dato se devuelve en otro registro.

La función `lapic_eoi` es necesaria al final de cada manejador de interrupción, así como después de inicializar el IOAPIC (la parte "*eoi*" significa "*end of interrupt*" por tanto podemos suponer cual es su función).

El código que sigue es una parte pequeña de la secuencia de inicialización. No es importante conocer los pasos concretos (aunque por cosas como pequeñas sutilezas en el orden no fue trivial de programar), por lo que nada más se indica la función de cada linea en los comentarios del código.

4.4. Interrupciones y excepciones

Como dijimos, las interrupciones del procesador en la arquitectura APIC tienen números fijos así que nos quedan 224 vectores de interrupción disponibles para conectar dispositivos. Además APIC soporta IPIs del inglés: "*inter-processor interrupt*") pero ¡OS no hace uso de ellas. En esta arquitectura un I/O APIC es como un *router* que envía cada interrupción al procesador que se le haya programado.

En esta sección vamos a entrar en más detalles de implementación aunque superficialmente; para tener todos los detalles podemos mirar en el manual o en el código.

De las primeras cosas que ¡OS hace en el código es detectar las tablas ACPI (archivo *kernel/acpi.c*). Se necesita hacer pronto ya que al estar colocadas en RAM podríamos sobreescribirlas con cualquier otro dato, además necesitaremos su información para lo que vayamos construyendo. Nótese que como se hace antes de la memoria dinámica, habremos de ser ingeniosos con el código anterior a su detección para no desperdiciar demasiada memoria con reservas estáticas.

El siguiente paso es inicializar las interrupciones. Para ello definimos una tabla tal como la que necesita la CPU (como comentamos, x86 usa una tabla que se llama "*interrupt descriptor table*") y desde código en C rellenamos cada elemento con un

descriptor de puerta de interrupción o de "*trap*" ("*interrupt gate*" o "*trap gate*" según se trate de una interrupción o un trap respectivamente, la diferencia es que unas ocurren en cualquier momento y las otras después y como consecuencia de ejecutar una instrucción). Esta es la estructura:

Listado 4.3: Estructura para los traps de CPU.

```
static struct {
        u16 offset1;
        u16 selector;
        u8 ist;
        u8 flags;
        u16 offset2;
        u32 offset3;
        u32 reserved;
} __attribute__ ((__packed__, aligned (8))) idtentry[256];
```

Los elementos "*offset*" indican (por partes) el "*entry point*" del procedimiento que manejará la interrupción, "*selector*" es el segmento en el que se encuentra esa dirección (segmento de código del kernel en este caso), "*ist*" es para cambiar de pila para las interrupciones (si se quisiera un *call stack* separado para ellas), y los flags principalmente se usan para gestión de privilegios.

Como queremos que las interrupciones se manejen en código C, es común poner código en ensamblador que despache hacia el código C. Mediante un mecanismo ingenioso ¡OS se salta esta necesidad y la dirección que se instala es la de las funciones en C (casi).

Vamos a ver este mecanismo. Cuando se instala un manejador de interrupción no se instala la dirección de la función sino una dirección 16 bytes mayor. La razón es que GCC instala un prólogo para las funciones y dado que las interrupciones cambian el estado de la pila no lo necesitamos[6].

Dado que añadimos una dirección 16 bytes mayor, tenemos que asegurarnos de que nuestro código del manejador se encuentre después de esa dirección y también de que el prólogo no sobrepase los 16 bytes. Vamos a ver un prototipo de manejador:

Listado 4.4: Prototipo de manejador de interrupción.

```
__isr__
manejador (struct intr_frame r)
{
        intr_enter ();
```

[6]GCC no incluye un atributo "*naked*" que permita omitir el código, existe un "bug" reportado para añadir esta característica (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=25967`) y aunque se pensó en escribir el código GCC que lo resolviera se tomó el camino más pragmático de saltarlo

```
        ...

        intr_exit ();
}
```

Lo primero que llama la atención es lo de `__isr__`. Sólo es una macro que se define así:

Listado 4.5: Macro para valor devuelto desde isr

```
#define __isr__          \
        __attribute__ ((aligned (16))) static void
```

De esta forma nos aseguramos que la función es estática para que nadie pueda llamarla desde ninguna parte fuera del archivo y por tanto que sólo se pueda llamar mediante su dirección, que no devuelve ningún valor y que está alineada a 16 bytes.

La llamada a la función "`intr_enter`" ha de hacerse antes incluso de la definición de las variables (algo permitido en los estándares modernos de C). Es posible que en lugar de `intr_enter` haya que usar: `intr_err_enter()` ya que para algunas excepciones el procesador pasa un dato por la pila (por ejemplo cuando hay un fallo de página el procesador pasa la dirección de memoria que provocó el fallo). Esta es la definición:

Listado 4.6: Definición de intr_enter.

```
#define pushaq()                \
        "pushq %rax\n\t"        \
        "pushq %rcx\n\t"        \
        "pushq %rdx\n\t"        \
        "pushq %rbx\n\t"        \
        "pushq %rsi\n\t"        \
        "pushq %rdi\n\t"        \
        "pushq %r8\n\t"         \
        "pushq %r9\n\t"         \
        "pushq %r10\n\t"        \
        "pushq %r11\n\t"        \


#define intr_enter()                    \
        asm volatile (                  \
                "nop; nop; nop; nop;"   \
                "nop; nop; nop; nop;"   \
                "nop; nop; nop; nop;"   \
                "nop; nop; nop; nop;"   \
                "pushq $0\n\t"          \
                pushaq()                \
```

```
                "subq $16, %rsp\n\t"    \
                "cld\n\t"               \
                )

#define intr_err_enter()                \
        asm volatile (                  \
                "nop; nop; nop; nop;"   \
                "nop; nop; nop; nop;"   \
                "nop; nop; nop; nop;"   \
                "nop; nop; nop; nop;"   \
                pushaq()                \
                "subq $16, %rsp\n\t"    \
                "cld\n\t"               \
                )
```

Poniendo 16 bytes con instrucciones `nop` nos aseguramos de que por pequeño que sea el prólogo de la función nunca saltaremos a una dirección posterior a la primera instrucción que necesitamos. Es posible que el prólogo fuera mayor de 16 bytes pero se ha comprobado empíricamente con varias versiones de GCC y el prólogo generado con optimizaciones ocupa exactamente 4 bytes; en la práctica este mecanismo funciona[7].

Como vemos, `intr_enter` sólo difiere de `intr_err_enter` en que añade un cero donde el procesador pondría su código en caso de haberlo. Esto es lo que hace la siguiente instrucción, y se trata de otro detalle ingenioso para hacernos la vida más fácil; dado que, como dijimos, el procesador mete un dato en algunas interrupciones, en las que no lo pone lo ponemos nosotros y así el "*stack frame*" de la pila en las interrupciones queda homogéneo haya o no código puesto por el procesador.

Después hay otra macro: `pushaq`. Esta macro guarda el valor de los registros que GCC podría modificar en el manejador. La idea es dejar la pila como estaba cuando terminemos nuestro manejador y esa es exactamente la tarea de `intr_err_exit`.

La siguiente linea es algo más compleja. Si observamos el prototipo de la función manejadora, vemos que se le pasa una estructura como parámetro. Este es el prototipo de esa estructura:

Listado 4.7: Definición de intr_enter.

```
struct intr_frame {
        u64 r11;
        u64 r10;
        u64 r9;
        u64 r8;
        u64 rdi;
```

[7]Este pragmatismo es necesario ya que si examinamos una instrucción cualquiera en los manuales del procesador, es imposible cubrir todas las posibilidades que podrían generarse como resultado de su ejecución.

```
        u64 rsi;
        u64 rbx;
        u64 rdx;
        u64 rcx;
        u64 rax;
        u32 ecode;
        u32 pad4;
        u64 rip;
        u8 cs;
        u8 pad0; u32 pad1;
        u64 rflags;
        u64 rsp;
        u8 ss;
        u8 pad2; u32 pad3;
} __attribute__ ((__packed__));
```

Tenemos los mismos campos que la macro `pushaq` puso en la pila pero colocados al revés, y justo después (antes en la pila ya que recordemos que crece hacia abajo), el código de error y otros registros.

Ya podemos ver claramente por que añadimos el código de error, de esta forma todos los manejadores pueden recibir la misma estructura como parámetro.

Y volviendo a la linea que resta 16 a la pila (es decir, suma al tope), se hace para que la estructura que se le pasa como parámetro pueda funcionar puesto que GCC no sabe que estamos modificando la pila y espera que los datos estén en una dirección concreta.

Ya solo queda comentar el procedimiento de vuelta. Este procedimiento deshace la resta de 16, restaura los registros a partir de los datos que tiene en la pila (en el orden inverso al usado en el *push*) y por ultimo ejecuta una instrucción `iretq`. Esta instrucción hace que nunca se ejecute el `retq` normal de regreso de la función sino el que regresa de una interrupción y deshace los cambios que la puerta (*gate*) de la interrupción hizo.

Una vez tenemos este mecanismo ya podemos registrar manejadores. ¡OS en en el inicio instala los manejadores para las excepciones de la CPU y el resto los rellena con uno genérico que no hace nada. Después cada driver podrá sobreescribirlos al registrar los suyos.

Los manejadores de dispositivos ademas tienen otra peculiaridad que ya comentamos; han de llamar cuando terminan a la función que envía el EOI ("*end of interrupt*" o fin de interrupción) al LAPIC.

Los siguientes pasos son conceptualmente simples y de bajo nivel: instalar la tabla

con los vectores, deshabilitar el PIC y detectar e inicializar el LAPIC y el IOAPIC (lo que implica redirigir las interrupciones que antes manejaba el PIC a los vectores 32 en adelante).

Lo único que nos falta sería el mecanismo para que los drivers puedan registrar sus interrupciones. Para ello hacemos una función que admita un número de interrupción y una dirección de memoria. Algo como lo siguiente:

Listado 4.8: Para instalar manejadores de interrupciones.

```
void
intr_install_handler (u8 num, void *addr)
{
        switch (num) {
        case 0 ... 31:
                idt_set_gate (ioapic.pic[num].dest + 32, addr, K_CS,
                    GATE_INT);
                ioapic_redir_unmask (num);
                break;
        case 32 ... 63:
                break;
        case 64 ... 255:
                idt_set_gate (num, addr, K_CS, GATE_INT);
                break;
        }

}
```

Las 32 primeras quedan reservadas para la CPU, de la 33 a la 64 se ocupan con las interrupciones del IO-APIC y el resto quedan libres para dispositivos.

Con esto tenemos un sistema de interrupciones relativamente completo sobre el que seguir construyendo.

El siguiente elemento debería ser la memoria dinámica, pero antes vamos a hablar superficialmente de teclado y de los relojes y temporizadores, ya que son dispositivos sencillos y mediante los cuales podemos completar la discusión del sistema de interrupciones.

4.5. Teclado

Aunque la mayoría de los teclados de hoy en día son USB, vamos a hablar del teclado del PC, el cual es más simple (sobre todo porque nos ahorra implementar el estándar USB): cuando se pulsa una tecla se lanza una interrupción estándar, se llama

al manejador, se lee un código de tecla (llamado "*scancode*") de un puerto y se saca por pantalla.

El driver es tan simple que sólo tiene que registrar su interrupción y proveer un manejador. El manejador se instala así:

Listado 4.9: Driver de teclado: instalando el manejador

```
intr_install_handler (1, do_keyboard);
```

Cuando haya una pulsación se llama a dicha función, la cual se define más o menos así:

Listado 4.10: Manejador de interrupción de teclado.

```
__isr__
do_keyboard (struct intr_frame r)
{
        intr_enter ();

        u8 scancode = inb (0x60);

        switch (scancode) {
        case 0x2a:
        case 0x36: /* Mayúsculas pulsadas: */
                keyb.shift = 1;
                break;
        case 0xaa: /* Mayúsculas soltadas: */
        case 0xb6:
                keyb.shift = 0;
                break;
        default:
                {
                char c;

                /* Si el bit alto está activo,
                 * la tecla se ha soltado */
                if (scancode & 0x80)
                        break;

                /* Si mayúsculas está pulsado,
                 * usa tabla de mayúsculas y sino de minúsculas. */
                if (keyb.shift)
                        c = kb_en_caps[scancode];
                else
                        c = kb_en[scancode];

                switch (c) {
```

```
                case '\b':
                        kbdbptr--;
                        kprintf ("%c", c);
                        break;
                default:
                        if (kbdbptr - kbdb < KBDB_LEN)
                                *kbdbptr++ = c;
                        kprintf ("%c", c);
                }

                }
        }

        lapic_eoi ();
        intr_exit ();
}
```

Las tablas de teclado son sencillas, solamente se definen dos tablas con código *ascii* para mayúsculas sin pulsar o pulsadas respectivamente:

Listado 4.11: Tablas de teclado para inglés

```
char kb_en[256] = "\0\0"
        "1234567890-=\b"
        "\tqwertyuiop[]\n"
        "\0asdfghjkl;'`"
        "\0\\zxcvbnm,./\0"
        "\0\0 \0\0\0\0\0"
        "\0\0\0\0\0\0\0\0"
        "\0\0\0-\0\0\0+"
        "\0\0\0\0\0\0\0<";
char kb_en_caps[256] = "\0\0"
        "!@#$%^&*()_+\b"
        "\tQWERTYUIOP{}\n"
        "\0ASDFGHJKL:\"~"
        "\0|ZXCVBNM<>?\0"
        "\0\0 \0\0\0\0\0"
        "\0\0\0\0\0\0\0\0"
        "\0\0\0-\0\0\0+"
        "\0\0\0\0\0\0\0>";
```

Un driver de un sistema operativo real se complicaría ya que tendría que comprobar a qué terminal le corresponde la pulsación, cuál es el proceso en primer plano de dicha terminal, copiar la pulsación a memoria, etcétera. Un recurso bueno para aprender sobre terminales es: [Ker10].

4.6. Temporizadores

El control del tiempo es importante para un kernel. Es necesario ejecutar trabajos periódicos como llamar al planificador de tareas para que reajuste las colas de procesos (prioridad, orden o similar), también necesita planificar eventos para el futuro ("*delayed IO*" o cualquier driver, tal como el de una unidad de disco que podría apagar el motor después de unos segundos sin usarse) y por último necesita saber la fecha y la hora (para actualizar fechas de archivos o pasarla al usuario).

Si examinamos la arquitectura x86 disponemos de varios tipos de temporizadores:

- PIT: "*programmable interval timer*". Compuesto por los modelos Intel 8253 y 8254, cuya especificación puede encontrarse en [core], es el temporizador clásico del PC. Dado que la especificación y el código de ¡OS detallan su programación y funcionamiento, no entraremos en detalles. A grandes rasgos, tiene un error diario de +/- 1.73 segundos, y dispone de 3 canales, uno conectado al IRQ0, otro servía para refrescar la RAM y el último para generar sonido por el PC speaker.

  El que nos interesa por tanto es el que se conecta a la interrupción 0. Aunque tiene varios modos de funcionamiento y puede generar pulsos de distintas formas, el uso que haremos será cargarlo con una cuenta y programarlo para que genere una interrupción cada cierto tiempo. Dado que la frecuencia de este dispositivo es de 18,2065 Hz, podremos tener una interrupción IRQ cada 54,9254 ms.

- Timer LAPIC: cada LAPIC tiene disponible un temporizador. La ventaja es que a diferencia del anterior, existe uno por cada CPU. Este temporizador es más complejo de usar porque su frecuencia deriva directamente de la velocidad del bus FSB ("*front side bus*") y por tanto hemos de calibrarlo, pero proporciona más precisión (en microsegundos).

- TSC: "*time stamp counter*". En realidad no estamos hablando de un dispositivo sino de una instrucción de la CPU: `rdtsc`, que permite leer el número de *ticks* de reloj que ha habido en una CPU desde que se inició el sistema. Por tanto, para un procesador cuya frecuencia ronde los gigahertzios (todas las CPUs que implementan esta instrucción alcanzan dicha frecuencia), la precisión es de nanosegundos. El problema que tiene es que dado que las CPUs modernas cambian de frecuencia es posible que no sea constante (algunos modelos sí que tienen TSC constante pero no podemos contar con ello), o que la CPU haya dormido o hibernado y que por tanto el resultado no sea fiable. Además, para que el TSC nos de un dato de tiempo, hemos de conocer la velocidad de la CPU y como no

hay ningún mecanismo estándar para ello, habremos de hacerlo empíricamente con algún otro temporizador.

Otro uso de esta instrucción es medir el rendimiento o uso de procesador que hace nuestro código. Si leemos el TSC, ejecutamos cierto código y volvemos a leer, sabremos cuantos ciclos de CPU hemos usado. En la práctica es necesario poner barreras de código porque puede haber ejecución fuera de orden ("*out of order execution*") e incoherencias entre las cachés de las CPUs y como no existe una dependencia entre la instrucción y el código, es posible que el procesador la ejecute antes o después de éste. Para solucionar este problema algunos procesadores tienen otra instrucción similar: "*rdtscp*".

- RTC: "*real-time clock*". Este dispositivo es ajeno al sistema y se mantiene encendido aún cuando apagamos el ordenador. Esta es la razón de que la fecha y hora se mantengan cuando el equipo no está conectado a la corriente. Lo incluimos en esta sección porque incluye un oscilador con una frecuencia de 32.768 kHz y la posibilidad de lanzar una interrupción (IRQ8) cada cierto intervalo programable de tiempo.

- HPET: "*high precision event timer*". Se trata de un dispositivo moderno para mantener varios temporizadores simultáneos y de alta precisión en un sistema.

Ahora que tenemos las distintas alternativas podemos pensar en diseñar el sistema de temporizadores.

El temporizador del APIC local será útil para gestionar el uso de CPU de los procesos. Podemos lanzar varias interrupciones por segundo (con 100 tendríamos un "*time slice*" o tiempo por proceso de 10 milisegundos, con 1000 uno de 1 milisegundo o incluso podríamos tener "*time slices*" que variaran en función de la prioridad del proceso, en todo caso estamos en un momento prematuro como para pensar con un nivel de detalle tan alto) y permitir a un proceso usar la CPU durante ese tiempo. Al terminar cada periodo comprobamos si otro proceso tiene mayor prioridad o si este proceso lleva demasiado tiempo y en caso de fuera así, mover el estado del proceso a memoria y recuperar otro proceso de un estado guardado anteriormente.

EL RTC nos sirve para tener la fecha y la hora. Al igual que otros Unix, haremos uso del "*Unix time*". Se trata de mantener en una variable el número de segundos que han pasado desde el 1 de enero de 1970. Cuando necesitemos convertir a año, mes, día, usaremos una función que convierta este dato a una estructura "*tm*" tal como la que usa la biblioteca estándar de C.

El TSC lo podemos usar para esperas cortas pero antes hemos de calibrarlo. El me-

canismo es el siguiente: arrancamos un temporizador de varios milisegundos, leemos el TSC y bloqueamos; cuando llegue la interrupción leemos de nuevo el TSC y restamos a este nuevo valor el anterior (y abrimos paso al código que estaba bloqueando puesto que en caso de no hacerlo dejaríamos el sistema bloqueado). Dado que sabemos el tiempo que pasó entre ambos y como la resta es el número de ciclos de CPU que ocurrieron en ese intervalo, tenemos la frecuencia del procesador. En la práctica ejecutaremos este mecanismo varias veces y calcularemos el promedio ya que no es un método muy científico (podría llegar una interrupción entre medias y de hecho ¡OS no detecta bien la velocidad del procesador en todos los emuladores en que se ha probado).

Aunque la intención es usar algún día el HPET, ¡OS usa el PIT para el temporizador global del sistema ya que es más sencillo de programar y está disponible siempre. Para el planificador de tareas se ha dejado listo el APIC (al haber uno por CPU podríamos cambiar de proceso en cada procesador de forma independiente). Para espera ocupada se usa TSC (funciones "*nsleep*", "*usleep*" y "*msleep*"). Y para la hora del sistema no nos queda de otra que usar el RTC.

Aunque no puede haber procesos bloqueados (de hecho, ni siquiera puede haber procesos), en un futuro se podría mantener una lista de eventos futuros e ir comprobando en cada interrupción provocada por uno de los temporizadores si hay alguna tarea programada desde otra parte del código.

No tiene mucho sentido entrar en los pormenores de los temporizadores ya que estos están disponibles en el código. Al final, aunque debido a los detalles de bajo nivel del hardware se ha dedicado mucho tiempo a hacerlos funcionar de forma fiable, todos funcionan de la misma forma: se programan unos registros, una cuenta, una frecuencia, una interrupción, un handler y se inicia la cuenta; si es repetitiva no hay que hacer más y sino, al final del manejador de la interrupción se vuelve a programar otra vez la siguiente cuenta.

Si que hay que comentar el concepto de "*jiffies*", que es una variable que cuenta el número de interrupciones del temporizador global del sistema. Es común en los drivers programar cosas "para N *jiffies* más tarde" o similar, por tanto interesa al menos definir el concepto.

También se define en código una macro: HZ, que indica cuantos ticks de reloj hay por segundo. Es un valor que comúnmente se define como una macro y determina cada cuanto tiempo revisamos si hay otro proceso de mayor prioridad listo para correr.

Si pensamos que hoy en día los procesadores cambian de frecuencia (incluso a cero si el equipo está suspendido o hibernado), y que cada interrupción supone un gasto de energía extra, veremos que es mejor tener un menor número de interrupciones, pero

al mismo tiempo, queremos ofrecer al usuario una experiencia interactiva y que todo suceda rápidamente cuando se realiza una acción. Algunos sistemas operativos son internamente "*tickless*", es decir, en cada interrupción de reloj programan la siguiente y por tanto no tienen un intervalo fijo definido de antemano.

4.7. RTC

Como ya hemos comentado RTC es un dispositivo que se mantiene encendido y funcionando con una batería externa incluso cuando el equipo está apagado.

Vemos en esta sección cómo leer la hora del sistema. No es necesario para hacerse una idea general del funcionamiento del kernel pero es un buen ejemplo sobre el funcionamiento de cualquier dispositivo hardware.

Listado 4.12: Como leer la hora del RTC

```
static inline u8
rtc_read (u8 port)
{
        outb (0x70, port | (1<<7));
        return inb (0x71);
}

static void
rtc_write (u8 port, u8 data)
{
        outb (0x70, port | (1<<7));
        outb (0x71, data);
}


...

interrupts_disable ();
rtc_write (0xb, rtc_read (0xb) | 0b1000110);
interrupts_enable ();

msleep (500);

while ((rtc_read (0xa) & 0x80) != 0)
        ;

segundos = rtc_read (0);
minutos = rtc_read (2);
año = rtc_read (9) + 2000;
```

```
etc....
```

Las funciones `rtc_read` y `rtc_write` leen y escriben del dispositivo RTC. La llamada que escribe y por tanto programa el dispositivo ha de estar protegida porque si llegase una interrupción no se haría correctamente. El número binario que se le pasa indica: queremos interrupciones periódicas, en modo binario (también permite BCD), usando formato de 24 horas y sin DST ("*daylight saving*" u horario de verano) habilitado.

La siguiente instrucción inserta una espera, ya que el dispositivo la necesita para obtener el dato. 500 ms es mucho tiempo pero tenemos que pensar que el dispositivo es externo al sistema. Es posible que la espera se pueda hacer de otra forma pero es algo que se hace una vez en el arranque por lo que no dedica excesivo tiempo.

La lectura siguiente en el bucle se necesita porque si el bit indicado estuviera activo significaría que el RTC está actualizando la cuenta. Una vez que no está siendo actualizada podemos leer los datos que necesitemos.

¡OS no hace más uso que este del RTC, ya que actualiza la fecha y hora sumando 1 al *Unix time*. No obstante cada hora vuelve a leer del RTC para evitar desfases.

Un punto que hay que destacar es que con la implementación actual, la hora no se detecta de forma completamente correcta en algunos equipos sino que hay un desfase. Esto comenzó a ocurrir con un cambio de versión del emulador y resulta cuanto menos curioso ya que se supone que el emulador nos proporciona un hardware más estable y predecible. Como no ocurre en todos los equipos la resolución del fallo se ha pospuesto con la esperanza de que otra actualización lo solucione.

Capítulo 5

Manejo de la memoria

Una de las tareas más importantes de un sistema operativo es gestionar la memoria RAM del equipo. Vamos a ver el enfoque adoptado por ¡OS.

Antes de gestionar la memoria, tenemos que saber cuanta tenemos disponible. Al menos necesitaremos tanta como para guardar la imagen del kernel; esto no será problema puesto que en total (código, datos y *.bss*) ocupa una cantidad que difícilmente llegará a 1MB (con lo descrito hasta ahora junto con el sistema de memoria y de disco ronda los 100KB, y eso sin haber escatimado en ningún momento).

Por tanto disponemos del total de la memoria, menos lo ocupado por el kernel menos lo ocupado por la BIOS y otros dispositivos del PC.

En nuestro caso, como las páginas de memoria son de 2MB cada una y como el kernel se carga en el primer MB de memoria, siempre que no sobrepasemos ese primer MB ocuparemos una página. Parece ineficaz desperdiciar casi 2MB de RAM pero este número permanecerá constante en el futuro puesto que es difícil que el kernel sobrepase 1MB y en tal caso podríamos colocar el código a partir del primer MB y los datos en los huecos libres que nos deja la arquitectura antes del primer MB o viceversa, y si no fuera suficiente, se podría usar una página más, 4MB para la imagen del kernel y datos estáticos lo cual sigue sin ser una gran cantidad.

Los usuarios o consumidores de memoria RAM serán el kernel y los procesos de usuario.

El kernel usa memoria para los datos cuyo tamaño no puede conocer en tiempo de compilación, y los procesos de usuario consumen memoria primero al ejecutarse (su propio código, datos y *stack*) y después en tiempo de carga cuando piden memoria con "*malloc*" (posiblemente implementada con la llamada al sistema "*brk*", la cual cambia

el tamaño del "*program break*" o límite del segmento de datos).

Aunque el espacio de direcciones de los procesos podría verse junto con el sistema de memoria, al final es un consumidor como cualquier otro y se relaciona más fuertemente con la ejecución de procesos. Como prueba de ello en el desarrollo del ¡OS se implementó el sistema de memoria dinámica, el sistema virtual de archivos (VFS) y la necesidad de mapear segmentos (no confundir con los segmentos de la CPU) no aparece hasta que se quiere crear una entidad de ejecución (proceso).

En este capítulo será importante tener en mente la figura 5.1, la cual indica las capas que componen el sistema de memoria del ¡OS ya que iremos recorriéndolas de abajo hacia arriba (de menor a mayor grado de abstracción). Para mantener la coherencia con el código se siguen los nombres anglosajones que se le ha dado a cada una de las capas.

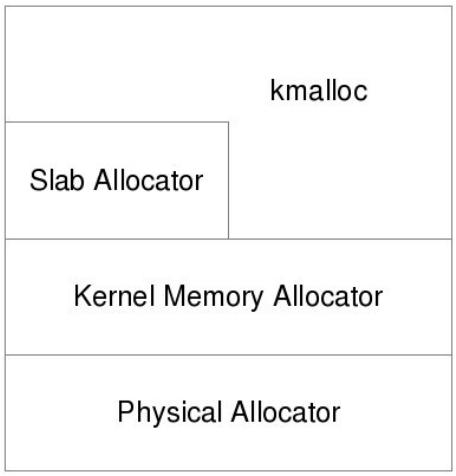

Figura 5.1: Capas del sistema de memoria

5.1. Gestión de memoria física (*physical allocator*)

Como dijimos, el cargador GRUB nos deja unos datos en memoria y si activamos el flag correspondiente (recordamos que esto se hace en la estructura de datos que se prepara contenida en el binario) uno de esos datos será la cantidad de memoria del equipo y los rangos usables.

Hemos visto como el dispositivo VGA se mapea en RAM, por tanto no podremos usar esas direcciones. Igualmente habrá otros dispositivos o direcciones reservadas para otras funciones. En todo caso se trata de intervalos de memoria RAM que no podremos usar.

También hemos visto que la memoria se divide en páginas, por tanto hemos de mantener una contabilidad de cuales están disponibles.

En este momento surge el problema de mayor dificultad de cuantos hemos visto hasta ahora: ¿donde guardar el estado de cada página? Este problema no existiría si supiéramos a la hora de compilar el kernel la cantidad de memoria disponible, pero no parece razonable pedir a cada usuario que se recompile su kernel cada vez que cambie la memoria instalada.

Una posible solución sería reservar una cantidad grande de memoria (la cual sería la máxima soportada por el kernel) y desperdiciar la que el sistema no tuviera instalada. Hoy en día parece una solución factible, hace unos años toda la memoria del sistema se habría ido en esta metainformación.

Otros sistemas operativos toman soluciones dispares. Linux por ejemplo usa lo que se conoce como "*boot memory allocator*" (ver [Gor08]), se trata de un sistema de memoria dinámica que usa de forma temporal hasta que puede retirarlo y pasar a usar el definitivo. Esta solución es buena pero si ya supone trabajo programar un sistema, programar dos es al menos el doble de trabajo.

¡OS usa la solución indicada más arriba, reserva memoria para un número de páginas y desperdicia los bits que referencien a una dirección de memoria no disponible. Como cada página ocupa 2MB, con un array de 32 elementos de 64 bits es capaz de cubrir 2048 páginas o 4GB de memoria. Perfectamente se podría usar un número mayor de elementos (ya que se ha definido con una macro) y soportar una cantidad mayor de memoria sin un gasto mayor del que supondría programar una solución más compleja. No merece la pena complicarse más porque es muy probable que el código necesario para implementarla, ocupara mucho más que nuestro array de bits.

El proceso seguido es suponer que no hay ninguna memoria usable y después recorrer los rangos que GRUB nos proporciona, marcando como disponibles las páginas que caigan completamente en un rango usable, salvo que no puedan ser usadas por alguna razón. En este momento no se dejan marcar como usables las páginas que contienen código del kernel ni las que contienen direcciones de memoria que mapean registros (como los rangos usados por los APIC).

El interfaz que esta capa proporciona es muy simple, una función para reservar páginas y otra para liberarlas, bien por número de página o por dirección de memoria. Cuando una página se entrega el bit que la representa se marca como ocupado y cuando se libera como libre.

¡OS no usa el clásico algoritmo del "*buddy allocator*". En un principio se programó una implementación sencilla pero se desechó por complejidad y porque con un

tamaño de página de 2MB, es muy probable que aunque sólo se pueda pedir una página cada vez, se nos proporcione más memoria contigua que la que nunca nos pueda dar un sistema operativo que use páginas de 4KB.

Este problema se puede solucionar casi siempre mapeando dos páginas cualesquiera contiguamente en memoria virtual (no siempre, ya que hay dispositivos más allá de la MMU que no ven memoria virtual). En general es poco probable que se llegue a necesitar nunca tanta memoria físicamente contigua en un proyecto amateur.

5.2. Memoria del kernel (*kernel memory allocator*)

Como indica la figura 5.1, este subsistema está colocado sobre el "*physical allocator*" y por tanto toda la memoria que pida será en páginas. La función de esta capa es entregar memoria en cantidades más pequeñas ya que sería impracticable entregar una página cada vez que alguien necesitara solamente unos bytes.

De todos los subsistemas de memoria este es el que contiene código más complejo. El diseño que sigue ¡OS es relativamente sencillo y la forma más simple de entenderlo es con un ejemplo.

Para el lector que lo conozca, se trata de una versión sencilla del "*buddy allocator*" que se comentó en la sección anterior, sólo que esta vez es para entregar trozos de páginas y no para páginas completas (algo poco común ya que tampoco es común usar páginas de 2MB).

Supongamos que nos piden 16KB de memoria (esta será la menor cantidad que esta capa entrega a las capas superiores, cuando veamos los "*slabs*" sabremos por que el límite inferior es tan alto), y supongamos que el "*pool*" de memoria libre está vacío (porque el KMA no ha solicitado aún páginas al "*physical allocator*" o porque dicho pool se agotó).

Como el KMA no tiene memoria disponible para su uso, pide una página (2MB), y la parte en dos; entrega una mitad al pool libre de 1MB y repite el proceso con el otro MB. Como sigue siendo demasiado grande repite el proceso, la mitad va para el pool de 512 KB, e intenta de nuevo con los otros 512 KB. El proceso se repite hasta que obtiene un trozo tal que dividido en dos sería demasiado pequeño.

En la siguiente reserva, el array que representa a los distintos pools, se recorre de menor a mayor hasta que haya un pool de tamaño suficiente con memoria disponible. Si hubiera memoria disponible se entrega salvo que fuera demasiado grande, en cuyo caso se divide tal como se hizo anteriormente, y si se terminara el array sin haber encontrado

memoria disponible del tamaño deseado, se pide otra página al "*physical allocator*" y se repite el proceso.

Este proceso tiene sus complicaciones y no es tan sencillo como parece. En caso de que hubiera 2 o más elementos en un mismo pool, ¿cómo se guardan? Aunque no lo hemos indicado, el pool es un array con un elemento para cada tamaño posible por lo que sólo puede guardar como mucho un elemento.

Podríamos mantener un array de arrays pero ¿de cuántos elementos? La opción elegante sería usar memoria dinámica pero precisamente es lo que estamos programando y salvo que reserváramos una página completa tampoco se podría, además de que la gestión de punteros en esa página sería compleja ya que habría que contabilizar a que elemento del array se asociaría cada uno. Esta solución sería posible pero no es buena y con un poco de ingenio podemos encontrar algo mejor.

El pool de elementos libres contiene NULL en caso de que no haya un elemento libre de ese tamaño y la dirección de memoria del elemento en caso de que lo haya. El contenido de esa dirección de memoria siempre estará libre y disponible para nuestros usos ya que precisamente se trata de elementos pendientes de ser entregados. Lo que hacemos es instalar en esa memoria unos punteros para tener una lista doblemente enlazada (podría ser enlazada sin más pero el API de listas sólo proporciona listas doblemente enlazadas), y cada vez que queramos un elemento, lo eliminamos de la lista y lo entregamos. Ya no hace falta memoria dinámica porque usamos cada elemento libre para apuntar al siguiente.

Pero el proceso complicado no es la reserva de memoria sino la liberación. Cualquier programador espera hacer algo así:

Listado 5.1: Malloc y free.

```
var = malloc (128);

...

free (var);
```

Y no:

Listado 5.2: Malloc y free no usuales.

```
var = malloc (128);

...

free (var, 128);
```

Por tanto cada vez que demos memoria hemos de apuntar el tamaño que entregamos.

Lo que haremos será guardar las entregas en una estructura de árbol, cosa que parece simple pero no lo es tanto puesto que al igual que antes, no sabremos cuantos elementos tendrá esa estructura y reservarlos dinámicamente no es posible porque se produce un círculo vicioso ya que no podremos entregar memoria hasta que no tengamos el árbol.

Como solución se ha optado por reservar estáticamente unos elementos de ese árbol. También se añade un flag que indica si el sistema de memoria se ha inicializado y en caso de que aún no lo esté usa el pool estático. Cada vez que libere tiene que comprobar si la dirección pertenece al pool y en tal caso, marcarla como libre.

Como ese pool se puede terminar cuando haya más entregas, solamente lo usa en lo que inicializa el subsistema y después el propio KMA usa un *slab* para generar un árbol dinámicamente (algo que por cierto provoca recursividad por lo que habremos de ser cuidadosos). Vemos en la siguiente sección como funcionan los slabs pero no parece una violación de capas ya que cualquier subsistema del kernel pide memoria y el KMA es un subsistema más. La única peculiaridad es que necesita esa memoria antes de poder pedirla y por eso se usa el array estático. Hay comprobaciones para avisar en caso de que el array se desbordara y en caso de que ocurriera sería en el arranque del sistema ya que como decimos, es cuando se usa.

5.3. Reserva en losas (slab allocator)

Como vimos, el tamaño más pequeño de los bloques de memoria que entrega el KMA es de 16KB. Es probable que la mayoría de peticiones de memoria que necesiten otros subsistemas del kernel sean menores, pero si hiciéramos que KMA entregase bloques más pequeños, con el tiempo sería difícil unir bloques pequeños en grandes al liberar memoria, es decir, habría fragmentación (lo que se conoce como fragmentación externa, la interna sería la memoria que perdemos al entregar trozos de memoria más grandes de los que nos pidieron).

Por otro lado, sería interesante contabilizar la memoria que usa cada subsistema y saber cuanto se ha gastado por ejemplo en bloques de disco o en *sockets*. Cada subsistema podría usar una función propia para reserva y contabilidad de memoria, pero sería tedioso tener que repetir la misma función y caótico exportar datos tan similares y dispersos a los usuarios y aplicaciones; además sería casi imposible tomar decisiones globales, como limitar a un tanto por cien la memoria de un subsistema o el número de

objetos dedicados a una tarea, así como comunicar a un subsistema que reduzca su uso de memoria en caso de escasez (es común en caso de que aumente la presión sobre la memoria ram, reducir cachés de inodos, de buffers y de "*dentries*", que ¡OS por ahora no usa y se emplean como entradas de directorios o "*dir entries*").

Los interfaces "*kmalloc*" y la capa de "*slabs*" (en código conocida como "*kcache*" o "*kernel cache*") intentan resolver todos estos problemas.

El interfaz a los *slabs* es simple y se compone de cuatro funciones, dos para crear y destruir una caché de objetos, y otras dos para reservar o liberar objetos en esa caché. La ventaja de gestionar los objetos en esta capa, es, una vez más y resumiendo, que el código es común para los subsistemas que la usen y además es posible añadir gestión y optimizaciones.

Aunque la capa se conoce como "*slab allocator*" sería más correcto usar "*kernel cache*" porque en realidad se trata de una caché de objetos que se compone de *slabs*.

La implementación de ¡OS es, de nuevo, muy simple (poco más de 300 lineas incluyendo comentarios). Cuando se crea una caché, se le indica el tamaño de los objetos y en función de ese tamaño reserva una cantidad de memoria inicial.

Para escoger este tamaño inicial se ha usado un script en varios sistemas Linux (sobre el archivo */proc/slabinfo*), y se han calculado los promedios de uso de memoria para slabs con objetos de distintos tamaños, y en base a ellos se ha escogido una cantidad inicial. Dado que el algoritmo está contenido en una función, podrá modificarse fácilmente en el futuro en caso necesario o se podrían incluso ofrecer varios algoritmos intercambiables al usuario. Para los curiosos esta es la función:

Listado 5.3: Tamaños de slab según tamaño de objeto

```
static u32
cache_get_size (size_t objsz)
{
        u16 sz;

        objsz >>= 10;

        if (objsz <= 8)
                sz = 64;
        else if (objsz <= 16)
                sz = 128;
        else if (objsz <= 32)
                sz = 256;
        else
                sz = 512;
```

```
        return sz << 10;
}
```

En caso de que la reserva inicial de memoria se termine, se pide una cantidad que será el doble a la anterior y así sucesivamente. Para liberar, en caso de que un slab quede vacío se libera sin que el usuario de la caché tenga que hacerlo explícitamente (de hecho el usuario de la caché no tiene forma de saber el estado de los slabs).

Esto presenta un problema, si casualmente nos piden un objeto que requiere de más memoria, esta se solicitará, si justo después lo liberan, la memoria se liberará; es posible que estas peticiones y liberaciones requieran que la capa inferior (el "*kernel memory allocator*") tenga que dividir y juntar trozos, e incluso ¡podría darse el caso de que este proceso fuera recursivo! Como sería computacionalmente pesado, se añade una tolerancia, no se libera un slab hasta que al anterior no baje de un cierto uso (digamos un 95 % aunque el número exacto puede variar).

Aún queda otro problema con este subsistema que no hemos resuelto: donde colocar la información sobre una caché y sobre un slab. Como los slabs serán grandes (al menos de 64KB como podemos ver en el listado 5.3), la estructura con información de la caché se coloca al principio del primer slab (como consecuencia, este slab no se libera nunca), y la que contiene información de cada slab al principio también de cada uno salvo en el primero, que va detrás de la información de la caché.

Este sistema es mucho más sencillo que los usados en el resto de sistemas Unix ya que como tienen páginas de 4KB, se hace mucho más complicado y los slabs constan de páginas mientras que en ¡OS eso no puede ocurrir por diseño.

El artículo original que describe el sistema de slabs lo podemos encontrar en [Bon94]. Este artículo además menciona lo que se conoce como "*slab coloring*", que consiste en añadir un número (color) de bytes de "*padding*" (es decir, se deja un espacio en blanco) al principio de cada slab, de forma que se maximice el uso de las cachés de la CPU (recordemos que estas cachés son asociativas). ¡OS no implementa tal funcionalidad aunque sería sencilla de implementar y transparente para el usuario, además de interesante de cara al rendimiento.

5.4. Reserva general de memoria (kmalloc)

Para terminar sólo nos falta la reserva de memoria de cantidades aún más pequeñas. Esta capa sería opcional y podría hacerse que la capa de slabs entregara slabs más pequeños. Pero es más optimizado crear una caché para cada posible tamaño que nos

pudieran pedir, y como lógicamente no podemos cubrirlos todos, lo hacemos en potencias de dos. De esta forma se crean cachés para 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 y 8192 bytes, y kmalloc, cuando va a pedir memoria usa una u otra caché en función de la cantidad. En caso de que nos pidieran más de 8 KB, usaría el KMA.

Para liberar un objeto, sería necesario saber en que slab está, por ejemplo guardando en las direcciones anteriores a la entregada el tamaño para esa entrega (lo cual haría que para reservar por ejemplo 8 bytes se usaran 16 ya que cruzaría al siguiente tamaño). Tal vez se podría investigar como optimización, pero lo que hace ¡OS, es intentar liberar en la caché más pequeña y si ese objeto no está, `kcache_free` devuelve cero por lo que se prueba en el siguiente hasta que se encuentre en alguno; lógicamente si no estuviera en ninguno es un error porque están liberando una dirección que no se entregó.

5.5. Conclusión

El sistema de memoria dinámica de ¡OS no es perfecto pero proporciona una buena base. Cuando un subsistema necesite páginas completas (por ejemplo cuando se cree un proceso), podrá pedirlas en tamaño de páginas al"*physical allocator*", cuando necesite memoria para varios objetos iguales podrá usar la capa de slabs y cuando necesite simplemente unos bytes (hasta 16KB), podrá usar *kmalloc*.

Además, de forma conceptual, no es muy distinto que cualquier sistema de memoria de un Unix, con la diferencia de que se codifica en poco más de 1000 lineas de código y en comparación, el de Linux ocupa más de cien veces más entre código independiente de la arquitectura y código para x86/x86-64 (como ambas arquitecturas son parecidas, Linux comparte archivos entre ellas).

Capítulo 6

Disco

Si pensamos en dispositivos como si fueran objetos de software, nos daremos cuenta de que todos tienen (o pueden tener) más o menos las mismas operaciones: abrir (o hacer uso), cerrar (o liberar), leer y escribir, etcétera. Parece razonable por tanto pensar en agruparlos de alguna forma.

Aunque se podrían tratar todos de la misma manera, ¡OS sigue el modelo de Unix y separa los dispositivos en dos tablas, una con los llamados dispositivos de bloque y la otra con los de carácter. La principal diferencia es que los de bloque proporcionan un método[1] para leer o modificar el offset del dispositivo. Por ejemplo, en un disco podemos leer de una parte o de otra, pero un ratón, un teclado o un modem, tenemos que leer obligatoriamente el dato que llega en ese momento sin poder elegir la posición.

El desplazamiento en esa tabla de dispositivos es un número que identifica al tipo de dispositivo y se conoce como "*major number*". Como podemos tener varios dispositivos del mismo tipo, se proporciona otro número que se conoce como "*minor number*". El *major number* se asocia con un driver y el *minor number* se le pasa como parámetro al driver para que opere sobre el dispositivo adecuado.

Como veremos, otra diferencia entre dispositivos de bloque y de carácter, es que los primeros tienen caché. Tiene sentido guardar en memoria los últimos datos leídos de un disco porque es posible que se vuelva a necesitar en el futuro (imaginemos cuantas veces se ejecuta una porción de código de una biblioteca estándar o cuantas veces se lee un documento sobre el que estamos trabajando), pero no tiene sentido guardar las posiciones de teclado o los movimientos del ratón ya que es poco probable que estos

[1] Aunque se usa la terminología de la programación orientada a objetos, en código se implementan con punteros a funciones metidos en estructuras ya que C no es un lenguaje orientado a objetos pero permite simularlo.

datos se necesiten una vez que se ha escrito la tecla o actualizado la posición del ratón[2].

Es posible que un mismo dispositivo se encuentre en el sistema en modo bloque y modo carácter. En el caso de un disco nos permitiría saltarnos las cachés del sistema (con riesgo de una posible incoherencia) y para otros dispositivos puede tener otros usos que dependerán del dispositivo en cuestión. No todos los dispositivos tienen acceso en modo bloque y en modo carácter por lo que no es algo que podamos dar por hecho.

A nivel de código y para hacerlo más flexible, podemos insertar unas funciones que permitan a los drivers registrar y liberar dispositivos. De esta forma la tabla de dispositivos podría implementarse de cualquier forma sin que el interfaz se vea afectado. Esta es la función que registra los dispositivos de bloque (la que registra los de carácter será similar):

Listado 6.1: Registrando dispositivo de bloque.

```
void
bdev_register_dev (struct bdevsw *bdev, u32 major)
{
        if (major > BMAJORMAX)
                kpanic ("BDEV: too big major number registered.");

        bdevsw[major] = bdev;
}
```

La función comprueba que el número indicado está dentro de los límites y la estructura que define las operaciones se asocia mediante un puntero en la tabla global de dispositivos de bloque. Para leer sólo tendremos que llamar al método adecuado (el cual teniendo el major number será un simple offset en la tabla):

Listado 6.2: Leyendo de un dispositivo de bloque

```
size_t
breadu (dev_t *dev, size_t pos, void *addr, size_t count)
{
        if (dev->major >= BMAJORMAX)
                return 0;

        if (bdevsw[dev->major]->read == NULL)
                return 0;

        return bdevsw[dev->major]->read (dev, pos, addr, count);
}
```

[2]Esto no significa que no pueda haber un buffer que guarde los datos hasta que los lea una aplicación de usuario. En todo caso, una aplicación cualquiera podría gestionar una caché propia.

Se comprueba si existe un driver para el dispositivo y si proporciona un método de lectura, en tal caso lo llama.

Una cuestión que hemos dejado deliberadamente en el aire es ¿cómo se llama al driver adecuado desde una aplicación cualquiera? Aunque lo veremos de forma práctica con el sistema de archivos virtual (VFS), cada dispositivo se representa en disco por un archivo especial. Este mecanismo no supone nada extraordinario, simplemente se trata de unos archivos que no ocupan espacio (salvo por el inodo como veremos) y que sólo contienen los números major y minor. Podremos llamarlos de cualquier manera, ubicarlos en cualquier directorio, etcétera, siempre que contengan los números correctos, aunque la convenció es que vayan en el directorio */dev*.

6.1. IDE/ATA

La especificación IDE (después ATA y luego SATA) es una de las más complejas necesarias para este trabajo, y una vez más dejaremos los detalles de implementación puesto que se encuentran disponibles en el código. No se trata de pequeñas cosas ya que hacer funcionar correctamente este driver fue una de las partes que más tiempo y compilaciones infructuosas llevó.

Un disco duro IDE puede funcionar en varios modos. En un sistema moderno lo hará usando alguna variante de DMA, que consiste en un dispositivo programable que transfiere datos entre disco y memoria sin intervención de la CPU (nótese que la CPU aún se necesita para programar la transferencia y que la mejora de rendimiento dependerá de la secuencialidad de los datos que queramos mover). La ventaja de no usar la CPU es que, dado que la diferencia de velocidad es grande, la CPU puede estar haciendo otras cosas en lo que se transfieren los datos.

Usar DMA es complejo: existe un planificador de E/S (del inglés "*I/O scheduler*" que es capaz de reordenar las peticiones que se envían a los dispositivos e incluso de juntarlas (del inglés "*merge*"), de forma que se optimice el movimiento giratorio de los discos. En el caso de las lecturas, el proceso que hizo la operación ha de ser bloqueado hasta que esta esté disponible (en el de escritura no es necesario salvo que se especifiquen flags puesto que esta se puede retrasar).

Para ¡OS no se ha optado por esta solución debido a la complejidad. En el futuro, su implementación es de las funcionalidades con mayor prioridad aunque con los nuevos discos SSD puede no ser tan importante dado que tienen tiempos de acceso rápidos y la secuencialidad de las operaciones no es significativa (dado que tienen transferencias que están un orden de magnitud por encima de los discos rotacionales, es posible

que sea deseable diseñar esta capa de otra forma). Y lo mismo ocurre en discos que funcionan en controladoras de gama alta ya que la propia controladora reordena las operaciones para conseguir un rendimiento óptimo.

El método usado por ¡OS se conoce como PIO (del inglés: "*programmed input/output*"), que es un método más antiguo en el que la CPU hace las operaciones una a una y transfiriendo 16 bits cada vez. La CPU queda ocupada esperando a que se copien los datos y no es necesario que haya después una interrupción que indique que la finalización de la operación puesto que sabemos que ha finalizado cuando termine el bloque de código que la hace.

Volviendo al interfaz para discos IDE, lo más común es que un PC tenga al menos una controladora con dos canales, uno maestro y otro esclavo, pero puede haber cualquier configuración y no podemos presuponer nada.

La primera función de nuestro driver será detectar la controladora, y para ello no existe ningún método estándar (o mejor dicho, el que existe no funciona). Lo que hacemos es escribir un dato cualquiera en un par de registros y si podemos leerlo es que hay una controladora. Es importante que las operaciones de lectura y escritura no sean consecutivas, según la poca documentación encontrada (ya que se trata de un método no estándar), sería posible que se leyera del bus el último dato escrito aunque no hubiera dispositivo y por tanto podríamos pensar que hay controladora cuando no la hay.

En caso de que la hubiera tenemos que comprobar en cada uno de los canales si hay un disco conectado y esta vez sí tenemos un método estándar que consiste en lanzar una operación "*identify*" que identifica las características del disco. Sólo para que nos hagamos una idea general del código reproduzco unos fragmentos de la parte que detecta un disco:

Listado 6.3: IDE identify.

```
static void
ide_cmd_write (struct _drive *dev, u16 reg, u8 data)
{
        ide_ready_wait (dev);

        outb (dev->channel->iobase + reg, data);
}


static u8
ide_identify (struct _drive *dev)
{
        /* Las operaciones afectan al dispositivo seleccionado.
         * Esta función cachea el dispositivo en uso y si cambia
```

```
         * y fuera necesario, selecciona el nuevo. */
        ide_dev_sel (dev);

        /* Esta función envía el comando identify: */
        ide_cmd_write (dev, ide_cmdreg, IDECMD_IDENTIFY);

        /* Lee el puerto de estado.
         * Si devuelve cero, la unidad no existe,
         * El bit 5 significa que hubo un error en la unidad
         * y el 0 que hubo cualquier otro error. */
        u8 s = inb (dev->channel->iobase + ide_status_alt);
        if (s == 0 || bittest (s, 5) || bittest (s, 0))
                return 0;

        /* Leemos en un array los datos que devuelve identify: */
        for (c = 0; c <= 255 ; c++)
                b[c] = ide_cmd_read16 (dev, ide_datareg);

        /* Ahora sólo hay que ir analizándolos: */

        /* Regresa si el dispositivo no es ATA: */
        if (b[0] & 0x800)
                return 0;

        /* Regresa si no soporta LBA 48: */
        if ((b[83] & (1<<10)) == 0)
                return 0;

        /* Después de otros datos y comprobaciones omitidos,
         * detecta el número de sectores: */
        dev->nsectors = ((u64)*(b + 103) << 48)
                | ((u64)*(b + 102) << 32)
                | (*(b + 101) << 16)
                | *(b + 100);

        /* Si falla LBA 48 probamos con LBA 28: */
        if (dev->nsectors == 0)
                dev->nsectors = (*(b + 61) << 16) | *(b + 60);

        return 1;
}
```

No hay mucho más que decir de esta parte puesto que sólo se trata de inicializar el dispositivo para poder leer después.

6.2. Lectura y escritura de disco

Volviendo a la tabla de dispositivos, nuestro driver hará en la estructura que registra en dicha tabla, algo así (nótese que se usan extensiones de GCC para asignar los campos):

Listado 6.4: Operaciones IDE

```
...
.read = &ide_read,
.write = &ide_write,
...
```

Como ambas operaciones son similares, podemos examinar la lectura y hacernos una idea de la escritura (la cual puede encontrarse en código).

Los parámetros que necesita la función de lectura que registramos tienen que coincidir con lo que nos ofrece la capa de dispositivos de bloque y cualquier dispositivo de bloque que registremos tendrá los mismos. Son estos:

Listado 6.5: Función de lectura para discos IDE

```
static size_t
ide_read (dev_t *device, size_t pos, void *addr, size_t count)
{
...
```

En primer lugar recibe un dispositivo, el cual simplemente contiene un par de números major y minor, después recibe la posición de disco de la que queremos leer, la dirección de memoria donde se guardan los datos leídos de disco y el tamaño de dicha dirección de memoria.

Dado que podemos tener varias controladoras y discos, ¿cómo repartimos el uso de números major y minor? Por un lado podemos usar un major para cada controladora o para cada disco, y por otro podemos usar el mismo major para todos y repartir entre los minor.

No hay ninguna ventaja o desventaja por hacerlo de una forma u otra por lo que hemos optado por usar el mismo major para todos los discos (IDE, ya que para otro tipo de disco hará falta otro driver) y reservar 16 minor para cada disco. De esta forma estamos limitados a 15 particiones (veremos que son las particiones en la siguiente sección, es necesario posponerlo ya que para detectarlas necesitamos lectura), ya que el primero se usa para representar al disco entero y por hacernos una idea, la partición número cinco del primer disco de la segunda controladora tendría el número minor 37. En el

futuro (y dado que pasará mucho tiempo hasta que necesitemos compatibilidad binaria con versiones anteriores), podría ser interesante hacer estos números compatibles con su numeración.

Regresando a la función, su principal tarea es comprobar las operaciones para que sólo se hagan dentro de los límites de una partición. Es importante controlarlo cuanto antes (en las capa de nivel más bajo posible) porque podría darse que un error en cualquier otra parte hiciera que se leyera o escribiera fuera de los límites permitidos (por ejemplo, en otra partición).

Dado que para leer las particiones necesitamos lectura y dado que para leer necesitamos las particiones, ¿cómo rompemos esta recursividad?

El código que lee las particiones no usa esta función sino una de menor nivel: "`ide_read_blocks`". Esta función si que realiza la lectura del dispositivo y se implementa aproximadamente así:

Listado 6.6: Función de lectura a bajo nivel

```
static size_t
ide_read_blocks (struct _drive *dev, size_t sector,
        void *data, size_t count)
{
        u16 *dptr = data;
        size_t seccount;

        /* Es necesario que al menos se lea un bloque
         * ya que es la unidad mínima de lectura: */
        seccount = count / dev->blocksize;
        if (seccount == 0)
                return 0;

        /* Seleccionamos el dispositivo correcto: */
        ide_dev_sel (dev);

        /* Carga la cuenta que va a leer y la dirección LBA.
         * El orden de los comandos es extraño porque escribir
         * dos veces seguidas en un mismo puerto es más lento
         * que intercalar los puertos. */
        ide_cmd_write (channels->drive,
                ide_seccount0, seccount >> 8);
        ide_cmd_write (channels->drive,
                ide_lba0, ((u64)sector >> 24) & 0xff);
        ide_cmd_write (channels->drive,
                ide_lba1, ((u64)sector >> 32) & 0xff);
        ide_cmd_write (channels->drive,
                ide_lba2, ((u64)sector >> 48) & 0xff);
```

```
        ide_cmd_write (channels->drive,
                ide_seccount0, seccount & 0xff);
        ide_cmd_write (channels->drive,
                ide_lba0, (u64)sector & 0xff);
        ide_cmd_write (channels->drive,
                ide_lba1, ((u64)sector >> 8) & 0xff);
        ide_cmd_write (channels->drive,
                ide_lba2, ((u64)sector >> 16) & 0xff);

        /* Una vez cargados los datos, lanza la lectura: */
        ide_cmd_write (channels->drive,
                ide_cmdreg, IDECMD_READ_SECTORS_EXT);


        /* Cada lectura lee 2 bytes por lo que son la mitad
         * de operaciones: */
        count = seccount * dev->blocksize >> 1;

        do {
                ide_ready_wait (dev);

                *dptr++ =  inw (dev->channel->iobase + ide_datareg));

                if (--count == 0)
                        break;
        } while (1);

        return seccount * dev->blocksize;
}
```

6.3. Particiones

Ahora que podemos leer de disco, como dijimos, lo primero que tenemos que leer son las particiones.

Las particiones son divisiones administrativas de los discos. Aunque podríamos tener todos los datos desperdigados por igual, es común hacer divisiones y usar cada una para datos o sistemas operativos diferentes.

Las particiones se definen en el primer sector de disco (como vimos, MBR o "*master boot record*", más información en: `http://en.wikipedia.org/wiki/Master_boot_record`). Será necesario hacer una operación como esta:

Listado 6.7: Función de lectura a bajo nivel

```
ide_read_blocks (dev, 0, first_sector, 512);
```

Una vez leído tendremos que comprobar si es válido (byte 510 con valor 0x55 y byte 511 con valor 0xaa), y leer sus 4 posibles entradas. Estas entradas contienen el tamaño de la partición, la ubicación, el tipo y unos flags.

Esta es la razón por la que el PC sólo soporta 4 particiones que se conocen como primarias. Dado que resultaron insuficientes, se reservó un tipo de partición primaria que se conoce como "extendida", y que contiene otra definición de particiones que se conocen como lógicas. Por ahora ¡OS no soporta particiones lógicas y esto quiere decir que tiene que instalarse en una primaria necesariamente y que no puede leer datos de una partición ubicada en una extendida. En el futuro no sería difícil añadir dicho soporte de particiones lógicas.

6.4. Sistema de archivos virtual, VFS

Un proceso de usuario tiene funciones de biblioteca para abrir archivos, para cerrarlos, para leer y escribir en ellos, para cambiar la posición de lectura, etcétera. Este interfaz se definió por POSIX ("*Portable operating system interface*") en el estándar "*IEEE Std 1003.1-1988*" y en otros posteriores que se conocen con el nombre de *SUS* ("*Single Unix specification*") y por tanto es una base de la que partir al diseñar los subsistemas.

Esas funciones de biblioteca se apoyan en llamadas al sistema y estas han de ser iguales sin tener en cuenta con que sistema de archivos estén tratando.

VFS es la capa que recibe esas peticiones y la que hace que el acceso sea transparente e independiente del sistema de archivos en uso. Pero no sólo proporciona interfaz a las capas superiores, también proporciona un interfaz para que las inferiores (sistemas de archivos) puedan registrarse.

Digamos que en alguna parte del código (nos referimos a código kernel pero en caso de ser código de usuario no sería muy distinto como se verá más adelante) se va a ejecutar algo así:

Listado 6.8: Apertura lectura y cierre de archivo.

```
int fd = open ("/dir/archivo", 0, O_RDONLY);

lseek (fd, 256, SEEK_SET);
```

```
read (fd, buffer, 512);

close (fd);
```

En el resto del capítulo será importante tener este código tan simple presente ya que vamos a examinar lo que sucede cuando se ejecuta.

Cuando se abre el archivo, se localiza el número de descriptor de fichero más bajo libre para ese proceso y se reserva. Como aún no tenemos la noción de proceso, vamos a suponer que existe una estructura de tipo "*task*" que contiene una entrada así[3]:

Listado 6.9: Descriptores de archivo para un proceso

```
struct file *fds[NFDS];
```

¡OS define la estructura "*file*" como:

Listado 6.10: Estructura file

```
struct file {
        struct inode *inode;
        u16 count;
        u16 mode;
        u16 flags;
        off_t pos;
};
```

Y la de inodo como:

Listado 6.11: Estructura de inodos

```
struct inode {
        struct super *sb;
        size_t num;
        u16 mode;
        u8 nlinks;
        uid_t uid;
        gid_t gid;
        size_t filesize;
        size_t nblocks;
        time_t atime;
        time_t mtime;
        time_t ctime;
        u32 flags;
        u32 count;
```

[3]Este paso preliminar no complicará el paso a la multitarea ya que el diseño habitual es declarar una variable global que apunte en cada momento al descriptor de la tarea activa en ese momento.

```
        struct list_head l;
        struct super *covered;
        struct inode_ops *ops;
        void *priv;
};
```

Nótese que esta es la estructura genérica para un inodo, además existirá otra similar no visible al resto de subsistemas para cada sistema de archivos, que será igual que la estructura de los inodos en disco (en caso de que el sistema de archivos en cuestión soporte inodos ya que no todos los sistemas de archivos funcionan con inodos, en caso de que uno no funcione con inodos, tendrá que rellenar la información como pueda).

Como podemos ver en esta sección se manejan muchas estructuras relacionadas por lo que habrá que prestar atención a estas relaciones a pesar de que ¡OS las simplifica mucho respecto de un modelo de un Unix cualquiera.

Volviendo a la discusión, una vez que tiene el descriptor libre más bajo, *open* llama a otra función que le devuelve una entrada *file* para el archivo que queremos abrir. Esta función localiza una entrada libre en la tabla, convierte la ruta en un inodo (ver la definición más arriba, de un inodo podemos decir que contiene toda la información sobre un archivo menos el nombre) y devuelve la dirección de dicha estructura.

Para convertir la ruta en un inodo usa la función "*namei*" (es un nombre estándar entre distintos UNIX, de hecho existe un comando de usuario para hacer exactamente lo mismo).

Esta función sólo recibe una ruta y a partir del directorio actual o del raíz (según sea una ruta relativa o absoluta respectivamente), va recorriendo los distintos elementos que la componen.

Volviendo al ejemplo, queríamos abrir "*/dir/archivo*", por tanto se trata de una ruta absoluta. Obtiene el inodo del directorio raíz[4] "*/*" el cual estará almacenado en una variable global, quizás distinta por proceso (esa variable global la inicializa el código que monta la partición, en seguida vemos que significa esto).

Por otro lado, si se tratara de una ruta relativa, usaría el directorio actual del proceso, el cual se encuentra en la estructura de la tarea. Volviendo a la función *namei*, si nos fijamos en la estructura que representa a un inodo, contiene un campo que son las operaciones sobre esos inodos; la función *namei* llama a la operación "*lookup*" que se declara así:

Listado 6.12: Operación lookup en un inodo

[4]El directorio raíz es el directorio padre de todos los directorios. En Unix cada proceso puede tener una visión diferente del sistema de archivos y por tanto un directorio raíz distinto.

```
struct inode *(*lookup) (struct inode *, const char *, u8);
```

Esta llamada es similar a la llamada a un método en programación orientada a objetos, pero vamos a ver otra cosa antes de ver como funciona.

Cuando el sistema operativo arranca, un sistema de archivos "se monta" sobre el directorio raíz. Podemos entender "montar" como "hacer visible". Este sistema de archivos cuando se registró a si mismo, registró también unas operaciones que proporciona, entre las cuales se encuentra "leer superbloque". Esta función devuelve información común sobre ese sistema de archivos concreto; más concretamente devuelve una estructura así:

Listado 6.13: Estructura de superbloque

```
struct super {
        dev_t dev;
        size_t blocksize;
        size_t blocksizephys;
        struct list_head bcache;
        struct list_head icache;
        struct super_ops *ops;
        u64 magic;
        time_t time;
        struct inode *mounted;
        u32 flags;
        void *priv;
};
```

Una vez más existe un superbloque general para el sistema y otro para cada sistema de archivos, y como vemos, de nuevo tiene operaciones (campo "*ops*"). Una de ellas lee un inodo y se declara así:

Listado 6.14: Operación de lectura de inodos:

```
struct inode *(*inode_read) (struct super *sb, u64 inum);
```

Cuando el sistema de archivos raíz se monta, se lee el inodo para "/", y este contiene la operación lookup llamada por *namei*. La operación que resuelve el path no la ejecuta por tanto VFS sino el sistema de archivos que contenga la componente actual del nombre (y por tanto el inodo) que tratamos de resolver.

Por ahora jOS no soporta puntos de montaje (salvo para el raíz), pero podría darse el caso de que para resolver un path hubiera que recorrer varios sistemas de archivos diferentes. Tampoco sería difícil puesto que el inodo contiene un puntero a sus operaciones y por tanto se consultaría el sistema de archivos correcto (al ejecutar mount

sobre un directorio, se le asociarían las operaciones sobre otro sistema de archivos y por tanto perderíamos visibilidad del anterior).

Como esta sección ya es de por sí densa, trataremos la lectura física del inodo más adelante cuando veamos el sistema de archivos. Lo importante por ahora es que *namei* va resolviendo la ruta hasta que devuelve el inodo del componente final, en este caso el archivo "*archivo*".

Una vez que open tiene el inodo final, marca en la estructura que representa al archivo que queremos abrir, el modo de apertura, los flags y la posición, y devuelve el número de descriptor.

La siguiente operación coloca el puntero de lectura en la posición 256 a contar desde el principio del archivo. "*lseek*" se implementa muy sencillamente por lo que la reproducimos:

Listado 6.15: Función lseek

```
off_t
lseek (s32 fd, off_t offset, u8 origin)
{
        switch (origin) {
                case SEEK_SET:
                        current->fds[fd]->pos = offset;
                        break;
                case SEEK_CUR:
                        current->fds[fd]->pos += offset;
                        break;
                case SEEK_END:
                        current->fds[fd]->inode->filesize += offset;
                        break;
                default:
                        return (off_t) -1;
        }

        return current->fds[fd]->pos;
}
```

Nótese que es posible colocar el puntero más allá del final del archivo. En tal caso el estándar nos dice que una lectura siempre devolverá ceros.

A la hora de leer, read llama a la función "`block_read`". Esta operación llama a la operación "`bmap`" del sistema de archivos, la cual resuelve una combinación de inodo y offset en un bloque de disco (es decir, en que bloque se encuentra una posición de un archivo), y cuando sabe de que bloque lógico leer, convierte a bloque físico (el sistema de archivos se instala en una partición y por tanto podría ser distinto del bloque físico)

y devuelve el *buffer*.

En este proceso hemos omitido dos cosas: la caché de buffers (del inglés "*buffer cache*") que veremos más adelante (por ahora nada más necesitaremos saber que si un buffer está en memoria, no es necesario leerlo de disco), y la representación de un buffer.

Dado que un buffer es un conjunto de datos de un tamaño dado, necesitamos una estructura que lo defina. El nombre clásico de Unix para esta estructura es "*bhead*" (de "*buffer head*"), y ¡OS la define así:

Listado 6.16: Estructura que describe un buffer

```
struct bhead {
        struct list_head l;
        size_t bnum;
        void *data;
        u8 count;
};
```

El primer byte de un buffer siempre será múltiplo de 512 ya que los discos se leen en sectores de 512 bytes cada uno (como reseña, algunos discos modernos trabajan con bloques de 4096 bytes, ¡OS por ahora no soporta este modo de trabajo; cabe decir que nuestro trabajo más sencillo puesto que la mayoría de sistemas de archivos trabajan con bloques lógicos de 4KB).

En este caso leemos a partir de la posición 256 ya que se cambió la posición mediante "*lseek*", pero esta posición podía haberse modificado igualmente mediante un read anterior.

La función de lectura tiene que obtener el primer bloque, el cual se obtendrá de memoria si se leyó recientemente (de la caché de buffers), o de disco si no estaba en memoria, copiar los 256 bytes finales a los 256 bytes iniciales del buffer que nos pasaron, leer el siguiente bloque (de nuevo de memoria o disco) y copiar los 256 bytes a la parte final del buffer.

De este proceso no decimos nada ya que se trata de un simple movimiento de bytes que sólo se complica un poco más cuando tenemos en cuenta todas las posibilidades (fin del archivo, lectura al principio, medio o final de bloque, etcétera). Ya sólo es calcular los bytes leídos y devolver el dato como valor de retorno de la función.

La función close aún no ha sido implementada pero sería sencillo añadirla: liberaría el descriptor, decrementaría el contador de referencia ("*refcount*") en la tabla de archivos y liberaría el archivo de la tabla en caso de que fuera cero. Es decir, deshace

lo que hizo la función de apertura.

Es necesario mantener el contador de referencia porque en caso de que se creara un proceso hijo (algo de lo que con lo visto hasta ahora, ¡OS está lejos), este compartiría entradas en la tabla de archivos y podríamos tener dos (o más) descriptores apuntando al mismo archivo.

El sistema de archivos virtual para ¡OS es sencillo (unas 700 lineas de código) pero consta de muchas estructuras fuertemente relacionadas. De momento sólo se ha implementado un sistema de archivos pero no sería complicado añadir otros. Sin embargo tiene carencias como por ejemplo los "*dentries*" que representan entradas de directorio o la escritura. Todas ellas serán implementadas conforme se necesiten. Como anécdota, resulta curioso lo tarde que se necesita la escritura en disco ya que en principio, ningún proceso del kernel la require.

6.5. Sistema de archivos (extended 2)

A la hora de escoger un sistema de archivos, había varias opciones, incluso se podría haber implementado uno propio, pero se escogió *extended 2* (conocido también como *ext2* o por su nombre completo: "*second extended filesystem*") por ser estándar y simple de implementar. Su especificación podemos encontrarla en [Poi].

Después de *extended 2* se desarrollaron *extended 3* y *extended 4* pero estos sistemas de archivos tienen características avanzadas por lo que no serían tan simples de programar. La primera versión funcional, al menos de lectura, de la implementación de *ext2* en ¡OS ocupó unas 500 lineas de código; por hacernos una idea, la implementación de *extended 2* del kernel de Linux ocupa unas 10 mil lineas, la de *extended 3*, 18 mil y la de *extended 4* ocupa casi 40 mil.

Para implementar este sistema de archivos sólo hay que ir siguiendo la especificación y tener unas funciones de la capa de bloque que operen de forma fiable. De hecho, si lo intentáramos implementar como un proceso de usuario sobre una imagen de disco (o incluso abriendo el archivo que representa a una partición de un disco duro), sería algo mucho más fácil de lo que a priori pueda parecer.

Como ya vimos, el sistema de archivos registra unas operaciones en el VFS, tenemos dos tipos: operaciones de superbloque y operaciones de inodos. Esta sección consiste en describir estas operaciones ya que es el único interfaz mediante el cual la capa VFS puede acceder al sistema de archivos.

Las operaciones del superbloque se han mantenido simples: leer superbloque y

leer inodo. Con el tiempo se podrán necesitar más pero para nuestros propósitos son suficientes.

Leer el superbloque no supone ningún problema y la única complicación es que su ubicación (en bloques) depende del tamaño de bloque del sistema de almacenamiento. Como la función que "abre" el dispositivo devuelve el tamaño de bloque, usamos ese dato. Sin comprobación de errores este es el código:

Listado 6.17: Lectura del superbloque

```
blocksize = bopen (dev);
breadu (dev, 1024 / blocksize, buf, 512);
struct ext2_super *e2sb = (struct ext2_super *) buf;
...
```

Si como curiosidad queremos leer un superbloque de nuestro sistema de archivos extended en un linux, podemos usar cualquiera de estos comandos:

Listado 6.18: Comandos para lectura del superbloque

```
tune2fs -l /dev/sda1
...

dumpe2fs -h /dev/sda1
...

debugfs /dev/sda1
debugfs 1.42 (29-Nov-2011)
debugfs: show_super_stats -h
...
```

La operación de lectura de inodo se define en el superbloque porque el superbloque es la estructura que define a una instancia de un sistema de archivos, por tanto resulta una operación sobre el propio sistema de archivos.

La operación de lectura de un inodo es más compleja puesto que hay que localizarlo.

Extended 2 divide el disco en bloques y esos bloques se agrupan en estructuras que de forma no sorprendente se conocen como grupos de bloques (del inglés: "*block groups*"). Esto se hace para reducir la fragmentación y por tanto minimizar los movimientos de la cabeza lectora cuando los datos se leen de forma secuencial. La información sobre cada "*block group*" se guarda en una tabla en los bloques que va detrás del superbloque.

Al principio de cada "*block group*" hay dos bitmaps, uno para saber que bloques

están ocupados o libres y otro que realiza la misma función de contabilidad sobre los inodos de ese grupo. Además justo después de estos bitmaps se encuentra la tabla con los inodos del grupo. Ambas tablas ocupan un bloque por lo que el número de bloques y de inodos por grupo está limitado.

Después de esta introducción tenemos suficiente base como para entender el código que localiza el inodo cuyo número conocemos. El primer paso será leer el inodo para el directorio raíz "/", el cual suele ser el inodo número 11, pero como no podemos estar seguros de que siempre sea 11, tenemos que usar el número que nos proporciona el superbloque.

Dado que todos los grupos tienen el mismo número de inodos, jOS localiza el número de "*block group*" mediante una simple linea:

Listado 6.19: Localización del block group

```
bgroup = (inum - 1) / sbpriv->inodes_per_group;
```

Como hemos dicho, los grupos van en una tabla, luego el grupo que queremos será el offset en la tabla, con este código leemos el descriptor del grupo en el que se localiza el inodo:

Listado 6.20: Localización del inodo

```
/* Se posiciona en el bloque que contiene la entrada
 * que queremos en la tabla con los descriptores de grupos.
 *
 * El código no es tan difícil como parece y el bloque en el
 * que está bgroup será:
 * bloque base de tabla de grupos +
 * número de grupo / número de elementos en cada grupo.
 *
 * El número de elementos de cada grupo es igual a:
 * tamaño de bloque / tamaño de cada elemento
 */
struct bhead *buf = ext2_bread (sb, sbpriv->bgroup_base
        + bgroup / (sb->blocksize / sizeof (struct ext2_bg_desc)));

struct ext2_bg_desc *e2desc = (struct ext2_bg_desc *) buf->data;

/* Se posiciona en el elemento concreto en el bloque leído: */
e2desc += bgroup % (sb->blocksize / sizeof (struct ext2_bg_desc));
```

Los descriptores de grupos se definen así:

Listado 6.21: Descriptores de grupos

```
struct ext2_bg_desc {
        u32 bg_block_bitmap;
        u32 bg_inode_bitmap;
        u32 bg_inode_table;
        u16 bg_free_blocks_count;
        u16 bg_free_inodes_count;
        u16 bg_used_dirs_count;
        char pad[14];
} __attribute__ ((__packed__));
```

Como estos descriptores contienen el número de bloque donde empieza la tabla de inodos casi lo tenemos:

Listado 6.22: Localizando el inodo

```
iindex = (inum - 1) % sbpriv->inodes_per_group;

/* Lee el bloque donde está nuestro inodo
 * (usa el mismo razonamiento que para localizar el grupo): */
buf = ext2_bread (sb, e2desc->bg_inode_table
        + (iindex * sbpriv->inode_size) / sb->blocksize);

/* Se posiciona en el primer inodo del bloque leído: */
struct ext2_inode *e2ino = (struct ext2_inode *) buf->data;

/* Avanza hasta el correcto: */
e2ino += (iindex % (sb->blocksize / sbpriv->inode_size));
```

Ahora tenemos el inodo que buscábamos y aunque es tedioso, conceptualmente sólo tenemos que usar las estructuras para encontrar el bloque que contiene el dato, leerlo y colocarnos sobre el dato en ese bloque.

Como detalle, podemos ver que el número de inodos es limitado y prefijado en el momento de creación del sistema de archivos. Esta es la razón por la que un sistema de archivos puede ser inoperable a pesar de tener espacio libre. Podemos ver el uso de inodos por ejemplo con el comando: `df -i`. Es más común que un sistema de archivos se llene por espacio que por alcanzar el número de inodos pero si se alcanzara el límite de inodos, la consecuencia práctica sería que podríamos hacer los archivos más grandes pero no podríamos crear archivos nuevos[5].

Si volvemos al proceso que estábamos siguiendo para resolver "*/dir/archivo*", tenemos localizado el inodo del directorio raíz: "/", pero no es lo que buscamos por lo que habría que leer el directorio "/" y encontrar la entrada para "*dir*".

[5]Este problema ha sido observado experimentalmente en equipos en producción varias veces y suele confundir a administradores novatos.

Un directorio no es más que un archivo de tipo directorio (igualmente tenemos tipos para archivos regulares, dispositivos, enlaces, etcétera), cuyo contenido son las entradas del directorio, incluyendo "." y ".." con enlaces a si mismo y a su directorio padre respectivamente.

La función que realiza esta búsqueda para extended 2 se llama: "`ext2_lookup`" y recibe un puntero a una estructura *inode*, un nombre y una longitud del nombre (la razón de usar una longitud en lugar de terminar con un "\0" el nombre, es que así se pueden reusar las cadenas que contienen paths, tal vez se podría haber hecho que si la longitud es cero esta se obtuviera con "*strlen*" pero no se consideró necesario).

La implementación actual no soporta muchas entradas en un directorio ya que no se ha probado para el caso de que esas entradas ocupen más de un bloque aunque debería funcionar:

Listado 6.23: Buscando entradas en directorios

```
while (offset <= inode->filesize) {
        buffer = ext2_block_read (inode, block);
        if (buffer == NULL)
                break;

        dirent = buffer->data;

        while ((void *)dirent <
                buffer->data + inode->sb->blocksize) {
                if (len == dirent->name_len &&
                        strncmp (name, dirent->name, len) == 0)
                        /* Esto debería adaptarse para soportar
                         * puntos de montaje, por ejemplo:
                         * (inode->covered) ? inode->covered->sb
                         *        : inode->sb */
                        return ext2_inode_read (inode->sb,
                                dirent->inode);

                dirent = (void *)dirent + dirent->rec_len;
        }

        offset += inode->sb->blocksize;
        block++;
}
```

En *ext2* hay dos tipos de entradas de directorios: lista enlazada y formato indexado. ¡OS sólo soporta lista enlazada. El único problema que supone es de rendimiento ya que por retrocompatibilidad, en disco se mantienen ambos formatos (esta fue la razón de no soportar el formato indexado ya que al tener ambos se trata de una optimización

y no de algo obligatorio).

Ahora que hemos visto las operaciones del superbloque, pasamos a las operaciones de inodos, que dado que no soportamos escritura en disco serán básicamente dos: "*lookup*" y "*bmap*". En el futuro habrá más: crear archivo, enlazar, borrar, enlazar simbólicamente, crear y eliminar directorio, crear dispositivo, renombrar, leer enlace simbólico, seguir enlace simbólico, truncar y cambiar permisos.

La operación de *lookup* la acabamos de ver y *bmap*, como ya dijimos, transforma una combinación de inodo y offset en un número de bloque de disco. Este proceso es muy conocido en los cursos de sistemas operativos que describen los sistemas de archivos de Unix y no entraña mucha dificultad. Cada inodo contiene unos bloques de datos directamente insertados en el inodo, es fácil leer uno de esos bloques:

Listado 6.24: Leyendo un bloque directo

```
struct ext2_inode_priv *ipriv = inode->priv;

return ipriv->blocks[nblock];
```

Si siempre se usara este sistema, y como hay 12 bloques reservados para datos en el inodo, sólo podríamos acceder a 12 * blocksize elementos. En realidad aunque reserváramos espacio para 100, el límite de tamaño de archivo seguiría siendo pequeño. Lo que se hace es que el siguiente elemento (decimotercero) no contiene datos sino una lista de punteros a bloque con datos. Si usáramos un tamaño de bloque de 1KB, el elemento decimotercero indexaría los bloques 13 a 268.

Aún así el límite de tamaño no sería muy grande por lo que el elemento decimocuarto está doblemente indexado, y es un bloque que contiene punteros a bloques que contienen a su vez punteros a los bloques de datos. Ya para terminar el elemento decimoquinto contiene una triple indirección y contiene un bloque con punteros a bloques que contienen punteros a bloques que apuntan a los bloques datos.

Es un método simple pero ingenioso ya que la mayoría de archivos serán pequeños y se podrán leer directamente desde el inodo evitando por tanto una operación de lectura.

¡OS sólo implementa por ahora bloques indirectos y no se ha programado la parte que debería leer bloques doble y triplemente indirectos. El código no es complicado, sólo hay que hacer las cuentas matemáticas:

Listado 6.25: Leyendo un bloque indirecto

```
...
} else if (nblock < 12 + (inode->sb->blocksize >> 2)) {
```

```
        if (ipriv->blocks[12] == 0)
                return 0;

        block = bread (inode->sb, ipriv->blocks[12]);

        return *(u32 *)(block->data + 4 * (nblock - 12));
}
```

6.6. Caché de inodos y buffers

Acabamos de ver todo el proceso para abrir y leer un archivo pero no hemos visto como se hace en caso de que los datos y/o los inodos se encuentren en memoria (algo que ocurriría si se leyera un buffer que ya se leyó o se abriera una ruta que pase por un inodo que ya se recorrió).

Pensemos en un uso habitual de nuestro ordenador personal. Casi con toda probabilidad cargaremos muchas más veces en un espacio corto de tiempo los mismos programas, trabajaremos sobre los mismos archivos y gestionaremos los mismos datos. Por tanto y dado que son tan lentas, tiene sentido cachear las lecturas en memoria RAM y dado que los usuarios se mueven por los mismos directorios, se cachearán también los inodos.

Empíricamente podemos hacer una prueba rápida. Si en mi sistema vacío las cachés mediante el siguiente comando:

Listado 6.26: Vaciando cachés del sistema

```
sync ; sync ; sync ; echo 3 >  /proc/sys/vm/drop_caches
```

¡OS tarda en compilar un poco más de 7.5 segundos. Si repito el proceso nada más terminar, ese tiempo baja a 5 segundos. En otras palabras: si despreciamos el tiempo de acceso a memoria, de los 7.5 segundos, 2.5 fueron dedicados a leer de disco (bien el binario del compilador, bibliotecas o bien los archivos de código fuente).

Para ¡OS se ha optado por programar una caché de forma modesta. No es crítico que el algoritmo sea el más rápido puesto que se puede cambiar después y puesto que por lento que sea siempre superará a una lectura de disco.

Nada más se usa una lista enlazada de bloques o de inodos, y cada vez que se vaya a leer uno (mediante las funciones "*bread*" e "*iget*" respectivamente), se consulta primero la lista enlazada y en caso de que el número de bloque o de inodo coincida con el que buscamos (en el de inodo se necesita además el número de dispositivo puesto

que puede haber dos inodos con el mismo número en dos sistemas de archivos diferentes), se devuelve. En caso negativo se reserva memoria, se inicializa y se añade a la caché antes de devolverlo para que esté disponible en el siguiente uso.

Como ni los bloques ni los inodos se liberan nunca, no es necesario profundizar más. La idea para un futuro sería crear un hilo del kernel (algo que con la infraestructura que tenemos no es posible) que mida estadísticas y en caso necesario libere los inodos que estén al final de la lista y mueva los bloques o inodos que se usen al principio[6].

Este algoritmo se conoce con el nombre de LRU ("*least recently used*") y como decimos, las entradas que se usan se mueven al principio y las que no se usan van moviéndose de forma natural hacia el final; cuando la memoria escasee (por ejemplo porque se supera un número fijo de elementos) liberamos el último o últimos elementos. Además las consultas de caché se realizan empezando por delante porque es más probable que nos pidan algo que se usó más recientemente[7].

El hilo que realice la limpieza podrá tener en cuenta parámetros globales del sistema como porcentaje de memoria libre pero sin superar otro porcentaje de la memoria, relación con otras cachés, etcétera. Ajustar estos parámetros será la parte más complicada.

Para liberar elementos habrá que bloquear la lista y esto require sincronización. Ese tipo de problemas surgen cuando existe multiprogramación y por su naturaleza suelen ser aún más difíciles de resolver que los vistos hasta ahora.

6.7. Conclusión

Aunque la combinación de VFS y ext2 de ¡OS no se puede comparar con el que pueda tener por ejemplo Linux, conceptualmente no distan mucho y comprender como hacemos sirve para comprender como hacen otros kernels mucho más avanzados ya que nuestro diseño es conceptualmente similar.

En el documento hemos eliminado control de errores y caminos de código para situaciones límites porque lo más importante es entender como funciona conceptualmente, pero se ha recorrido con la suficiente profundidad y conocer un VFS completo no supone mucho más esfuerzo.

[6]Este es un buen ejemplo de desarrollo iterativo e incremental. Estamos dejando una característica deliberadamente sin terminar para terminarla en el momento adecuado. Hacer un sistema bueno de liberación en este momento sería muy complejo y no quedaría bien.

[7]El kernel de Linux usa un mecanismo similar sólo que contiene dos listas, una para "inodos calientes" ("*warm*") y otra para los que no se han usado habitualmente ("*cold*"). De esta forma evita cosas como que una búsqueda (inodos que podemos considerar ocasionales), haga que se eliminen de la lista inodos que si se usarán en corto espacio de tiempo.

Como el interfaz es de sobra conocido podemos consultarlo en varias obras de la bibliografía: [Bac86], [Vah95], [Lov10], [BP05], [Mau08] y [Bar01].

Capítulo 7

Carga de archivos binarios

Como ya tenemos disco y memoria podemos intentar cosas más ambiciosas.

En este capítulo vamos a ver como cargar un programa ejecutable de disco, como ejecutarlo (con los privilegios de CPU adecuados), y por último, como retomar el control cuando ese programa requiera un servicio del sistema operativo.

La mayor dificultad que implica ejecutar un programa será la interacción de varios subsistemas, el que estamos haciendo para ejecutar, otro nuevo para que el sistema operativo ofrezca los servicios, el del disco para leer y el de memoria, puesto que una CPU solamente ejecuta programas desde memoria. Además, y dado que los binarios están preparados para ejecutarse desde una dirección concreta, también veremos como mapear esa dirección y lo que se conoce como el "*address space*" del proceso (aunque podría haberse mezclado esta sección con el capítulo de la memoria, aquí tratamos la gestión de memoria para los procesos y en el otro la gestión de memoria del propio kernel). Por último, también habrá que programar el código que lee los binarios ya que estos usan un formato especial.

Dado que ejecutamos un programa hemos de desarrollar un poco más el concepto de proceso. Algo que ya definimos pero vagamente.

La manera más fiable de saber que es un proceso, es abrir el árbol del código fuente de un sistema operativo que nos lo proporcione y ver como lo define. Por ejemplo en Linux si abrimos "*include/linux/sched.h*", hacia la mitad del archivo tenemos la estructura "`task_struct`".

Esta estructura es bastante grande, pero entre los muchos campos tenemos los más importantes: estado, pila, prioridad, mapa de memoria, número de proceso, de hilo, enlaces a su proceso padre, a sus procesos hijos y a sus procesos hermanos (los que

tienen padre común), estadísticas de recursos consumidos, temporizadores activos, *ipcs* (comunicación entre procesos, del inglés “”), señales, además de bloqueos o “*locks*” (tanto para sincronización de procesos como sobre archivos) y archivos abiertos.

La estructura que define un proceso en ¡OS es mucho más modesta puesto que sabemos de antemano que tendremos solamente un proceso y porque sólo se han añadido los campos imprescindibles.

7.1. ELF

Para cargar un archivo ejecutable de disco es suficiente con que éste contenga las instrucciones que desea que se ejecuten. Por esa razón se pensó en un principio en usar directamente código objeto como formato ejecutable. Recordemos que el código objeto es una traducción literal de código en binario.

En realidad no habría habido problema alguno en un principio pero conforme quisiéramos más características si que habrían surgido inconvenientes.

Por ejemplo, si un ejecutable es un archivo binario cualquiera, ¿cómo nos aseguramos de que estamos ejecutando un archivo con código? Podríamos poner al principio del archivo una marca, pero ¿cómo sabríamos que ese archivo contiene instrucciones para nuestra arquitectura? Además no hemos de olvidarnos de la dirección de carga (aunque esto podría evitarse compilando código PIC o “*position independent code*”). Lo que podemos hacer es añadir campos para todos estos elementos.

Pero todo esto que nos surge sobre la marcha es justamente lo que nos soluciona ELF. Este formato incluye secciones para código, datos, carga de bibliotecas dinámicas y otras; elementos que en general iremos necesitando conforme avancemos. Además es un formato simple y bien documentado por lo que reinventar la rueda no nos supone ningún beneficio adicional.

La especificación de ELF puede encontrarse en [is] para archivos ELF de 32 bits y en [HP/98] para los de 64 bits, y como siempre, programar un cargador ELF es fácil teniendo buenas funciones en las capas inferiores y buena documentación.

Vamos a seguir el código (pseudocódigo en realidad), el cual por hacernos una idea general lee las cabeceras de ELF con las especificaciones de segmentos tal y como van en memoria y si son de datos, código, stack, etcétera.

Listado 7.1: Mapeando un segmento de ejecución.

```
/* Lee la cabecera del archivo */
bytes = read (fd, &elfh, sizeof (struct elf_header));
```

```
/* Regresa si no es ejecutable: */
if (elf_check_header (&elfh, ET_EXEC) == -1)
        return -1;

/* Este bucle da una vuelta por cada sección ELF */
do {
        lseek (fd, offset, SEEK_SET);

        bytes = read (fd, &elf_ph, sizeof (struct elf_ph_entry));

        switch (elf_ph.ph_type) {
                case PT_LOAD:
                        {
                        s8 ret;
                        ret = elf_load_section (fd, &elf_ph);
                        }
                        break;
                case PT_DYNAMYC:
                        /* Código para bibliotecas dinámicas */
                        break;
                case .. aquí otras secciones
        }

        offset += elfh.e_phentsize;
} while (más_secciones);
```

La función lee del binario que queremos ejecutar tantos bytes como ocupe la cabecera y en cada vuelta localiza la siguiente sección hasta que ya no queden más. En caso de ser una sección con código ejecutable usa la famosa función "*mmap*" para mapear la memoria. Como no recorremos secciones de datos no podremos ejecutar binarios que contengan datos. En todo caso con poder ejecutar por ahora un binario con cualquier limitación, nos sirve para seguir avanzando y cuando funcione pensaremos en darle más funcionalidad.

7.2. mmap

A partir de este punto ¡OS divergirá de cualquier otro UNIX ya que nos falta mucho para tener infraestructura que nos permita construir una función "*mmap*" completa.

Un Unix cualquiera crearía un segmento que cubriría la memoria asociada al segmento ELF ejecutable. Además marcaría las páginas como asociadas al segmento, pero no presentes. Cuando se intentara ejecutar el binario saltaría una excepción de fallo de

página y el manejador lo resolvería viendo que las páginas de esa dirección de memoria no están presentes pero que están respaldadas ("*backed up*") por un archivo, y que tal como indica la definición del segmento, lo que las respalda es código. Se leería el archivo, se copiarían unos bytes (con una página sería suficiente pero como optimización es común copiar varias) y el proceso seguiría ejecutándose como si no hubiera ocurrido nada.

¡OS simplifica el código dando por supuestas varias cosas. Primero suponemos que mmap sólo se usa para mapeos de archivos y no para mapeos anónimos. También que el código cabe en una página (algo no difícil dado que tenemos páginas de 2MB), y en general podemos suponer cualquier cosa porque esta función sólo se usará para esta tarea por ahora. No debería ser complicado ajustarla para que se comportara como el resto de Unix ya que sólo tendríamos que quitarle elementos y moverlos a los sitios correspondientes para que se ejecutaran en el momento oportuno.

El código de la función es pequeño y podríamos reproducirlo entero, pero en lugar de eso y por simplicidad, vamos a verla con pseudocódigo.

Listado 7.2: Función mmap

```
void *
mmap (void *addr, size_t length, int prot,
        int flags, int fd, off_t offset)
{
        si addr ya está mapeada: regresa.

        si length es mayor que una página: regresa.

        /* Nótese que en lugar de pedir una página al KMA,
         * se usa la dirección física de forma lineal.
         * Es algo temporal que hace que sólo podamos tener un
         * proceso en toda la vida del kernel, además de que un
         * ELF malicioso podría hacer que sobreescribiéramos
         * memoria del kernel o cualquier otra cosa.
         */
        mapea la página física addr en addr.

        copia leyendo de fd, length bytes en addr.
}
```

Teniendo en cuenta toda la funcionalidad que esconde esta función el resultado es bastante minimalista. A continuación vemos como se hace cada uno de los pasos.

7.3. Espacio virtual de un proceso

En nuestra terminal Unix favorita podemos ver el espacio de un proceso sobre el que tengamos permisos con el comando "*pmap*" (si estamos en Linux el comando muestra la misma información que el archivo `/proc/<pid>/maps` sólo que un poco más ordenado y de forma más portable).

Esta es parte de la salida del comando para un binario estático preparado para la ocasión (un comando dinámico será similar sólo que añade mapeos para biblitecas) en un equipo de 64 bits:

Listado 7.3: Salida de pmap

```
00400000-005a8000 r-xp 00000000 08:06 5908 static
011c5000-011e8000 rw-p 00000000 00:00 0 [heap]
7fff2986d000-7fff2988e000 rw-p 00000000 00:00 0 [stack]
```

La primera columna (las columnas se separan por espacios) muestra intervalos de direcciones, en este caso hemos mostrado tres segmentos mapeados. La siguiente columna muestra los permisos del mapeo, "p" significa "privado" que quiere decir que si otro proceso lo tuviera mapeado también y lo modificase, este proceso no vería los cambios porque se usa "*copy on write*" (COW). La siguiente columna es el offset sobre el archivo, la que la sigue muestra el dispositivo en formato major:minor y la penúltima es el número de inodo. Para terminar se representa el path del ejecutable o su función entre corchetes si el segmento no está respaldado con ningún archivo.

Volviendo a la ejecución del binario ELF, la función mmap miraba si la dirección ya estaba mapeada. Este proceso será de bajo nivel y consistirá en recorrer las tablas de páginas de la CPU (recordemos: para este proceso), y ver que dirección física está mapeada en la virtual que se indica. Quizás esta comprobación se podría omitir, en cuyo caso podría darse el caso de que nos pidieran mapear algo en un segmento ocupado. Es una decisión difícil puesto que no tenemos "`munmap`", en todo caso, con todo lo que hemos hecho hasta ahora, es un camino que no se dará.

Esta función que traduce se llamará "`virt2phys`", y usa funciones "*walk*" (tales como "`pml4_walk`", "`pdpe_walk`" y "`pde_walk`"), las cuales son "*oneliners*" (se refiere a funciones o comandos de una linea) puesto que solamente recorren una tabla.

La función que mapea una página de memoria se llamará no sorprendentemente "`map_one_page`" y será similar a "`virt2phys`", sólo que en lugar de implementar la lectura, implementa la asignación. Su función es recorrer la tabla de páginas, y en caso de que no haya un mapeo a cualquiera de los niveles, pide memoria libre y mapea el

siguiente nivel de memoria. Cuando llega al último, ya sólamente tiene que rellenar la dirección base de la página y ésta queda mapeada.

Este es el proceso. Hemos de ser cuidadosos puesto que estamos trabajando con las delicadas estructuras de la CPU:

Listado 7.4: Función map_one_page

```
if (!is_canonical (vaddr) || !is_canonical (phys))
        return 0;

pdpe = pml4_walk (vaddr);
if (pdpe == NULL)
        pdpe = pte_set (current->mm->pgd +
                pml4_offset (vaddr), pte_get_empty (), 1, 0);

pde = pdpe_walk (pdpe, vaddr);
if (pde == NULL)
        pde = pte_set (pdpe + pdpe_offset (vaddr),
                pte_get_empty (), 1, 0);

/* Se necesita: __va() porque pte_set devuelve una dirección
 * virtual en espacio de kernel y la CPU trabaja con físicas. */
pte_set (pde + pde_offset (vaddr), __va (phys & PAGE_ALIGN), 1, 1);
```

7.4. Ejecución

En esta sección vemos la función "exec", la cual tampoco nos supondrá mucho trabajo dadas las simplificaciones que estamos haciendo. Básicamente tendrá dos lineas que podrían incluso juntarse en una si no fuera por la comprobación de errores (no incluida):

Listado 7.5: Función exec

```
entry = elf_map (path);
usermode_jump (entry);
```

Todos los elementos vistos en las secciones anteriores del capítulo describen la primera función y la segunda la describimos en la siguiente sección.

7.5. Paso a userspace

El código anterior nos dejó la sección de código del binario en memoria, y como acabamos de ver, "`elf_map`" nos devuelve el "*entry point*". El siguiente paso es hacer que el registro "*rip*" salte a esa dirección, con la única complicación de cambiar el modo de privilegio de la CPU y de proporcionar alguna forma para que podamos recuperarlo después.

Los métodos de paso a *userspace* son variados y diversos. La forma clásica de hacerlo (y más compatible) es usar la instrucción "`iret`" con un stack que habremos preparado previamente. Este sistema es similar a como funcionan las interrupciones (salvo por el cambio de privilegio.

Si tenemos en cuenta que habrá que pasar entre modo usuario y kernel para cada servicio que el proceso de usuario pida al sistema operativo, será una operación que se repetirá casi constantemente y sobre la cual es interesante optimizar. Por esta razón las CPUs nuevas tienen instrucciones dedicadas a esta tarea, las cuales dan por hecho cosas que "`iret`" no puede dar puesto que es una instrucción más genérica pensada para regresar de interrupción y por tanto, son más rápidas.

Como para casi todo, los fabricantes no se pusieron de acuerdo y existen dos implementaciones: *syscall/sysret* y *sysenter/sysexit*, la primera implementada por AMD y la segunda por Intel. Como la versión de AMD es más compatible (la arquitectura x86-64 fue diseñada por AMD), esta es la opción elegida por ¡OS y la única discutida en este trabajo. Como siempre tenemos los manuales de las CPUs para más detalles.

Pasar a espacio de usuario implica rellenar dos registros MSR (como vimos, son unos registros especiales de la CPU), uno con el par de selectores de segmento que usamos para movernos a espacio de usuario y después a espacio de kernel, y el otro con la dirección de memoria del procedimiento que se llamará cuando pasemos a modo kernel. Además habrá que copiar la dirección a la que queremos saltar a un registro de la CPU y llamar a la instrucción. Este es el código implementado en realidad como una macro[1]:

Listado 7.6: Salto a usermode

```
#define usermode_jump(_addr)                                    \
        do {                                                    \
                msr_write (MSR_STAR, ((u64)U_CS)<<48            \
```

[1]La razón de poner en la macro un bucle con una condición falsa, es que imaginemos que la macro se usase en una estructura condicional (un "*if*"). En caso de no poner el bucle, el programador que la use podría escribir literalmente: "`if (cond) usermode_jump(addr); else ...`" y este código nos daría error al compilar (o lo que es peor, podría modificar la lógica del programa) ya que no se usaron llaves para delimitar el bloque. Usando el "`do { ...} while (0)`", esto nunca podría ocurrir.

```
                                | ((u64)K_CS)<<32 );          \
        msr_write (MSR_LSTAR, (u64)syscall_dispatch);    \
        asm volatile ("mov %0, %%rcx\n\t"                \
                "sysretq\n\t" : : "m"(_addr));           \
} while (0)
```

En este punto ya casi podemos decir que tenemos un sistema operativo puesto que somos capaces de ejecutar al menos un programa. Esta afirmación dista bastante de la realidad ya que este programa no puede comunicarse con el exterior ni hacer ninguna entrada o salida, solamente puede hacer procesamiento. Esto no es muy útil si no hay datos que procesar y poner esos datos en el propio ELF sería impracticable (no podemos pedir al usuario que recompile cada vez que quiera cambiar un parámetro de entrada a un programa) además de que no podríamos extraer los resultados después (como se puede comprobar, aún estamos en proceso de desarrollo ya que ni siquiera podríamos escribir esos datos en el disco).

La macro que salta a espacio de usuario instalaba una dirección para la vuelta a modo kernel. Esa dirección queda registrada en el procesador y la instrucción "*syscall*" sabe donde saltar cuando el programa de usuario requiera algún servicio del sistema operativo.

Por tanto podemos construir un ELF muy simple que nos pida algún servicio. Si observamos la tabla de llamadas al sistema de Linux, podemos intentar ofrecer compatibilidad y como prueba de concepto podemos implementar la llamada al sistema "*write*", la cual escribe en un descriptor de archivo. En los sistemas Unix tenemos tres descriptores de archivo estándar (0, 1 y 2) por lo que podemos intentar escribir en la salida estándar (descriptor 1) una cadena de caracteres.

Dado que el compilador de C genera demasiado código superfluo y dado que añade cosas que ¡OS no soporta, implementaremos este programa, por ahora, en ensamblador. Este código se compila con el comando "*as*", seguido del comando "*ld*":

Listado 7.7: Salto a usermode

```
.section .text
.globl _start
_start:
        movq $1, %rax;   // write( ...
        movq $1, %rdi;   // write(1, ...
        movq $str, %rsi; // write(1, str, ...
        movq $14, %rdx;  // write(1, str, 14);
        syscall
        movq $60, %rax;  // exit( ...
        movq $0, %rdi;   // exit(0);
```

```
        syscall
loop:
        jmp loop
        str:    .asciz "Hola mundo!\n$ "
```

En los comentarios se pone el codigo equivalente en C y dado que el codigo en C es más expresivo, vemos que se necesitan varias lineas de ensamblador para cada linea de C.

El código nada más llama a la función write pasándole el descriptor en el que quiere imprimir, la dirección de la cadena (la cual se ha definido en la sección de código, como acabamos de decir, aún no podemos mapear sección de datos) y el número de caracteres que se quieren imprimir. Por último llama a la función "exit" para salir del programa.

7.6. Llamadas al sistema

Veamos que ocurre cuando la aplicación de usuario ejecuta una instrucción "*syscall*" después de haber puesto los parámetros de la llamada al sistema en los registros adecuados.

La CPU mirará el registro MSR para saber la dirección a la que saltar, hace un cambio de privilegio usando el descriptor de segmento que se le pasó en el registro MSR_STAR y tal como indicamos ese salto se hará a la función "`syscall_dispatch`".

En realidad ¡OS no implementa "`syscall_dispatch`" como una función sino como una macro inicializada a la dirección de `syscall_dispatch_real + 16`. Este mecanismo es similar al usado en las interrupciones para permitir programar los manejadores en C sin un paso por un procedimiento en ensamblador.

De todas formas, aunque "*inline*", esta vez no nos libramos de usar ensamblador para casi toda la función:

Listado 7.8: Manejador de syscalls

```
/* Este código usa sintáxis 'inline' normal ya que no hay
 * restricciones ('constraints') al final. Por eso el nombre
 * del registro se prefija de un único símbolo '%'. */
asm volatile (
        "nop; nop; nop; nop;"
        "nop; nop; nop; nop;"
        "nop; nop; nop; nop;"
        "nop; nop; nop; nop;\n\t"
        /* Salta a la etiqueta si el número de syscall fuera
         * mayor que el número de syscalls: */
```

```
        "cmpq $" stringify (__NR_syscall_num) ", %rax\n\t"
        "jae 1f\n\t"
        /* Salva el contexto */
        pushaq()
);

/* Este código no puede juntarse con el anterior porque GCC
 * rellena los registros antes que el código ensamblador.
 *
 * En este caso se han usado restricciones por lo tanto
 * el nombre de los registros se prefija de dos símbolos '%%'.
 * Esto se conoce como 'formato extendido'. */
asm volatile (
        /* Cada puntero ocupa 8 bytes por lo que el offset
         * en la table será: número de syscall << 3 */
        "shl $3, %%rax\n\t"
        "addq %0, %%rax\n\t"
        "call *(%%rax)\n\t"
        : : "c"(syscall_table)
);

/* Esta parte tampoco puede juntarse con la anterior porque
 * popaq() usa %rax ya que no tiene restricciones. */
asm volatile (
        popaq()
        "1:\n\t"
        "sysretq\n\t"
);
```

El código está comentado. Los *nops* se usan para saltar el prólogo de GCC, luego se comprueba que el número de llamada al sistema no es mayor que el número de *syscalls* en la tabla y se guardan los registros para que posteriormente podamos dejarlos como estaban. En el futuro habrá que revisar esto ya que será posiblemente necesario modificar parámetros por ejemplo para devolver un dato.

Después se calcula el desplazamiento en la tabla y se llama a la dirección de memoria que contenga. No será necesario comprobar que el puntero sea NULL por ahora, puesto que no habrá llamadas al sistema no definidas (sólo se ha definido una llamada al sistema y comprobamos que el número que nos pasan no sea mayor que el número de elementos, por tanto no habrá huecos).

Una vez que la función se ha ejecutado, se restaura el contexto (los registros y el estado de la pila) y se regresa a modo de usuario.

Por ahora no tiene mayor complicación salvo que nos falta por ver como registra el resto de subsistemas del kernel las llamadas al sistema. Para ello usaremos un método

muy parecido al que se usó para las interrupciones. Definiremos una función que copie la dirección de la función que se le pasa en el elemento correspondiente de la tabla:

Listado 7.9: Tabla y registro de syscalls.

```
static void *syscall_table[__NR_syscall_num];

s8
syscall_register (u16 num, void *handler)
{
        if (num >= __NR_syscall_num)
                return -1;

        syscall_table[num] = handler;

        return 0;
}
```

Para la función que se llamará, proporcionamos un método totalmente transparente. Se define una función con los parámetros adecuados y estos quedan definidos sin más puesto que las aplicaciones de usuario pasan los parámetros en los registros en el mismo orden que GCC.

La única limitación que esto supone, es que en x86-64 solamente se pueden pasar 6 parámetros ya que no hay más registros disponibles según el ABI para esta labor.

Para completar esta es la implementación de la llamada al sistema que imprime por pantalla:

Listado 7.10: Llamada al sistema que imprime

```
s64
sys_write (int fd, const void *buf, size_t count)
{
        if (fd < 1 || fd > 2)
                return 0;

        kprintf (buf);

        return 0;
}
```

Esta función realiza perfectamente su labor pero presenta al menos cuatro problemas con los que tendremos que vivir por ahora.

Solamente funciona para STDOUT y STDERR (de ahí la comprobación de número de descriptor) y no permite escribir en archivos. Es peligroso seguir un puntero en modo

de usuario (durante la programación de este kernel se ha comprobado lo exigente que es la CPU y lo fácil que es obtener un fallo de protección general). "`kprintf`" usa formatos y requiere una cadena terminada con el carácter "\0" y no estamos haciendo estas comprobación dado que (y este es el cuarto problema) no tenemos la terminal programada y cuando la tengamos, no se usará *kprintf* directamente puesto que cada terminal podrá interpretar los caracteres de una manera diferente.

Capítulo 8

Pasos futuros

En la bibliografía se incluyen títulos sobre elementos que no se han implementado, algo que unido a la experiencia adquirida, nos permite comentar sobre las posibilidades de continuación del trabajo, así como hacer una pequeña descripción del posible enfoque a la hora de desarrollar el resto de subsistemas.

Con todo el esfuerzo realizado no estamos muy lejos de tener un intérprete de comandos o *shell* que permita ejecutar programas secuencialmente (monotarea), ya que a grandes rasgos sólo haría falta ir implementando llamadas al sistema, muchas de las cuales ya casi tenemos y sólo necesitaríamos una función que enlazase el espacio de usuario con una función del kernel. Otras sin embargo requerirán más trabajo pero nada fuera de nuestro alcance puesto que con la base disponible la dificultad de programación disminuye considerablemente.

Hasta ahora hemos sentado unas bases que un programa común de espacio de usuario ya tiene, y por tanto el 80-90 % del esfuerzo (y un porcentaje mayor de la frustración) consistía en saber como funcionan las cosas[1]. A partir de hora ya sabemos como funciona un PC (salvo por pequeñas partes de bajo nivel que no se han implementado aún) y podemos tomar decisiones de diseño con mayor libertad. El proceso por tanto continua como un desarrollo del software más convencional.

Los puntos que siguen se detallan en el orden subjetivo en el que el autor seguiría el trabajo, aunque quizás durante el avance se descubriría que no es el más óptimo y por tanto no se trata de un orden inamovible.

[1] A pesar de presentar un trabajo de 9500 lineas de código, la programación representa un pequeño porcentaje del tiempo invertido ya que la mayoría se ocupó en aprender las tecnologías y en detalles de bajo nivel tales como estabilizar el código y acertar con las sutilezas del hardware

8.1. Infraestructura para los binarios

En los binarios no hemos preparado un *stack* ni el segmento de datos. Esta sería probablemente la tarea de mayor prioridad. Además sería interesante hacer un "hola mundo" en C (un binario común salvo por ser estático, el dinámico puede dejarse para más adelante ya que necesitaría más infraestructura) e intentar que funcionara sin las artimañas que usamos para construir nuestro ejecutable.

Un ejecutable compilado con GCC nos presenta un reto mayor ya que incluye bastantes cosas. Por ejemplo, la mayoría de los binarios en mi sistema Linux en el directorio */bin* tienen 27 secciones ELF, y si ejecutamos: `readelf -a` sobre un ejecutable, veremos que tiene bastantes cosas más que aún no son familiares.

Es posible que además tuviéramos que tocar los flags de compilación de GCC al compilar mientras no soportemos todas las características necesarias para ejecutar un simple binario estático. Otra tarea interesante sería hacer que se ejecutara un programa que sólo ha sido compilado con `-static`.

Mediante "*objdump*" o "*readelf*" podemos ver elementos que son propios de GCC o del sistema con el que trabajamos, pero para saber que son secciones estándares como GOT, PLT, constructores y destructores ("*ctors*" y "*dtors*") y otras, el excelente recurso [Lev99] incluye explicaciones sobre todos estos elementos.

8.2. Terminal y descriptores estándar

El siguiente paso podría ser hacer una llamada al sistema para imprimir en pantalla, que no sea tan *ad hoc*, y para ello se necesita la terminal.

Tendremos que hacer un dispositivo de carácter que la represente ("`/dev/tty`" por ejemplo) y hacerlo interactuar con los drivers de video y de teclado. También tendremos que tener en cuenta los distintos modos, disciplinas, capacidades, etcétera, y que puede haber varias simultáneamente. Podemos hacernos una idea general con el siguiente recurso: `http://en.wikipedia.org/wiki/POSIX_terminal_interface`.

Además habremos de implementar dispositivos de carácter para los descriptores estándar: STDIN, STDOUT y STDERR.

En ausencia de presión por tener algo más tangible, la terminal sería el siguiente paso justo después de tener los binarios de GCC porque aunque no es algo muy vistoso (salvo por permitir al usuario cambiar de terminal), nos ayudaría a introducir mejoras en todos los elementos implicados y nos hará más fácil seguir creciendo después.

8.3. mmap mejorada

Ya comentamos las posibles mejoras sobre los mapeos de memoria y *mmap*: hacerla asíncrona, implementar regiones anónimas, copy on write, programar el manejador de fallos de páginas, etcétera.

Nos quedamos cortos porque *mmap* es una función sobre la que basar todo el sistema de memoria. Si lo pensamos tiene sentido puesto que cualquier reserva de memoria es un mapeo, esto es cierto incluso para lecturas de disco, memoria compartida entre procesos, etcétera.

Sería interesante que en lugar de que la caché de buffers se implemente sobre regiones de memoria pedidas con *malloc*, lo hiciera sobre páginas mapeadas (aunque habrá que pensarlo bien ya que al tener páginas de 2MB, la implementación diferirá del resto de implementaciones de Unix). También podríamos hacer algo parecido a lo que tenemos en el VFS: un diseño orientado a objetos con distintas operaciones según cual sea el subsistema que hay detrás de la página.

De nuevo serían cambios no muy visibles pero imprescindibles de cara a poder implementar varios procesos.

8.4. Hilos

Tanto la caché de buffers como la caché de inodos no se están liberando. Sería ideal crear otros hilos de ejecución que las liberaran. Además los hilos nos serían útiles para implementar otras tareas de mantenimiento del kernel y para empezar la multiprogramación.

Un hilo del kernel difiere de un proceso en cuanto a que se ejecuta usando la tabla de páginas del proceso activo. Por tanto sería relativamente fácil y dado que la multiprogramación es tan compleja, conviene ir detectando los recursos compartidos para introducir bloqueos.

No hay que confundir estos hilos con los "procesos ligeros" (del inglés LWP, o "*lightweight processes*") ya que en tal caso se trata de tener varios contextos de ejecución ("*call stacks*") simultáneos en un mismo proceso de usuario (algo de menos prioridad).

También podríamos ir haciendo todo lo que nos facilite después el camino hacia la multitarea. Si pensamos en términos prácticos, cuando un proceso de usuario se estuviera ejecutando, llegaría una interrupción de un temporizador, el kernel tendría

que salvar el *stack* para cambiar al suyo y ejecutaría el procedimiento asociado a ese temporizador. Si tuviéramos un planificador de tareas se llamaría para ver si interesa ejecutar otra cosa.

La parte más complicada sería guardar y restaurar el proceso (algo para lo que existe apoyo opcional por hardware), pero tampoco debería ser algo tan complejo ya que es similar a las interrupciones o llamadas al sistema.

Aprovechamos para comentar que el planificador de tareas sólo selecciona el siguiente proceso a ejecutar para que al hacer el cambio de contexto al proceso de usuario (al regresar del kernel), se ejecute el proceso seleccionado.

Este planificador de tareas sería una función parcialmente implementada en ensamblador, capaz de salvar el estado de los registros de un proceso, su *stack*, etcétera. y de restaurar otro contexto salvado previamente. Algunos kernels usan lo que se conoce como "*preemption*", que consiste en que también pueden hacer este cambio aunque el otro proceso se encuentre en modo kernel. Esto se hace para reducir la latencia ya que si no se pudiera interrumpir, habría que esperar a que el proceso volviera a modo de usuario (o alcanzara una porción de código "segura" como para hacer el cambio en ese punto).

Lo normal es que al principio usáramos un bloqueo para que sólo hubiera un proceso en modo kernel de forma simultánea. Esto simplifica mucho las cosas ya que el código del kernel no necesita ser "*reentrant*" (*reentrant* quiere decir que puede ser ejecutados simultáneamente por otro hilo de ejecución, bien porque se interrumpe o porque tenemos dos CPUs y ambas ejecutan el mismo código). La importancia de esta carencia es directamente proporcional al número de CPUs instaladas en el sistema.

8.5. DMA y planificador de E/S

El método de E/S usado sobre los discos IDE es de las partes más carentes y menos optimizadas. Lanzar una petición y esperar a que complete para lanzar las siguientes es una ruina de cara al rendimiento.

La CPU opera en nanosegundos y los discos en milisegundos por lo que dejando a un lado las cantidades, si una instrucción se ejecuta en un nanosegundo y una lectura se hace en un milisegundo, podremos ejecutar un millón de instrucciones de procesador por cada lectura de disco (en realidad los números variarán ya que hay instrucciones que tardan más o menos y lecturas que pueden necesitar movimiento de cabeza, pero estarán más o menos en el orden de magnitud indicado). Con discos SSDs las operaciones se hacen en microsegundos pero aún así sigue siendo un tiempo que la CPU puede

emplear para cualquier otra cosa.

La solución óptima es, en caso de lectura, bloquear al proceso hasta que llegue una interrupción indicando que el dato está disponible y en caso de escritura habrá que devolver el control al proceso y diferir la escritura para que se haga en el momento adecuado (probablemente mediante otro hilo).

Estos procesos no se hacen inmediatamente sino que las operaciones se encolan y el planificador de IO (del inglés "*IO scheduler*") las realiza en el orden que considere adecuado. Otra razón para diferir las escrituras es que en caso de ser lectura secuencial, la siguiente operación se puede fusionar con la anterior y ambas se pueden lanzar como una única operación de DMA.

Según lo dicho podemos deducir que implementar algún modo de DMA sería también interesante.

8.6. Bibliotecas dinámicas

Otro paso interesante sería dar soporte a las bibliotecas dinámicas.

Esta funcionalidad no es tan necesaria puesto que dejando a un lado el rendimiento podemos generar binarios estáticos, pero al igual que con otros elementos nos enseñaría como se mapean las bibliotecas y nos ayudaría a descubrir carencias en el código que maneja los segmentos de memoria virtual, ya que las bibliotecas se mapean en varios procesos de forma simultánea y las tablas de páginas se comparten.

8.7. Otros

Como podemos ver, todo lo que fantaseamos con implementar (salvo ejecutar binarios compilados con GCC, por eso es la tarea de mayor prioridad) se hace de cara a poder añadir más llamadas al sistema.

Por ejemplo, la terminal nos permitiría añadir control de flujo, "*controlling terminal*" y grupos de procesos ("*process groups*") así como la familia de funciones que operan con la estructura "*termios*". *mmap* podría permitir que programáramos la llamada al sistema "*brk*", la cual se necesita para implementar "*malloc*" en espacio de usuario.

El resto sería necesario de cara a tener procesos (llamadas "*fork*" y "*exec*"). Tan sólo DMA y el planificador de E/S se salen de este objetivo puesto que son optimizaciones;

sólo se considera prioritario por el rendimiento que supondría.

Otros proyectos interesantes podrían ser añadir escritura a la capa de VFS y a *Extended 2*, programar el primer IPC: las señales y hacer o portar una biblioteca de C, así como otras herramientas básicas (shell, comandos de copia, listado de archivos, *grep*, etcétera).

Lo que hiciéramos a partir de aquí dependería de las preferencias personales. Se podría hacer un sistema de archivos propio, una pila *tcp/ip* junto con un *driver* de red, una versión del sistema operativo para ARM (para poder funcionar por ejemplo en un teléfono móvil) o a un hypervisor (como XEN, esto sería especialmente útil puesto que como es imposible soportar todo el hardware existente en el mercado, soportando el hardware de XEN podríamos hacer que el sistema operativo funcionase de forma virtual en cualquier equipo), o un sistema de módulos.

También sería inmensamente instructivo programar una CPU (puede hacerse en C para uno de los emuladores o en VHDL y ejecutarse bajo un simulador o directamente sobre un FPGA) y portar ¡OS a dicha arquitectura; de esta forma podríamos innovar realmente ya que podríamos cambiar las reglas del juego y mover piezas desde el sistema operativo al procesador y viceversa.

Capítulo 9

Conclusión

Un sistema operativo es uno de los programas más complejos que podemos construir pero con el suficiente esfuerzo, paciencia y ganas se va haciendo.

Siempre que se mantenga el orden entre las capas es fácil ir sustituyendo cada parte por piezas más complicadas y funcionales. A veces, no obstante, hay que dividir una capa o rediseñar varias y esto es más complicado, pero nos podemos apoyar en la experiencia de los miles de programadores que han evolucionado UNIX desde que se inventó hace más de 40 años. Intentar construir un sistema operativo moderno sin esa experiencia previa, sería probablemente un proyecto condenado al fracaso o extremadamente caro.

Desde un año antes de empezar este trabajo (en realidad habría que sumar varios años más de experiencia) fue necesario empezar a estudiar elementos que dan una base porque tan sólo enlazar un binario y hacer que se ejecute en el arranque requiere de un esfuerzo considerable como hemos visto. Pero es un esfuerzo que merece la pena porque después de tener esa base los manuales técnicos cobran más sentido, el código de otras personas se entiende mejor y los libros avanzados sobre núcleos dejan de parecer tan cifrados.

El objetivo inicial de este proyecto era comprender los núcleos modernos de cara a realizar labores de optimización en servidores en producción. Este objetivo se ha cumplido tan sólo parcialmente ya que conociendo los subsistemas es más sencillo analizar la salida de herramientas que miden el rendimiento, pero para realizar una completa optimización de un kernel es necesario comprender los algoritmos de las implementaciones que se están usando, las interacciones, los tipos de carga y sobre todo, tener la experiencia de haberlo hecho durante años.

En el desarrollo se ha priorizado avanzar vs. implementar cosas con demasiada precisión debido a la limitación de recursos (tiempo disponible y ambición del proyecto). Igualmente se han preferido sistemas modernos a clásicos (por ejemplo, APIC vs. PIC) y además haber optado por 64 bits es un avance ya que pocos sistemas operativos han sido programados desde el principio sobre una CPU de 64 bits y con un tamaño de página de 2MB.

Como conclusión personal, espero que este esfuerzo sirva para alcanzar un mayor desarrollo profesional y quizás para entrar, en caso de que fuera necesario, en el desarrollo de un kernel usado en sistemas en producción. También podría servir para que un programador novato empezara en el mundo de la programación de sistemas operativos.

Índice alfabético

Bibliografía

[AMDa] AMD. *AMD Architecture Programmer's Manual Volume 1: Application Programming*. http://developer.amd.com/documentation/guides/pages/default.aspx#manuals.

[AMDb] AMD. *AMD Architecture Programmer's Manual Volume 2: System Programming*. http://developer.amd.com/documentation/guides/pages/default.aspx#manuals.

[AMDc] AMD. *AMD Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. http://developer.amd.com/documentation/guides/pages/default.aspx#manuals.

[AMDd] AMD. *AMD x86-64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions*. http://developer.amd.com/documentation/guides/pages/default.aspx#manuals.

[AMDe] AMD. *AMD x86-64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions*. http://developer.amd.com/documentation/guides/pages/default.aspx#manuals.

[ANSI96a] Inc. American National Standards Institute. *Information Technology - AT Attachment Interface with extensions (ATA-2) Revision 4c*, 1996.

[ANSI96b] Inc. American National Standards Institute. *Information Technology - AT Attachment with packet interface - 6 (ATA/ATAPI-6)*, 1996.

[Bac86] Maurice J. Bach. *The design of the UNIX operating system*. Prentice-Hall, 1986.

[Bar01] Moshe Bar. *Linux File Systems*. McGraw-Hill Companies, 2001.

[Blu05] Richard Blum. *Professional Assembly Language*. Wrox, 2005.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. *Usenix*, 1994. http://http://www.usenix.org/publications/library/proceedings/bos94/full_papers/bonwick.a.

[BP05] Daniel P. Bovet and Marco Cesati Ph.D. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 2005.

[CC03] Pramode C.E. and Gopakumar C.E. *The linux kernel 0.01 commentary*, 2003. http://ranger.uta.edu/~dliu/courses/os/.

[Chi07] David Chisnall. *The definitive guide to the xen hypervisor*. Prentice Hall, 2007.

[cop] Intel coporation. Using the rdtsc instruction for performance monitoring.

[cora] Intel corporation. *82091AA Advanced Integrated Peripherial (AIP)*. http://www.intel.com/design/archives/periphrl/docs/29048603.htm.

[corb] Intel corporation. *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC)*. http://www.intel.com/design/chipsets/datashts/290566.htm.

[corc] Intel corporation. *8237A High performance programmable dma controller (8237A-5)*.

[cord] Intel corporation. *8259A Programmable Interrupt Controller (8259A/8259A-2)*.

[core] Intel corporation. *82c54 CHMOS Programmable Interval Timer*.

[corf] Intel corporation. *Intel Processor Identification and the CPUID Instruction*. http://www.intel.com/content/www/us/en/processors/processor-identification-cpuid-instruction-note.html.

[corg] Intel corporation. *Intel(R) 64 and IA-32 Architecture Software Developer's Manual, Volume 1: Basic Architecture*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[corh] Intel corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[cori] Intel corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[corj] Intel corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[cork] Intel corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[corl] Intel corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[cor97] Intel corporation. *Intel Multiprocessor Specification*, 1997. http://www.intel.com/design/pentium/datashts/242016.htm.

[cor04] Intel corporation. *IA-PC HPET (High Precision Event Timers) Specification 1.0a*, 2004.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, 2005.

[CT] Steve Chamberlain and Ian Lance Taylor. *Using LD*. http://sourceware.org/binutils/docs/ld/.

[DMTF08] Inc. Distributed Management Task Force. *System Management BIOS (SMBIOS) Reference Specification DSP0134. 2.6*, 2008. http://www.dmtf.org/standards/smbios.

[ea] Bryan Ford et al. *Multiboot Specification*. https://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[ea09] Hewlett Packard Corporation et al. *Advanced Configuration and Power Interface Specification Revision 4.0*, 2009. `http://www.acpi.info/spec.htm`.

[EFf] Dean Elsner, Jay Fenlason, and friends. *Using AS*. `http://sourceware.org/binutils/docs/as/`.

[Gor08] Mel Gorman. Understanding the linux ® virtual memory manager, 2008.

[Hei] Kris Heidenstrom. Faq / application notes: Timing on the pc family under dos. `http://www.sat.dundee.ac.uk/~psc/pctim003.txt`.

[HP/98] HP/Intel. *ELF-64 Object File Format 1.5 Draft 2*, 1998.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.

[is] Tool interface standards. *Executable and linkable format (ELF) 1.1*.

[KD03] Helmut Kopka and Patrick W. Daly. *Guide to LaTeX (4th Edition)*. Addison-Wesley Professional, 2003.

[Ker10] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd edition*. Prentice-Hall, 1988.

[Lev99] Levine. *Linkers and Loaders*. Morgan Kaufmann, 1999. `http://www.iecc.com/linker/`.

[Lio77] John Lions. *A commentary on the sixth edition unix operating system*, 1977. `http://www.lemis.com/grog/Documentation/Lions/`.

[LMK+89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, Quarterm, and Samuel Leffler. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1989.

[Loe09] Jon Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, 2009.

[Lov05] Robert Love. *Linux Kernel Development (2nd Edition)*. Novell Press, 2005.

[Lov10] Robert Love. *Linux Kernel Development (3rd Edition)*. Addison-Wesley Professional, 2010.

[Mau08] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox, 2008.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[Mes95] Hans-Peter Messmer. *The indispensable Pentium book*. Addison Wesley, 1995.

[MHJM] Michael Matz, Jan Hubička2, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface. AMD64 Architecture Processor Supplement.* `http://www.x86-64.org/documentation/abi.pdf`.

[MM06] Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd Edition)*. Prentice Hall, 2006.

[MMG06] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris performance and tools: DTrace and MDB techniques for Solaris 10 and OpenSolaris*. Prentice Hall, 2006.

[MNN04] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2004.

[Mol] James Molloy. `http://www.jamesmolloy.co.uk/`.

[MS96] Inc. Mindshare and Tom Shanley. *Protected Mode Software Architecture*. Addison-Wesley Professional, 1996.

[OsD] Osdev.org. `http://wiki.osdev.org/`.

[Pat03] Steve D. Pate. *UNIX Filesystems: Evolution, Design, and Implementation*. Wiley, 2003.

[PH08] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann, 2008.

[Poi] Dave Poirier. *The Second Extended File System. Internal Layout.* `http://www.nongnu.org/ext2-doc/ext2.html`.

[PS02] PCI-SIG. *PCI Local Bus Specification Revision 3.0*, 2002. `http://www.pcisig.com/specifications/conventional`.

[RFS05] Claudia Salzberg Rodriguez, Gordon Fischer, and Steven Smolski. *The Linux Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures*. Prentice Hall, 2005.

[RHL08] Arnold Robbins, Elbert Hannah, and Linda Lamb. *Learning the vi and Vim Editors*. O'Reilly Media, 2008.

[Ros03] Winn L Rosch. *The Winn L. Rosch Hardware Bible, 6th Edition*. Que, 2003.

[SR05] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment, Second Edition*. Addison Wesley, 2005.

[Staa] Richard Stallman. *Debugging with GDB*. `http://www.gnu.org/s/gdb/documentation/`.

[Stab] Richard Stallman. *Using the GNU Compiler Collection*. `http://gcc.gnu.org/onlinedocs/`.

[Sto06] Jon Stokes. *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*. No Starch Press, 2006.

[Tan98] Andrew S. Tanenbaum. *Structured Computer Organization (4th Edition)*. Prentice Hall, 1998.

[Tan06] Andrew S. Tanenbaum. *Operating Systems, Design and Implementation (3rd Edition)*. Prentice Hall, 2006.

[Vah95] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall PTR, 1995.

[WD07] Ron White and Timothy Edward Downs. *How Computers Work (9th Edition)*. Que, 2007.

[wRMSMOD] Sandra Loosemore with Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C Library Reference Manual*. `https://www.gnu.org/s/libc/manual/`.

[x86] x86 disassembly. `http://en.wikibooks.org/wiki/X86_Disassembly`.