Proyecto de Sistemas Informáticos.
Curso 2008-2009.

MINIX 3 SOBRE ARQUITECTURA ARM

Componentes del grupo:

J. Adrián Bravo Navarro

Héctor Cortiguera Herrera

Jorge Quintás Rodríguez

Directores del proyecto:

Luis Piñuel Moreno

Manuel Prieto Matías

Facultad de Informática.
Universidad Complutense de Madrid.

Prefacio

En este trabajo detallamos el proceso de desarrollo de un *port* del sistema operativo Minix 3 a arquitectura ARM. Para una mejor comprensión de este proceso, es necesario introducir al lector en una serie de conceptos. Por ello, en primer lugar introducimos Minix 3 y realizamos un análisis de sus aspectos más importantes. A continuación, exponemos las características más relevantes de la arquitectura ARM, comparándola con la arquitectura x86 en algunos puntos relevantes para nuestro trabajo. En las secciones siguientes mostramos la relevancia del proyecto, enumeramos los objetivos iniciales y el estado final que ha alcanzado el desarrollo. Posteriormente describimos en detalle los entresijos de la implementación, centrándonos en los aspectos más relevantes. Debido a su importancia, a continuación se dedica una sección para introducir al lector en el entorno de desarrollo que hemos utilizado, sus componentes y su utilidad, ya que este entorno es un componente crucial del proceso de implementación, y su configuración no es un asunto trivial. Finalmente, exponemos las dificultades con las que nos hemos encontrado en la realización del proyecto, así como el alcance final del mismo y las lineas de trabajo futuro que quedan abiertas.

Palabras clave

ARM, Minix 3, sistemas operativos, system on chip, SoC, microkernel, kernel, dispositivo empotrado.

Abstract

In this work we detail the developing process of porting the Minix 3 operating system to ARM architecture. For a better understanding of this process, we should introduce the reader into some basic concepts. Therefore, we first introduce Minix 3 and make an analysis of its main features. Afterwards, we present the main features of the ARM architecture, comparing with the x86 architecture in some outstanding points for our project. In the next sections we explain the importance of our project, list the initial goals and the state the development has reached. Later we go into details about the internals of the coding process, focusing in the most significant points. Due to its importance, the next section is used to introduce the reader into the development environment that we have used, its components and usefulness, for this environment is a vital part of the coding process, and its setup it is not a trivial issue. Finally, we present the dificulties we have faced during the execution of the project, just as the project scope and the future work lines to pursue.

Keywords

ARM, Minix 3, operating systemas, system on chip, SoC, microkernel, kernel, Embedded systems.

Cesión de derechos

El código fuente del proyecto se encuentra disponible bajo la licencia GPL versión 2, descargable de `http://www.gnu.org/licenses/gpl-2.0.txt`. Por su parte, la memoria del mismo está disponible bajo la licencia GFDL versión 1.3, a su vez descargable de `http://www.gnu.org/licenses/fdl-1.3.txt`.

Sin perjuicio de lo anterior, J. Adrián Bravo Navarro, Héctor Cortiguera Herrera y Jorge Quintás Rodriguez, los autores del proyecto y abajo firmantes autorizamos además a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Madrid, a 18 de Septiembre de 2009.

Adrián Bravo Héctor Cortiguera Jorge Quintás

Agradecimientos

A mi familia, mamá, papá y Rober, por haberme puesto en el camino. A Vicky por haber retirado las piedras, y a Cris por haber caminado conmigo. A Jorge y Héctor, por ser un equipo increíble.

A Cris, por estar siempre conmigo. A mis padres, por haberme ayudado a hacer lo que quería. A mis amigos y compañeros de sobremesa en la facultad, por aguantar quejas cuando las cosas no salían como planeábamos. Y a Adrián y Jorge, por que encontré a los mejores compañeros de prácticas.

A mi extensa familia (¡sois muchos para poneros a todos!). A Unai y Ana por alegrarme la vida. A Pepa e Isidro por ejercer de abuelos. A Franciso, "Viriato", Miguel, Marcelo y Juan, seguís estando aquí. Con especial cariño a Javi e Isabel, a "Isita", a Clara, a Paula, a mi bisabuela Clara, a mis abuelos Carmen, José y Concha, a mi tío Antonio y a mis padres (a vosotros especialmente por aguantarme, sé que no lo pongo fácil). A los compañeros del viaje que se acaba, sobre todo a Adrián y Héctor por vuestro fantástico trabajo y por ser los mejores compañeros.

Al restaurante Tokio de Moncloa, por alimentarnos durante las jornadas más intensas.

Al Aula SUN UCM, por cedernos un espacio para reunirnos y trabajar a gusto.

A esta carrera, por haberse terminado.

Índice general

Capítulo 1

Introducción

1.1. Motivación

En los últimos años han ido apareciendo cada vez más aparatos dotados de pequeñas unidades de procesamiento. Se pueden encontrar sistemas empotrados en infinidad de aparatos, la reciente aparición de los ultraportátiles o la creciente capacidad de procesamiento de los teléfonos móviles necesitan procesadores distintos a los que podemos encontrar en cualquier computadora de sobremesa. Estos sistemas, destinados a aparatos portátiles, han de cumplir unos estrictos requisitos de consumo de energía, sin renunciar por ello a la potencia de cálculo.

De las diversas familias de procesadores destinadas a satisfacer esta creciente demanda, destaca la arquitectura ARM, con un dominio abrumador en mercados como la telefonía móvil, reproductores MP3, consolas portátiles, PDAs y una infinidad de aparatos más. Esta arquitectura combina diversos factores que la convierten en idónea para estos sistemas: bajo consumo, buenas prestaciones de cálculo, una arquitectura RISC que facilita la programación y extensiones que permiten reducir el tamaño del código y un diseño sencillo.

No sólo al *hardware* destinado a los sistemas empotrados y a los aparatos portátiles se le exige un bajo consumo de recursos, también el *software* destinado a ser ejecutado en esas plataformas ha de funcionar en sistemas con recursos limitados: poca memoria, no disponer de sistemas de almacenamiento de datos, etc. Mientras estos sistemas eran dedicados a tareas relativamente sencillas, la aproximación más habitual era el desarrollo de programas ad-hoc, sin embargo, la creciente complejidad de estos aparatos hace necesario un *software* cada vez más complejo.

Así pues, es necesario buscar un compromiso entre la necesidad de un *software* complejo y un sistema sin demasiados recursos, en especial, poca capacidad de memoria. Mientras que algunos de estos dispositivos tienen sus propios sistemas operativos, otros fabricantes han optado por utilizar sistemas operativos como Linux, diseñados para ser ejecutados en entornos menos limitados, y modificarlos para funcionar en estos dispositivos. Sin embargo, esta no parece ser una solución muy acertada, ya que, a pesar de poder realizarse ciertas optimizaciones, son sistemas operativos pensados para un entorno completamente distinto, con demasiada complejidad. Esta complejidad implica dos cosas: un gran volumen de código y una mayor posibilidad de contener algún tipo de error, asumiendo una cantidad mínima de errores por línea de código.

Minix 3 es un sistema operativo que se ha desarrollado con la idea de los dispositivos empotrados en mente. Esto hace que disponga de múltiples características deseables en este tipo de arquitecturas. Para empezar, el sistema al completo -*kernel*, servidores, algunos *drivers* y varias utilidades de usuario- es capaz de ejecutarse en 16MB de RAM, y aún pueden eliminarse componentes no esenciales para reducir el espacio ocupado. Además, uno de los objetivos principales del diseño de Minix 3 es la fiabilidad y la tolerancia a fallos. Cada dispositivo empotrado requiere el desarrollo de controladores específicos para su funcionamiento, que deben integrarse en el sistema operativo. Los estudios han demostrado que el código de los controladores contiene entre 3 y 7 veces más fallos que el resto. Esto hace que un sistema completo pueda quedarse totalmente bloqueado a causa de un error en alguno de sus controladores de dispositivo. En Minix 3, sin embargo, gracias a su arquitectura *microkernel* y en capas, es capaz de aislar este tipo de fallos e incluso solucionarlos sin intervención del usuario mediante el servidor de reencarnación y una política de planificación adecuada.

Otro punto a favor de Minix 3 es su modularidad. Debido a su estructura por capas, es sencillo incluir o eliminar servidores y controladores de dispositivos incluso sin tener que recompilar el *kernel*. Esto permite adaptar Minix a cada sistema empotrado con mayor facilidad, lo que sumado al resto de sus ventajas lo convierten en un sistema operativo idóneo para este tipo de entornos.

Estas ventajas de Minix 3 para el mundo de los sistemas empotrados fue una de nuestras motivaciones a la hora de iniciar el proyecto: a pesar de ser un campo donde iniciativas como las distribuciones Debian para ARM son

ya conocidas, creemos que Minix puede aportar interesantes alternativas como plataforma de desarrollo de nuevos sistemas empotrados, en especial de aquellos que dispongan de menos memoria, o que tengan fuertes requisitos de fiabilidad. Minix 3 ofrece la posibilidad de implantar un sistema de tipo Unix en entornos de recursos limitados.

Por otro lado, hemos de resaltar el valor académico de este proyecto. En la actualidad, Minix se utiliza para impartir cursos de sistemas operativos en gran cantidad de universidades. Minix es un sistema operativo que sobre el papel ofrece ciertas facilidades para ser adaptado a nuevas arquitecturas por su estructura y por su reducido tamaño. La mejor forma de medir esta adaptabilidad es, sin duda, abordarla. Consideramos que la implantación de Minix 3 sobre plataformas basadas en ARM ofrece la posibilidad de utilizarlo en un entorno académico, tanto en cursos de sistemas operativos como en asignaturas acerca de sistemas empotrados. Además, consideramos que este objetivo didáctico del trabajo se vería mermado por lo costoso de las herramientas de desarrollo necesarias para poder manipular una placa, por lo que buscamos una plataforma de desarrollo y depuración completamente basada en *software* libre al alcance de cualquiera.

1.2. Estructura del documento

Antes de proseguir, es interesante hacer una pequeña descripición de cómo está estructurado este trabajo, para poder comenzar la lectura con una idea global. Para ello, en primer lugar, realizamos un estudio del estado del arte, introduciendo los conceptos básicos de las tecnologías en las que se basa este trabajo y detallando aquellos aspectos más relevantes.

Esta primera sección se divide en varios capítulos, empezando en el Capítulo 2, donde se introduce al lector en el sistema operativo Minix 3, su filosofía de diseño y algunos aspectos claves de su arquitectura. Se hace especial hincapié en aquellas partes de Minix que más hubo que modificar durante el desarrollo de nuestro proyecto. A continuación, en el Capítulo 3 se analiza la arquitectura ARM, sus principales características y las diferencias más significativas respecto a la arquitectura Intel x86 para la que fue desarrollado Minix. Por último, en el Capítulo 4 se introducen algunos conceptos sobre gestión de memoria virtual, y se detalla el funcionamiento de la memoria virtual en las arquitecturas Intel x86 y ARM.

Una vez analizados todos estos temas, se procede a describir nuestro trabajo. En primer lugar, en el Capítulo 5 planteamos los objetivos que nos marcamos al comienzo de este trabajo, y los comparamos con los resultados finalmente obtenidos. A continuación, en el Capítulo 6 describimos el proceso de desarrollo y analizamos nuestra implementación del *microkernel* de Minix 3 para una arquitectura ARM, comentando en más detalle aquellos aspectos especialmente problemáticos durante el desarrollo.

Además de explicar nuestra implementación, resulta de especial relevancia hacer un análisis de las herramientas que conforman el entorno de desarrollo y depuración, ya que instalar y configurar un entorno completo y funcional no es un proceso trivial, y está plagado de problemas e imprevistos. El entorno usado para el desarrollo de este proyecto se describe en el Capítulo 7, y en el Apéndice A se explican con todo detalle los pasos necesarios para conseguir su correcto funcionamiento.

Por último, en el Capítulo 8 explicamos los principales obstáculos que nos hemos encontrado durante el desarrollo de este proyecto, y en el Capítulo 9 planteamos una serie de lineas de trabajo prometedoras a la hora de continuar con el desarrollo de Minix sobre ARM.

Capítulo 2

Minix 3

En este capítulo se hará una breve introducción a Minix 3, en primer lugar, comentando la historia de su creación y su evolución desde las primeras versiones del sistema operativo, para a continuación hacer un breve repaso de su filosofía de diseño y su arquitectura.

2.1. Historia de Minix

Minix 3, como su propio nombre indica, es la tercera versión del sistema operativo desarrollado inicialmente por Andrew S. Tanenbaum en 1987, tras la decisión de AT&T de prohibir la libre distribución del código fuente de UNIX. Sin embargo, poco o nada tiene que ver esta tercera versión con las dos anteriores. En 2005, Tanenbaum formó un nuevo equipo de programadores para reescribir completamente Minix como un sistema altamente confiable, por lo que, a pesar del parentesco, Minix 3 es en realidad un sistema nuevo.

Originalmente, Minix fue diseñado con fines educativos, pensado como complemento para los cursos de sistemas operativos que Tanenbaum impartía en la universidad. Actualmente, el objetivo principal de Minix 3 es su uso en dispositivos empotrados, aunque se ha mantenido la faceta educativa del mismo. La idea motora que Tanenbaum quiso darle al proyecto fue desarrollar un sistema tolerante a fallos, mediante la detección y reparación de los errores durante la ejecución y sin intervención del usuario. Esta característica, junto con su reducido consumo de recursos y su modularidad son los motivos que esgrime la comunidad Minix para apoyar la inclusión del sistema en el mundo de los dispositivos empotrados.

Fueron las primeras versiones de Minix las que comenzó a estudiar Linus Torvalds y favorecieron el desarrollo del *kernel* Linux que se usa en la actualidad. Pese a que Linus tenía profundos conocimientos de Minix, no copió ninguna sección de su código, como desmintió el mismo Tanenbaum. A pesar de que Linus comenzó el desarrollo de su kernel con Minix en mente, las decisiones de diseño que tomó alejaron ambos sistemas: mientras Minix 3 se ha mantenido con un *microkernel* sencillo y de pequeño tamaño. GNU/Linux se desarrolló como un *kernel* monolítico. A día de hoy, el núcleo de Minix está compuesto de menos de 5.000 líneas de código, mientras que a principios de 2009, se estima que el *kernel* de Linux contiene unos 10 millones de líneas de código [10].

Las primeras versiones de Minix tenían soporte para varias arquitecturas, entre ellas ARM, Macintosh, Amiga, SPARC... La nueva versión, sin embargo, actualmente sólo es compatible con la arquitectura IA-32 de Intel, aunque existe un *port* para PowerPC (PPC) desarrollado por Ingmar Alting [1] como su proyecto de tesis de máster, dirigida por Tanenbaum, y el soporte a esta arquitectura es bastante completo. Asimismo, existe un proyecto de dar soporte a XSCALE que se encuentra en fases muy tempranas de desarrollo. Las características de Minix 3 hacen del mismo un sistema operativo muy adecuado a los dispositivos empotrados, para lo cual es necesario que disponga de soporte para arquitecturas ARM, ya que los procesadores de esta arquitectura están presentes en la gran mayoría de los sistemas empotrados.

Todas las versiones de Minix están publicadas bajo la licencia BSD y pueden descargarse de manera gratuita de su web [12].

2.2. Introducción a Minix 3

En esta sección y las siguientes pretendemos dotar al lector de una ligera comprensión acerca del funcionamiento interno de Minix 3. Si el lector tiene conocimiento sobre las particularidades de Minix, puede saltarse esta sección con la conciencia tranquila, ya que la información contenida en ella le será conocida. Para el lector no introducido en Minix 3, es necesaria una base sobre el diseño y la arquitectura de Minix 3 para entender los capítulos posteriores. Mucha de la información que se muestra a continuación no es exhaustiva y se recomienda la lectura del libro "The Minix Book" [2], de Andrew S. Tanenbaum y Andrew Woodhull, para profundizar más en la materia.

2.2.1. Diseño y arquitectura

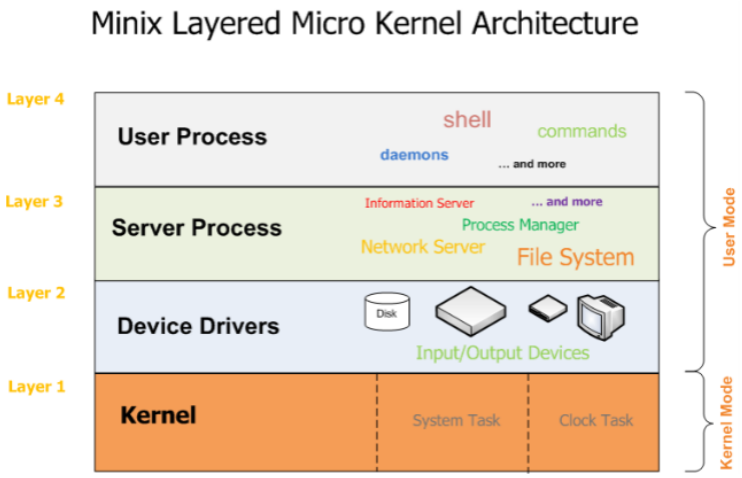

FIGURA 2.1: Organización por capas de Minix

El sistema operativo Minix implementa una arquitectura *microkernel* separada en capas como se muestra en la Figura 2.1. En la teoría de sistemas operativos, tal como se menciona en el libro de Tanenbaum [2] al respecto, existen al menos cinco maneras diferentes en las que se pueden estructurar los sistemas operativos: monolítico, por capas, máquina virtual, *exokernel* y cliente/servidor. Minix 3 combina dos de estas estructuras: la basada en capas y la arquitectura cliente/servidor. La arquitectura basada en capas divide el sistema en una serie de niveles que implementan funciones específicas. En dicha configuración, es habitual que las capas más altas dependan de los servicios ofrecidos por otras capas de nivel inferior. Minix 3 tiene cuatro capas, cada una con una función específica y bien definida.

En una estructura *microkernel* la mayoría de las funcionalidades clave del sistema operativo se implementan como servidores que se ejecutan separadamente del *kernel*. Este diseño hace que el sistema operativo sea modular y extensible, puesto que es posible desarrollar nuevos servicios con relativamente pocos cambios en el *kernel*. Los servicios fundamentales que provee un *microkernel* son la gestión del espacio de direcciones, gestión de hilos, comunicación entre procesos y gestión de los temporizadores.

2.2.2. Estructura de Minix 3

Como se ha comentado con anterioridad, Minix 3 está estructurado en cuatro capas, que se detallan a continuación:

Capa 1: Kernel

Esta capa provee los servicios de más bajo nivel que son necesarios para la ejecución del sistema. Entre ellos se incluyen la gestión de interrupciones, planificación y comunicación. La parte que ofrece servicios de más bajo nivel de esta capa, que trata con interrupciones y otros aspectos muy dependientes del *hardware*, está escrita en lenguaje ensamblador, mientras que el resto de funcionalidades están escritas en C. Esta capa se encarga de lo siguiente:

- Gestionar las interrupciones
- Salvar y restaurar registros
- Planificar procesos
- Ofrecer servicios a la capa superior
- Funciones de comunicación y mensajes

Capa 2: Controladores de dispositivos

En esta capa se encuentra el código que se encarga de las tareas de entrada/salida y da soporte a ciertas tareas que no pueden realizarse a nivel de usuario. Además, en esta capa se encuentran los controladores de dispositivos, para dar soporte a periféricos como discos duros, teclados, impresoras, lectores de CD...

Capa 3: Servidores

Esta capa ofrece servicios que son utilizados por los programas que se ejecutan en la capa cuatro. Los procesos en esta capa pueden acceder a los servicios de la capa dos (controladores de dispositivos) pero los programas de la capa cuatro no tienen acceso directo a los procesos de la capa dos.

Ejemplos de algunos de estos servicios incluyen: sistema de ficheros, servidor de reencarnación, servidor de red, servidor de información, gestor de memoria, gestor de procesos... En la típica aproximación por capas, los servidores

proveen servicios a los programas que se ejecutan en la capa cuatro mientras utilizan servicios ofrecidos por las capas inferiores.

Capa 4: Procesos de usuario

Esta capa comprende la sección de usuario de Minix en la que son ejecutados los programas de usuario. Estos programas utilizan los servicios que ofrecen las capas de nivel inferior. Los programas que se encuentran habitualmente en esta capa incluyen demonios de varios tipos, terminales, órdenes y cualquier otro programa que el usuario quiera ejecutar. Los proceso de esta capa tienen el nivel más bajo de privilegios para acceder a los recursos y normalmente acceden a ellos a través de los servicios que ofrecen las capas inferiores. Por ejemplo, un usuario podría ejecutar la orden `traceroute`, que necesita usar el controlador de red. La orden traceroute no invoca directamente al controlador de red. En su lugar, pasa a través del servidor de ficheros.

2.2.3. Ventajas de la arquitectura

- Modularidad: el sistema está bien estructurado y la relación entre los diferentes componentes está bien definida.

- Seguridad: la combinación de la estructura en capas y *microkernel* facilita la incorporación de mecanismos de seguridad. Las capas dos y tres se ejecutan en espacio de usuario, mientras que tan solo la capa uno se ejecuta en modo *kernel*, que posee todos los privilegios necesarios para acceder a cualquier parte del sistema.

- Extensible: para poder tener un sistema funcional, es necesaria la configuración del *kernel*, así como la de los servicios clave que son necesarios para comenzar. Todas las demás funciones pueden ser añadidas cuando sean necesarias. Esto hace más sencillo ampliar o especializar la función del sistema.

- Rendimiento y estabilidad: muchos problemas que provocan inestabilidad en un computador son resultado de controladores y programas pobremente diseñados. La arquitectura *microkernel* permite a estos programas ser ejecutados e implementados independientemente de los componentes principales del sistema operativo, lo que significa que un fallo en cualquiera de los controladores de dispositivo no es catastrófico para el sistema; puede mantenerse en ejecución, pese a los errores.

2.2.4. Desventajas de la arquitectura

- Complejidad: la arquitectura de Minix tiene una estructura complicada, lo que dificulta, en primer lugar, su diseño y además, su evolución. A pesar de las posibles ventajas de un diseño modular a la hora de adaptar un *software* a un nuevo entorno, en Minix 3 la modularidad no es total, existiendo muchas dependencias entre sus distintas partes. La industria de la informática está entre las más cambiantes dentro de la economía mundial y por lo tanto sufre una necesidad real de adaptarse a los nuevos desarrollos tanto *hardware* como *software*, aspecto en el que Minix no se ha mostrado demasiado apropiado.
- Comunicaciones y envío de mensajes: este tipo de estructura necesita una arquitectura rápida y eficiente de comunicaciones para asegurar la máxima velocidad en la comunicación entre los distintos procesos que se ejecutan en su espacio individual de direcciones, así como con variados niveles de seguridad. Una mala implementación de las comunicaciones tendrá un gran impacto en el rendimiento del sistema.

2.3. Gestión de Procesos

Dentro de la amplitud de la gestión de procesos en un sistema operativo, la parte que se lleva a cabo dentro del *microkernel* es la tarea de planificación. Un sistema multiprogramado se basa en las interrupciones, que permite al núcleo gestionar las peticiones de entrada/salida de los procesos y además controlar los tiempos de ejecución de cada proceso. Los procesos se bloquean cuando hacen peticiones de entrada/salida, permitiendo la ejecución de otros procesos. Cuando la petición ha sido resuelta, el proceso en ejecución es interrumpido por el disco, el teclado o cualquier otra pieza de hardware, y deja de estar activo mientras el dispositivo atiende su petición. El reloj también genera interrupciones, utilizadas para garantizar que un proceso que no ha realizado entrada/salida libere la CPU en algún momento y permita la ejecución de otros procesos. Es tarea de las capas más bajas de Minix 3 ocultar esas interrupciones transformándolas en mensajes. Desde el punto de vista de los procesos, cuando una operación de E/S termina, envía un mensaje a algún proceso, despertándolo y marcándolo como listo para ejecutar.

Las interrupciones también pueden ser generadas por *software*, caso en el que suelen ser llamadas *traps*. Las operaciones `send` y `receive` que se describen en la Sección 2.5 son traducidas por la librería del sistema como interrupciones *software*, que tienen exactamente el mismo efecto que las interrupciones

generadas por *hardware* -el proceso que lanza una interrupción *software* se bloquea inmediatamente y el *kernel* se activa para procesar la interrupción. Los programas de usuario no invocan directamente `send` o `receive`, pero las llamadas al sistema implicadas ejecutan `sendrec` y generan una interrupción *software*.

Cada vez que un proceso es interrumpido (ya sea por un dispositivo E/S convencional o por el reloj) o debido a la ejecución de una interrupción *software*, existe una oportunidad para determinar nuevamente el proceso que tiene más derecho a ejecutarse. Por supuesto, esto también debe realizarse cada vez que un proceso termina, pero en un sistema como Minix 3 las interrupciones debidas a E/S, el reloj o el paso de mensajes ocurren de manera más frecuente que la finalización de un proceso.

El planificador de Minix 3 utiliza un sistema de varios niveles de colas, cada una con distinta prioridad. Se definen dieciséis colas, aunque puede recompilarse para usar más o menos colas de manera sencilla. Para algunas de nuestras pruebas iniciales, el número de colas fue reducido a dos, aunque una vez comprobado que funcionaba correctamente volvimos a restaurar el valor por defecto. La cola de menor prioridad es utilizada únicamente por el proceso `IDLE`, que se ejecuta cuando no hay nada más que hacer. Los procesos de usuario comienzan por defecto en una cola varios niveles de prioridad por encima de la más baja.

Los servidores normalmente se planifican en colas con prioridades más altas que las permitidas a los procesos. Los controladores de dispositivos en colas con prioridades mayores que los servidores y `Clock Task` y `System Task` se planifican en las colas de máxima prioridad. No tienen por qué estar en uso las dieciséis colas en un momento determinado. Los procesos se inician únicamente en algunas de ellas. Un proceso puede ser movido a una cola de prioridad diferente por el sistema o -con limitaciones- por un usuario que invoca la orden `nice`. Los niveles adicionales están disponibles para experimentación, y según se vayan añadiendo controladores a Minix 3, la configuración por defecto puede ajustarse para mejorar el rendimiento.

Además de la prioridad determinada por la cola en la que se coloca un proceso, se utiliza otro mecanismo para dar ventaja a unos procesos sobre otros: el *quanto*, un intervalo de tiempo mínimo que puede ejecutar un proceso antes de ser expropiado, aunque no es idéntico para todos los procesos. Los procesos de usuario tienen un *quanto* relativamente bajo, mientras que los controladores

y los servidores normalmente se ejecutan hasta que ellos mismos se bloquean. Sin embargo, como medida contra el funcionamiento incorrecto de los mismos, se han programado de manera en que puedan ser expropiados, pero se les asigna un *quanto* mayor. Tienen permitido ejecutar por un periodo largo, pero finito, de tiempo, pero si utilizan todo su *quanto* son expropiados para evitar que el sistema se bloquee. En estos casos, el proceso se considera preparado para ejecutar y se coloca al final de su cola. Sin embargo, si un proceso que ha utilizado todo su *quanto* fue el mismo que se ejecutó por última vez, se interpreta que puede estar bloqueado en un bucle y puede estar evitando que otros procesos se ejecuten. En estos casos, su prioridad se ve reducida, colocándolo al final de una cola de prioridad inferior. Si el proceso se quedase de nuevo sin tiempo, su prioridad se vería reducida de nuevo. Tarde o temprano, algún otro proceso tendrá una oportunidad para ejecutarse.

Un proceso al que se le ha reducido su prioridad puede recuperarla. Si un proceso utiliza todo su quanto pero no impide que otros procesos se ejecuten, se asciende ese proceso a una cola de prioridad superior, hasta la prioridad máxima que el proceso tenga permitida. Dicho proceso necesita su *quanto*, pero no está siendo desconsiderado con otros.

Los procesos son planificados utilizando un *round robin* ligeramente modificado. Si un proceso no ha utilizado todo su *quanto* cuando pasa a estar en estado no ejecutable, se interpreta que el proceso se ha bloqueado esperando a E/S, y cuando vuelva a estar listo para ejecutar, se colocará en la cabeza de su cola, pero tan solo con la cantidad de *quanto* que le quedaba cuando se bloqueó. La idea es proporcionar a los procesos de usuario una respuesta rápida a la E/S. Un proceso que es expropiado porque ha terminado su *quanto* se coloca al final de su cola, siguiendo el esquema *round robin*.

Con las tareas situadas en la prioridad más alta, los controladores después, los servidores detrás de los controladores, y los procesos de usuario al final, un proceso de usuario no se ejecutará hasta que ningún proceso de sistema tenga nada que hacer, y un proceso de usuario no puede evitar que un proceso de sistema se ejecute.

Cuando se selecciona un proceso para ejecutar, el planificador comprueba si hay algún proceso esperando en la cola de mayor prioridad. Si hay alguno listo, el que se encuentre en la cabeza de la cola es ejecutado. Si no hay ninguno listo, se comprueba la cola de prioridad inmediatamente inferior, y así repetidamente. Puesto que los controladores responden a peticiones de los servidores,

y los servidores a peticiones de procesos de usuario, en algún momento todos los procesos de prioridad alta completarán la tarea que estén realizando. Entonces se bloquearán sin nada que hacer hasta que algún proceso de usuario tenga oportunidad de ejecutarse y realice más peticiones. Si no existe ningún proceso preparado, entonces el proceso `IDLE` es elegido. Esto coloca a la CPU en un estado de bajo consumo hasta la siguiente interrupción.

En cada *tick* de reloj, se realiza una comprobación para ver si el proceso actual ha agotado su *quanto*. Si lo ha hecho, el planificador lo mueve al final de su cola (lo que puede suponer no hacer nada si está sólo en esa cola). Entonces, se elige el siguiente proceso para ser ejecutado, como se ha descrito anteriormente. Sólo si no hay procesos en las colas de mayor prioridad y el proceso anterior está solo en su cola, será seleccionado para ejecutar inmediatamente. En otro caso, el proceso en la cabeza de la cola de mayor prioridad será ejecutado. Los controladores esenciales y los servidores tienen un *quanto* tan largo que normalmente no son interrumpidos jamás por el reloj. Pero si algo va mal, su prioridad puede ser temporalmente reducida para evitar que el sistema llegue a un bloqueo completo. Probablemente no se pueda hacer nada útil si esto sucede con un controlador imprescindible, pero quizá sea posible apagar el sistema correctamente, evitando pérdida de datos y recolectando información que puede ayudar en la depuración del problema.

2.4. Gestión de Memoria

A pesar de estar diseñado para una arquitectura x86 con memoria segmentada paginada, Minix no hace uso de los mecanismos de paginación, utilizando únicamente memoria virtual segmentada. Esta decisión de diseño fue motivada por varias razones, entre ellas, tratar de mantener el sistema operativo lo más sencillo posible, ya que el primer objetivo de Minix fue ser utilizado en el aprendizaje de los aspectos internos del diseño y la implementación de sistemas operativos. Otras razones fue evitar la excesiva dependencia del código con la MMU de la arquitectura Intel 80386 para poder hacer implementaciones en otras máquinas sin paginación de la época. Esta implementación se ha mantenido hasta la versión más reciente del sistema operativo, aunque una de las propuestas de mejora de Minix a corto-medio plazo es añadir soporte a los sistemas de paginación, y se han añadido algunas características más avanzadas del control de memoria, como la posibilidad de usar memoria de intercambio (*swap*) en disco.

Gran parte de las tareas de gestión de memoria son realizadas por el Gestor de Procesos (*Process Manager*), que mantiene las estructuras de datos necesarias para gestionar los espacios de memoria de cada proceso y la memoria aún no asignada.

2.4.1. Estructuras de datos

Minix 3 mantiene dos tipos de estructuras para la gestión de memoria: estructuras para controlar la memoria aún no asignada, como la lista de huecos en memoria y estructuras para la gestionar los segmentos de memoria asignados a procesos, que forman parte de la tabla de procesos. En Minix, la tabla de procesos no es una única estructura de datos, si no que está repartida entre el *kernel*, el gestor de procesos y el sistema de ficheros, en función de los datos que necesita cada una de estas partes del sistema operativo. Entre los datos incluídos en el gestor de procesos están los mapas de memoria del proceso. El mapa de memoria de un proceso describe los segmentos que forman la imagen en memoria de un proceso: el segmento de código, el segmento de datos y el segmento de pila. Ha de hacerse notar que en la terminología de Minix se utiliza el término segmento para hacer referencia a una unidad puramente lógica, aunque estos segmentos lógicos hayan de ser implementados mediante los registros y descriptores de segmento de la arquitectura x86. A lo largo de esta sección, al hablar de segmentos se hace referencia a los segmentos lógicos, salvo que se especifique que se hable de registros o descriptores de segmentos de la arquitectura.

La definición de la estructura de datos para la tabla de procesos (llamada `mproc`) están en `src/servers/pm/mproc.h`. Dentro de la tabla `mproc`, el campo utilizado para almacenar los segmentos del proceso es el siguiente:

```
struct mem_map mp_seg[NR_LOCAL_SEGS];
```

En cada una de las tres posiciones de este vector se almacena una estructura de datos que define uno de los segmentos del proceso. Esta estructura está definida en `src/include/type.h`

```
/* Memory map for local text, stack, data segments. */
struct mem_map {
  vir_clicks mem_vir;        /* virtual address */
  phys_clicks mem_phys;        /* physical address */
```

```
  vir_clicks mem_len;         /* length */
};
```

En ella se almacenan los tres datos necesarios para describir un segmento: la dirección de comienzo, tanto en memoria virtual como en memoria física y la longitud del segmento. Un detalle a destacar es que estos campos no están medidos en bytes, si no en una unidad lógica propia de Minix, llamada *click*, que en la versión más reciente del sistema operativo se corresponde con 4 KB. A partir de esta estructura de datos se puede generar un descriptor de segmento de la arquitectura x86, rellenando los campos restantes con datos que indican si el segmento pertenece al sistema o a un proceso o si es un segmento de código, datos o pila. El código que permite crear los descriptores de segmentos está principalmente en los ficheros `kernel/system/do_newmap.c` y `kernel/protect.c`.

La otra estructura de datos para la gestión de memoria es la lista de huecos, definida en `include/minix/type.h`:

```
/* Memory allocation by PM. */
struct hole {
  struct hole *h_next; /* next entry on the list */
  phys_clicks h_base; /* where does the hole begin? */
  phys_clicks h_len; /* how big is the hole? */
};
```

En esta lista se guardan los huecos libres en la memoria física, salvo los espacios entre los segmentos de datos y de pila de cada proceso que se consideran espacio ya asignado.

2.4.2. Llamadas al sistema y gestión de memoria

Para poder llevar a cabo algunas de las llamadas al sistema implementadas en el gestor de ficheros, como `exec()` o `fork()` es necesario manipular la memoria virtual, bien sea para reservar la memoria donde se ubicarán el proceso entrante y crear sus segmentos y descriptores de segmento o para liberar esta memoria cuando un proceso ha terminado su ejecución.

Cuando un proceso hace una llamada a `fork()`, el gestor de procesos comprueba en la lista de huecos si hay espacio para alojar los segmentos del nuevo proceso hijo y crea los segmentos necesarios, eliminando la memoria asignada de la lista de huecos, y copia los segmentos de datos y la pila del proceso padre

a los nuevos segmentos de datos y de pila del hijo. Sólo es necesario copiar los segmentos de datos y pila, ya que Minix permite que varios procesos compartan su segmento de código. Una vez asignados estos segmentos, se crea una nueva entrada en la tabla de procesos y se rellena con la información del hijo y su nuevo mapa de memoria.

Para ejecutar una llamada a `exec()`, el proceso es más complicado, ya que además de reservar memoria para el nuevo segmento de código que va a ejecutarse, es necesario liberar la memoria ocupada por el segmento de código del proceso que ha llamado a `exec()`, devolviendo ése segmento a la lista de huecos. Una vez se ha añadido el nuevo hueco, se comprueba que si es adyacente con ningún otro hueco, en cuyo caso se unen para formar un único hueco de un tamaño mayor.

La tercera llamada al sistema del gestor de procesos que cambia el mapa de memoria es `brk()`, que permite cambiar el tamaño del un segmento de datos. En este caso, el espacio de memoria entre el segmento de datos y el segmento de pila no está en la lista de huecos, así que la única comprobación que ha de hacer `brk()` es que el nuevo límite se encuentre en posiciones de memoria del otro segmento.

2.4.3. Cambio de contexto y memoria

Mientras se está ejecutando un proceso, éste tiene guardados sus propios descriptores de segmentos en los registros propios de la arquitectura x86. Al realizar un cambio de contexto, es necesario sustituir los descriptores del antiguo proceso y generar los descriptores del proceso entrante a partir de la información almacenada en su estructura `mproc`.

2.5. Paso de mensajes

El paso de mensajes conforma el mecanismo básico de comunicación entre procesos en Minix. Existen tres primitivas para realizar el paso de mensajes:

```
send(dest, &message);
```

para enviar un mensaje a un proceso,

```
receive(source, &message);
```

para recibir un mensaje desde un proceso, y

```
sendrec(src_dst, &message);
```

para enviar un mensaje y esperar respuesta del mismo proceso. El argumento `message` de cada llamada es la dirección local de los datos del mensaje. El mecanismo de paso de mensajes del *kernel* copia el mensaje de la fuente en el destino, la respuesta (en `sendrec()`) sobrescribe el mensaje original. Este mecanismo podría ser reemplazado por una función que permitiera el envío y recepción de mensajes a través de una red para implementar un sistema distribuido.

Cada tarea, controlador o servidor puede intercambiar mensajes con ciertos procesos. Los procesos de usuario no pueden enviar mensajes a otros procesos de usuario. Cuando un proceso envía un mensaje a otro proceso que no lo espera, el remitente se bloquea hasta que el destinatario efectúa una llamada a `receive()`. Este mecanismo introduce la posibilidad de que se produzcan interbloqueos, por lo que existen ciertas comprobaciones en el *microkernel* para detectarlos y romperlos. Para ciertos eventos, existe la llamada

```
notify(dest);
```

que se usa cuando un proceso tiene que comunicar a otro la ocurrencia de un evento importante. Esta llamada no es bloqueante, evitando la posibilidad de producir un interbloqueo. En este caso el mecanismo de paso de mensajes se utiliza para entregar una notificación, en general el mensaje contiene únicamente la identidad del remitente y un *timestamp* añadido por el *microkernel*. Existen casos en los que una notificación no es suficiente, pero una vez recibida la notificación el proceso destino puede enviar un mensaje al remitente de la notificación para solicitar más información.

Existe una razón para que las notificaciones sean tan simples. Debido a que una llamada a `notify()` es no bloqueante, puede realizarse aunque el receptor no haya llamado a `receive`. La simplicidad del mensaje permite que una notificación no atendida pueda ser fácilmente almacenada hasta que el receptor solicite un `receive()`, de hecho, es necesario un único bit. Las notificaciones se emplean entre procesos del sistema, de los cuales hay un escaso número. Por esto, cada proceso del sistema tiene un mapa de bits para notificaciones pendientes, con un bit para cada proceso del sistema. El mapa de bits proporciona la identidad del remitente, mientras que el *timestamp* es incorporado por el *microkernel* cuando el mensaje es entregado.

Existe un refinamiento sobre el mecanismo de notificaciones. En ciertos casos se utiliza un campo adicional del mensaje de notificación. Cuando la notificación es generada por una interrupción, un mapa de bits de todas las posibles fuentes de interrupción se incluye en el mensaje. Cuando la notificación proviene del *System task* se añade al mensaje un mapa de bits con todas las señales pendientes del receptor. Estos mapas de bit se almacenan en estructuras del *microkernel*, no necesitan ser copiadas al receptor.

Capítulo 3

ARM

ARM es un juego de instrucciones (ISA) RISC de 32 bits desarrollado por ARM Limited. Atendiendo a datos de producción, la arquitectura ARM es el repertorio de instrucciones de 32 bits más extendido. Fue originalmente concebida como procesador para PC's de sobremesa por Acorn Computers, un mercado dominado por la familia x86 usada por los computadores IBM PC. Sin embargo la relativa sencillez de los procesadores ARM los hacían apropiados para aplicaciones de bajo consumo eléctrico. Esto los ha hecho liderar el mercado de la electrónica móvil y de los dispositivos empotrados como microprocesadores y microcontroladores pequeños y asequibles.

Hasta el año 2007 el 98 por ciento de los más de mil millones de teléfonos móviles vendidos anualmente usaban al menos un procesador ARM. En 2009 aproximadamente el 90 % de los procesadores RISC de 32 bits empleados en dispositivos empotrados de 32 bits eran ARM. Estos procesadores son ampliamente usados en electrónica de consumo, incluyendo PDA's, iPods y otros reproductores, videoconsolas portátiles, calculadoras y periféricos de computador como discos duros y routers.

La arquitectura ARM es licenciable. Algunas compañías que son o han sido licenciadas incluyen: Alcatel, Atmel, Broadcom, Cirrus Logic, Digital Equipment Corporation, Freescale, Intel (a través de DEC), LG, Marvell Technology Group, NEC, NVIDIA, NXP (previamente Philips), Oki, Qualcomm, Samsung, Sharp, ST Microelectronics, Symbios Logic, Texas Instruments, VLSI Technology, Yamaha and ZiiLABS.

Los procesadores ARM son desarrollados por ARM y por las empresas titulares de una licencia. Ejemplos de las familias de procesadores de ARM Limited

son ARM7, ARM9, ARM11 y Cortex. Procesadores desarrollados por los principales empresas licenciadas por ARM son DEC StrongARM, i.MX de Freescale, Marvell (anteriormente Intel) XScale,Tegra de NVIDIA, ST-Ericsson Nomadik, y Snapdragon de Qualcomm.

3.1. Historia de ARM

Tras lograr cierto éxito con el microcomputador BBC, la compañía Acorn Computers Ltd consideró cómo pasar del procesador MOS 6502 a un mercado que pronto sería dominado por el PC de IBM, lanzado en 1981. El plan de Acorn Business Computer requería una serie de procesadores secundarios que trabajaran con la plataforma BBC Micro, pero procesadores como el Motorola 68000 y el 32016 de National Semiconductor no servían, y el 6502 no tenía la potencia suficiente para manejar una interfaz gráfica de usuario (GUI).

Dado que los procesadores disponibles no se adaptaban a sus necesidades, Acorn consideró diseñar su propio procesador, con una nueva arquitectura. En este punto los ingenieros de Acorn consultaron las investigaciones llevadas a cabo en las universidades de Stanford y Berkeley acerca de diseños RISC. El proyecto oficial Acorn RISC Machine comenzó en Octubre de 1983 con el objetivo clave de lograr E/S de baja latencia, como la tecnología del MOS 6502 empleado en diseños de computadores Acorn previos. La arquitectura de acceso a memoria del 6502 permitió a los desarrolladores producir máquinas veloces sin usar un costoso *hardware* de acceso directo a memoria (DMA).

El 26 de Abril de 1985 VLSI Technology fabricó el primer chip ARM. Los primeros sistemas de producción reales, ARM2, estuvieron disponibles el año siguiente. El primer uso que recibió el ARM1 fue de procesador adicional para el BBC Micro, donde se usaba para desarrollar *software* de simulación para concluir el trabajo en los chips de apoyo(VIDC ,IOC ,MEMC) y para acelerar el software CAD empleado en el desarrollo del ARM2.

El ARM2 disponía de un bus de datos de 32 bits, un espacio de direcciones de 26 bits(64MB) y dieciséis registros de 32 bits. El código debía estar en los primeros 64MB de memoria, ya que el contador de programa estaba limitado a 26 bits por estar los 6 bits de mayor peso reservados para su uso como registro de estado. El ARM2 era probablemente el microprocesador de 32 bits más simple, con sólo 30.000 transistores, frente a los 70.000 usados en el Motorola

68000. Mucha de esta simplicidad se debe a la ausencia de microcódigo y, como en muchos procesadores del momento, la inexistencia de memoria cache. Esta simplicidad redundaba en un bajo consumo, mientras que el rendimiento era superior al de un Intel 80286. El sucesor ARM3 fue producido con una cache de 4KB, lo que permitió mejorar el rendimiento.

A finales de los 80 Apple Computer y VLSI Technology comenzaron a trabajar con Acorn en nuevas versiones del núcleo ARM. El trabajo fue tan importante que Acorn creó una nueva compañía llamada Advanced RISC Machines Ltd. a partir del equipo de diseño en 1990. Advanced RISC Machines pasó a ser denominada ARM Ltd. cuando ARM Holdings plc. salió a la bolsa de Londres y al NASDAQ en 1998.

El nuevo trabajo de Apple-ARM terminaría siendo el ARM6, lanzado en 1992. Apple usó el ARM 610 (basado en el ARM6) como base de la PDA Apple Newton. En 1994, Acorn empleó el ARM 610 como CPU principal de su computador RiscPC. DEC licenció la arquitectura ARM6 (algo que produjo cierta confusión, ya que DEC también producía los microprocesadores Alpha) y desarrolló el StrongARM. Esta CPU sólo consumía un vatio funcionando a una frecuencia de 233 MHz. (nuevas versiones consumen mucho menos). Este trabajo fue posteriormente cedido a Intel por una resolución legal. Intel acabaría dejando de comercializar sus microprocesadores i960 en favor de StrongARM. Posteriormente, Intel desarrolló su propia implementación de alto rendimiento conocida como Xscale, que fue vendida a Marvell.

El negocio de ARM ha consistido siempre en vender diseños de arquitecturas, llamados IP cores, las cuales son usadas por las empresas licenciadas para crear microcontroladores y CPUs basadas en el núcleo ARM.Las compañías fabricantes (ODM) combinan el núcleo ARM con componentes adicionales para producir una CPU completa que pueda ser producida físicamente y que proporcione un buen rendimiento a bajo coste. La implementación más exitosa ha sido el ARM7TDMI presente en la práctica totalidad de dispositivos dotados de microcontrolador. ARM licenció mil seiscientos millones de núcleos en 2005, de los cuales mil millones fueron a parar a teléfonos móviles. En enero de 2008 se habían vendido más de diez mil millones de núcleos ARM.

La arquitectura común en *smartphones*, PDAs y otros dispositivos portátiles es la ARMv4. Los procesadores XScale y ARM926 son ARMv5TE y actualmente son más empleados en dispositivos de altas prestaciones que los procesadores basados en ARMv4 como StrongARM, ARM925T y ARM7TDMI. El mayor

exponente de la arquitectura ARM está representado por el procesador Cortex-A8, basado en la arquitectura ARMv7. Es capaz de escalar de 600MHz a más de 1GHz, con un consumo menor a 300mW y un rendimiento de 2000 MIPS, evaluado según el *benchmark* Dhrystone. Cortex-A8 es el primer chip superescalar de ARM que incorpora tecnologías para mejorar la densidad de código y el rendimiento, la tecnología NEON TM para multimedia y tratamiento de la señal y la tecnología Jazelle RCT para proporcionar compilación AOT y JIT para Java y otros lenguajes de código intermedio (*bytecode*). En 2009 algunos fabricantes introdujeron *netbooks* basados en CPUs de arquitectura ARM, en competencia directa con los *netbooks* basados en Intel Atom.

3.2. Arquitectura

Para mantener el diseño limpio, sencillo y rápido se optó por una solución cableada en lugar de usar microcódigo, como en el procesador 6502 usado previamente en microcomputadores de Acorn.

La arquitectura ARM incluye las siguientes características RISC:

- Arquitectura Load/Store
- No hay soporte para accesos a memoria no alineados (soportado en núcleos ARMv6, con ciertas excepciones referentes a instrucciones load/store múltiples)
- Banco de registros uniforme compuesto por 16 registros de 32 bits
- Instrucciones de ancho fijo de 32 bits para facilitar la decodificación y el *pipelining*, a costa de la densidad de código
- En general, ejecución en un único ciclo

Para compensar el diseño simple, comparado con procesadores contemporáneos como el Intel 80286 y el Motorola 68000, se añadieron algunas características adicionales:

- Ejecución condicional de la mayor parte de las instrucciones, reduciendo la sobrecarga producida por las instrucciones de salto, compensando de este modo la ausencia de un predictor de saltos
- Las instrucciones aritméticas sólo alteran los bits de condición cuando se precisa este comportamiento

- *Barrel shifter* de 32 bits que puede usarse con la mayor parte de las instrucciones aritméticas y para el cálculo de direcciones sin penalizar el rendimiento
- Potentes modos de direccionamiento indexado
- Un registro de enlace (*link register*)
- Subsistema de interrupciones sencillo pero rápido con dos niveles de prioridad con bancos de registros separados

Los procesadores ARM disponen de otras características raramente vistas en otras arquitecturas RISC, como el direccionamiento relativo al PC y los modos de direccionamiento pre y post incrementados.

Un punto a tener en cuenta es que el juego de instrucciones se ha ido ampliando a lo largo del tiempo. Los procesadores ARM previos al ARM7TDMI, por ejemplo, no disponían de una instrucción que permitiese almacenar una cantidad de dos bytes, de este modo no es posible generar código para ellos que se comporte del modo esperado para elementos en C de tipo `volatile int16_t`.

ARM7TDMI y diseños previos disponen de un *pipeline* de tres etapas, *fetch* (búsqueda de la instrucción), *decode* (decodificación de la instrucción), y *execute* (ejecución). Diseños de mayor rendimiento, como el ARM9, disponen de un *pipeline* de cinco etapas. Otras mejoras para lograr mejor rendimiento incluyen un sumador más rápido y lógica de predicción de saltos mejorada.

La arquitectura ofrece una forma no intrusiva de extender el repertorio de instrucciones mediante el uso de coprocesadores, los cuales pueden ser direccionados mediante los comandos `MCR`, `MRC`, `MRRC` y `MCRR` desde el código. El espacio de coprocesadores está lógicamente dividido en 16 coprocesadores numerados desde el 0 al 15, estando el coprocesador 15 (CP15) reservado para funciones típicas de control como gestión de caché y control de la MMU (en procesadores que dispongan de ella).

En máquinas basadas en ARM los dispositivos periféricos se suelen conectar al procesador *mapeando* sus registros físicos en el espacio de memoria ARM, en el espacio de memoria del coprocesador o conectándolo a otro dispositivo (un bus) que se conecta al procesador. Los accesos a coprocesador tienen menor latencia así que algunos periféricos (por ejemplo el controlador de interrupciones del XScale) están diseñados para ser accesibles de ambas formas (a través

de memoria o de coprocesadores).

Entrando en detalle en el funcionamiento del núcleo, la arquitectura ARM dispone de 31 registros de propósito general de 32 bits, de los cuales 16 son visibles en cualquier momento. El resto se emplean para acelerar el procesamiento de excepciones. Los especificadores de registro de las instrucciones ARM pueden direccionar cualquiera de esos 16 registros visibles.

El banco principal de 16 registros es usado por el código sin privilegios, son los registros del modo usuario (User). El modo *User* es distinto del resto de modos:

- El modo *User* sólo puede pasar a otro modo de ejecución generando una excepción. La instrucción SWI aporta esta funcionalidad
- Los sistemas de memoria y coprocesadores pueden restringir el acceso al modo *User*

Tres de los 16 registros visibles tienen una funcionalidad específica:

Puntero de pila (*Stack Pointer* o SP): El software suele utilizar R13 como puntero de pila

Registro de Enlace (*Link Register* o LR): El registro 14 es el registro de enlace. Este registro contiene la dirección de la siguiente instrucción a ejecutar tras la ejecución de una instrucción de salto con enlace, que es la instrucción que se emplea para realizar una llamada a subrutina. También se emplea para almacenar la dirección de retorno al cambiar de modo de ejecución. El resto del tiempo, LR puede utilizarse como un registro de propósito general

Contador de Programa (*Program Counter* o PC): El registro R15 es el contador de programa

Los restantes trece registros no tienen un propósito especial y pueden ser empleados libremente por las aplicaciones.

Excepciones

La arquitectura ARM soporta siete tipos de excepción, y un modo privilegiado de de ejecución por cada uno de ellos. Los siete tipos son:

- Reset

- Intento de ejecutar una instrucción indefinida
- Interrupciones software (SWI), pueden usarse para implementar llamadas al sistema operativo
- *Prefetch Abort*, error al buscar una instrucción en memoria
- *Data Abort*, error al acceder a datos en memoria
- IRQ, interrupciones normales
- FIQ, interrupciones rápidas

Cuando ocurre una excepción, algunos registros comunes son reemplazados por otros específicos del modo. Todos los modos de excepción disponen de versiones privadas de R13 y R14. El modo FIQ, además, dispone de más registros privados adicionales para un procesamiento rápido de las interrupciones. Cuando se entra en el manejador de la excepción se guarda en R14 la dirección de retorno. Esto sirve para retornar una vez la excepción ha sido tratada y para conocer la dirección de la instrucción causante de la excepción. El registro R13 es privado en cada modo para permitir punteros de pila independientes. Los registros R8-R12 en el modo FIQ son privados para evitar tener que salvarlos y restaurarlos. Existe un sexto modo privilegiado, el modo *System*, que utiliza los mismos registros que el modo *User*. Se usa cuando una tarea requiere acceso privilegiado a memoria y/o coprocesadores. *Reset* comparte el modo privilegiado con SWI.

3.2.1. Modelo de Programación

Los tipos de datos soportados por ARM son:

- *byte* 8 bits
- *halfword* 16 bits
- *word* 32 bits
- *double word* 64 bits

Modos de ejecución

La arquitectura ARM soporta siete modos de ejecución:

Modo de ejecución	Número de modo	Descripción
User (usr)	0b10000	Modo de ejecución normal
FIQ (fiq)	0b10001	Interrupciones rápidas
IRQ (irq)	0b10010	Interrupciones de propósito general
Supervisor (svc)	0b10011	Modo protegido para el sistema operativo
Abort (abt)	0b10111	Permite implementar memoria virtual y/o protección de memoria
Undefined (und)	0b11011	Soporte para emulación *software* de coprocesadores
System(sys)	0b11111	Ejecución de tareas privilegiadas del sistema operativa

TABLA 3.1: Modos de ejecución ARM

El cambio de modo puede ser producido por *software*, a causa de una interrupción externa o por el procesamiento de una excepción. La mayoría de las aplicaciones se ejecutan en modo *User*. En este modo el programa en ejecución no puede acceder a ciertos recursos ni cambiar de modo, a menos que tenga lugar una excepción. Esto permite al sistema operativo controlar el uso de los recursos.

Todos los modos menos el modo *User* son modos privilegiados. Tienen total acceso a los recursos del sistema y pueden cambiar el modo de ejecución libremente. Cinco de ellos son los modos de excepción:

- FIQ
- IRQ
- *Supervisor*
- *Abort*
- *Undefined*

Se accede automáticamente a estos modos cuando se produce una excepción. Cada uno de ellos posee una serie de registros adicionales para evitar corromper el estado del modo *User* cuando se produce la excepción.

El modo restante es el modo *System*, al cual no se accede mediante una excepción y emplea los mismos registros que el modo *User*. Al ser un modo privilegiado carece de las restricciones del modo *User*. Su uso está indicado

para tareas del sistema operativo que necesitan acceder a los recursos del sistema, pero necesitan evitar usar los registros adicionales asociados a los modos de excepción. De este modo se asegura que el estado de la tarea no se altera al ocurrir una excepción.

Registros

En total el procesador ARM dispone de 37 registros:

- 31 de propósito general de 32 bits, incluyendo el contador de programa
- 6 registros de estado. Todos son de 32 bits, aunque no todos los bits son usados

Los registros están dispuestos en bancos parcialmente solapados, siendo el modo de ejecución actual quien controla qué banco está actualmente visible. En cualquier momento, 15 registros de propósito general (de R0 a R15), uno o dos registros de estado, y el contador de programa son visibles.

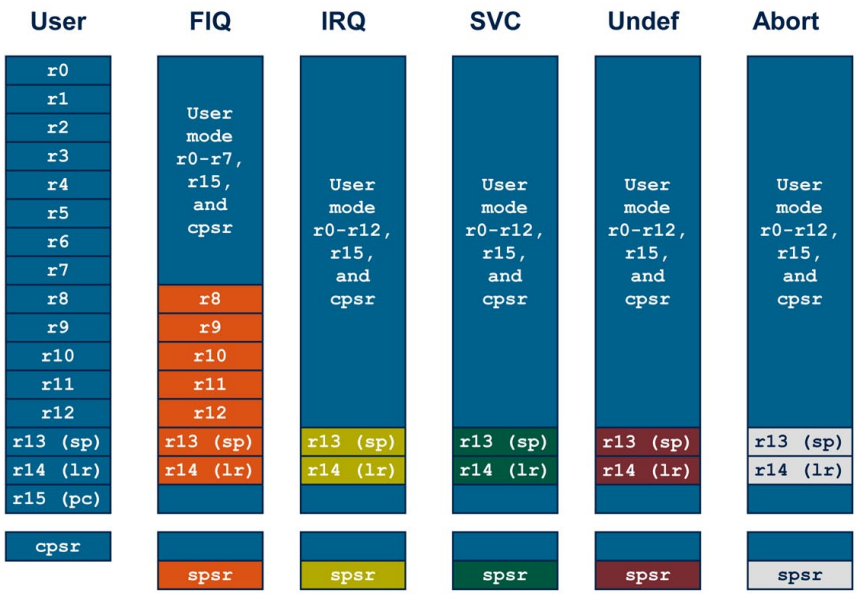

FIGURA 3.1: Esquema de los registros de la arquitectura

Los registros de propósito general R0-R15 pueden dividirse en tres grupos en función de la forma en que se solapan y en sus propósitos específicos:

- Registros no solapados, de R0 a R7
- Registros solapados, de R8 a R14
- El registro 15, el contador de programa

Los registros R0 a R7 no se solapan, esto significa que cada uno de ellos se refiere al mismo registro físico de 32 bits en todos los modos de ejecución. Son registros de propósito general puros, sin usos especiales impuestos por la arquitectura.

Los registros R8 a R14 son registros solapados. El registro físico al que se refiere cada uno de ellos depende el modo de ejecución actual Los registros R8 a R12 disponen de dos bancos registros físicos cada uno. Uno de los bancos está reservado para el modo FIQ y el otro es común para el resto de modos. Estos registros no tienen un cometido especial en la arquitectura, sin embargo, para interrupciones que pueden ser tratadas solamente con los registros R8 a R12, la existencia de un banco aislado para el modo FIQ permite un procesamiento muy rápido de las interrupciones.

Los registros R13 y R14 tienen seis bancos de registros físicos cada uno. Los modos *User* y *System* utilizan uno, y cada uno de los restantes modos de ejecución tiene un banco propio. El registro R13 se usa como puntero de pila y se denomina SP. El registro R14 es el registro de enlace o LR. Tiene dos usos especiales en la arquitectura:

- Dentro de cada modo, LR contiene la dirección de retorno de la subrutina. El retorno de la subrutina se realiza cargando LR en el PC.
- Cuando tiene lugar una excepción el LR del modo de excepción apropiado se carga con la dirección de retorno de la excepción (más un *offset*, dependiendo de la excepción). El retorno se realiza de forma similar al retorno de la subrutina, teniendo en cuenta que es preciso restaurar completamente el estado del programa que se estaba ejecutando.

El registro R15 es el contador de programa, y por tanto las instrucciones tienen ciertas limitaciones al acceder a él. Leer el PC se suele emplear para direccionar instrucciones y datos cercanos independientes de la posición, incluyendo saltos independientes de la posición dentro de un programa. Escribir en el PC supone saltar a la dirección escrita en él.

Registros de estado del programa

El *Current Program Status Register* (CPSR) es accesible en todos los modos de ejecución. Contiene los bits de condición, el bit para habilitar/inhabilitar las interrupciones, el modo de ejecución y otra información de estado y control. Cada modo de excepción dispone, además, del *Saved Program Status Register* (SPSR), que contiene el CPSR del programa interrumpido. El formato del CPSR y del SPSR se muestra en la Figura 3.2:

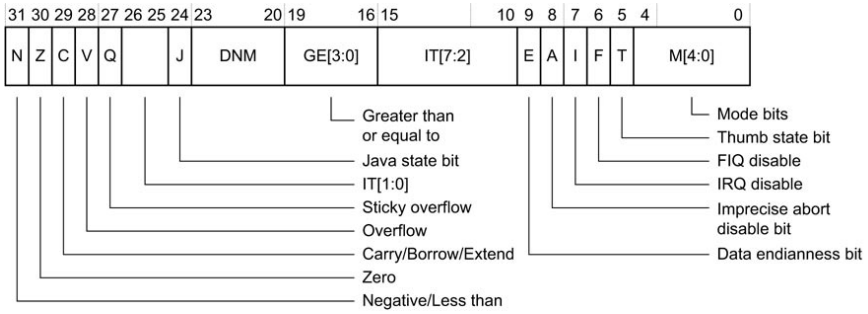

FIGURA 3.2: Esquema de los registros de la arquitectura

Los bits de condición: Los bits N,Z,C y V (Negativo, cero, arrastre y desbordamiento) son los bits de condición o *flags*. Se comprueban para saber si una instrucción condicional debe ser ejecutada

El bit Q: En las variantes E de ARMv5 y superiores se usa este bit para indicar que se ha producido un desbordamiento y/o saturación en un ciertas instrucciones DSP

Los bits GE|3:0| : En ARMv6 y posteriores se usan estos bits para comprobar relaciones de igualdad y de orden en instrucciones SIMD

El bit E: En arquitecturas ARMv6 y posteriores controla el *endianess* del manejo de datos

Los bits de inhabilitación de interrupciones : El bit A deshabilita los *Data Aborts* imprecisos cuando vale 1. En arquitecturas ARMv6 y superiores. El bit I deshabilita las interrupciones IRQ cuando vale 1, mientras que el bit I hace lo propio con las interrupciones IRQ

Los bits de modo: Indican el modo de ejecución del Núcleo.

Los bits T y J: Estos bits se utilizan para seleccionar el juego de instrucciones.

Modo	M[4:0]	Registros accesibles
User	0b10000	PC,P0 a R14, CPSR
FIQ	0b10001	PC, R0 a R7, R8_fiq a R14_fiq, CPSR, SPSR_fiq
IRQ	0b10010	PC, R0 a R12, R13_irq, R14_irq, CPSR, SPSR_irq
Supervisor	0b10011	PC, R0 a R12, R13_svc, R14_svc, CPSR, SPSR_svc
Abort	0b10111	PC, R0 a R12, R13_abt, R14_abt, CPSR, SPSR_abt
Undefined	0b11011	PC, R0 a R12, R13_und, R14_und, CPSR, SPSR_und
System	0b11111	PC, R0 a R14, CPSR

TABLA 3.2: Registros disponibles por cada modo

T	J	Repertorio de instrucciones
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	Reservado

El resto de bits están reservados para futuras expansiones. Es recomendable que el código se escriba de modo que esos bits no sean modificados para prevenir efectos indeseados al ejecutar ese código en versiones futuras de la arquitectura.

Excepciones

Las excepciones son generadas por fuentes internas o externas para que el procesador trate un evento, como una interrupción externa o el intento de ejecutar una instrucción indefinida. Normalmente el estado del procesador (el contenido de CPSR y el PC) es guardado en el momento que se produce la excepción para poder recuperar el programa original, una vez el tratamiento de la excepción ha concluido. Pueden producirse múltiples excepciones al mismo tiempo. La arquitectura ARM soporta siete tipos de excepción, en la siguiente tabla se muestran éstos y el modo de ejecución empleado para procesar cada uno de ellos. Cuando se produce una excepción, se fuerza la ejecución de código desde unas posiciones de memoria fijas, denominadas vectores de excepción[1]:

[1] para FIQ e IRQ deben estar habilitadas las interrupciones en el CPSR para que se salte al vector de excepción.

Tipo de excepción	**Modo**	**Dirección Normal**	**Dirección *High Vector***
Reset	Supervisor	0x00000000	0xFFFF0000
Undefined instructions	Undefined	0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor	0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort	0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C

TABLA 3.3: Excepciones

3.2.2. Extensiones de la arquitectura

Los procesadores ARM han ido incorporando una serie de características destinadas a mejorar ciertos puntos de la arquitectura, como el rendimiento, la densidad de código o las capacidades multimedia. A continuación se introducen estas extensiones.

Thumb

Para mejorar la densidad del código compilado, los procesadores ARM a partir de ARM7TDMI cuentan con el modo Thumb. Cuando el procesador se encuentra en este modo, ejecuta instrucciones de 16 bits. La mayoría de esas instrucciones de 16 bits se *mapean* directamente a instrucciones ARM normales, el ahorro de espacio viene de hacer implícitos algunos operandos de las instrucciones y de limitar las posibilidades con respecto al repertorio ARM completo. La reducción de los códigos de operación (*opcodes*) disminuye la funcionalidad, por ejemplo en este modo unicamente los saltos son condicionales y muchos *opcodes* tienen restringido el acceso a la mitad de los registros de propósito general de la CPU. Los *opcodes* reducidos aumentan la densidad de código incluso a pesar de que para realizar algunas operaciones se requieren instrucciones extra. En situaciones en las que el puerto de memoria o la anchura del bus se reduce a menos de 32 bits, la menor longitud de las instrucciones del repertorio Thumb repercute en una mejora del rendimiento ya que se necesita cargar menos código, algo importante teniendo en cuenta que la situación planteada supone un ancho de banda de memoria mermado. Los dispositivos

empotrados, como las videoconsolas portátiles Game Boy Advance o Nintendo DS, suelen disponer de una pequeña cantidad de RAM accesible a través de una ruta de datos de 32 bits; la mayor parte de la RAM es accedida a través de una ruta de datos secundaria de 16 o menos bits. En esta situación es preferible compilar código Thumb y optimizar a mano las partes que hacen un uso más intensivo de CPU con código ARM de 32 bits, y colocar esas partes optimizadas dentro de la memoria accesible a través del bus de 32 bits. El primer procesador con decodificador Thumb fue el ARM7TDMI. Todos los ARM9 y las familias siguientes, incluyendo el XScale, han incluido un decodificador de instrucciones Thumb.

Instrucciones DSP

Para mejorar la arquitectura ARM para el tratamiento digital de la señal (DSP) se añadieron unas pocas instrucciones. Los procesadores que disponen de este aditamento llevan una E en el nombre, como las arquitecturas ARMv5TE y ARMv5TEJ. Las nuevas instrucciones son comunes en arquitecturas DSP, como suma y resta saturada o multiplicación-acumulación con signo.

Jazelle

Jazelle es una técnica que permite ejecutar *bytecode* Java directamente sorbre al arquitectura ARM como una tercera etapa de ejecución junto a los modos existentes ARM y Thumb.

Thumb-2

La tecnología Thumb-2 se presentó en el núcleo ARM11556, anunciado en 2003. Thumb-2 extiende el limitado conjunto de 16 bits de Thumb con instrucciones adicionales de 32 bits para lograr mayor amplitud en el repertorio de instrucciones. El objetivo perseguido es lograr una densidad de códigio cercana a la de Thumb con un rendimiento cercano al del repertorio de instrucciones ARM sobre memoria de 32 bits. Además, Thumb-2 extiende los repertorios ARM y Thumb con nuevas instrucciones, como manipulación de campos de bits, saltos de tabla y ejecución condicional. Todos los chips ARMv7 soportan el repertorio Thumb-2, otros como el Cortex-M3 solo soportan el repertorio Thumb-2. Otros chips de las series Cortex y ARM11 soportan tanto el repertorio ARM como el Thumb-2.

Thumb Execution Environment (ThumbEE)

ThumbEE, también conocido como Thumb-2EE, y comercializado como Jazelle RCT (*Runtime Compilation Target*), fue anunciando en 2005, apareciendo por primera vez en el procesador Cortex-A8. ThumbEE supone una pequeña extensión al conjunto de instrucciones Thumb-2 que a su vez extiende a Thumb, haciendo el repertorio particularmente apropiado para código generado en tiempo de ejecución (compilación JIT) en entornos de ejecución gestionados. ThumbEE es el objetivo de lenguajes como Limbo, Java, C#, Perl y Python, y permite a los compiladores JIT emitir código compilado más reducido sin penalizar el rendimiento. Algunas nuevas características aportadas por ThumbEE incluyen la comprobación automática de punteros nulos en cada instrucción load/store, una instrucción para realizar comprobaciones de límites en arrays, acceso a los registros R8-R15 (donde reside el estado de la máquina virtual Jazelle DBX), e instrucciones especiales para invocar un *handler*(pequeños fragmentos de código ejecutado frecuentemente, comúnmente empleado para implementar una característica de un lenguaje de alto nivel, como reservar memoria para un nuevo objeto).

Advanced SIMD (NEON)

La extensión Advanced SIMD, comercializada como tecnología NEON, es una combinación de instrucciones SIMD de 64 y 128 bits que ofrece aceleración estandarizada para aplicaciones multimedia y procesamiento de la señal. NEON puede decodificar audio MP3 en CPUs a 10MHz y ejecutar el codec GSM AMR (*Adaptative Multi-Rate*) a 13 MHz. Está compuesto por un completo repertorio de instrucciones, bancos de registros separados y *hardware* de ejecución independiente. NEON soporta enteros de 8, 16, 32 y 64 bits y datos en punto flotante de precisión simple (32 bits). Realiza operaciones SIMD para tratamiento de audio y vídeo así como procesamiento de gráficos y juegos. En NEON se pueden ejecutar hasta 16 operaciones SIMD al mismo tiempo.

VFP

La tecnología VFP es una extensión de coprocesador de la arquitectura ARM. Permite cálculos en coma flotante de simple y doble precisión a bajo coste que cumple totalmente los requisitos del estándar ANSI/IEEE 754-1985

para aritmética binaria de punto flotante. Esta extensión ofrece computación de punto flotante adaptable a un amplio espectro de aplicaciones como PDAs, *smartphones*, compresión/descompresión de voz, gráficos tridimensionales y audio digital, impresoras, *set-top boxes*, y aplicaciones de automoción. La arquitectura VFP también soporta la ejecución de instrucciones con vectores cortos, permitiendo paralelismo SIMD. Esto es útil en aplicaciones gráficas y de procesamiento de señal, al reducir el tamaño del código y aumentar el rendimiento. Otros coprocesadores de punto flotante y/o SIMD incluidos en procesadores basados en ARM son FPA (*Floating Point Accelerator*), FPE (*Floating Point Engine*), iwMMXt (versión para XScale de MMX). Aportan la misma funcionalidad que la unidad VFP pero no son compatibles con ella a nivel de opcode.

Security Extensions (TrustZone)

Las Extensiones de seguridad (*Security Extensions*), comercializadas bajo el nombre de TrustZone, se encuentran en el ARMv76KZ y posteriores arquitecturas orientadas a aplicaciones. Ofrece una solución de bajo coste para añadir un núcleo de seguridad adicional dedicada a un SoC (*System on Chip*), ofreciendo dos procesadores virtuales resguardados por un control de acceso *hardware*. Esto capacita al núcleo para alternar entre dos estados, a los que se denomina mundos, de modo que se puede evitar que la información pase del mundo más confiable al menos confiable. Un mundo puede operar de forma totalmente independiente del otro usando el mismo núcleo, ya que la característica previamente comentada se aplica a toda la funcionalidad del procesador. Se puede restringir el acceso a memoria y periféricos en función del mundo que esté en control del núcleo. Aplicaciones típicas de la tecnología TrustZone son ejecutar un sistema operativo en el mundo menos confiable y un código más pequeño especializado en seguridad en el mundo más confiable (conocido como TrustZone Software, una versión optimizada de Trusted Foundations Software desarrollado por Trusted Logic), permitiendo un DRM (*Digital Rights Management*) más exhaustivo para controlar el uso de medios en dispositivos basados en ARM, previniendo cualquier uso no aprobado del dispositivo.

3.3. ARM frente a Intel

La arquitectura sobre la que está implementado Minix difiere sustancialmente de la arquitectura objetivo de este proyecto. El repertorio de instrucciones Intel es CISC, mientras que el de ARM es RISC. Además, la implementa-

ción de Minix utiliza características de la arquitectura x86 de Intel que no están presentes en la arquitectura ARM, como el cambio de contexto *hardware* y la segmentación.

3.3.1. Segmentación frente a paginación

La memoria virtual en la arquitectura x86 permite utilizar segmentación y memoria segmentada paginada, sin embargo, a pesar de la posibilidad de utilizar segmentación, la gran mayoría de los sistemas operativos no utilizan este mecanismo, o bien utilizan el ardid de agrupar toda la memoria en un único segmento, para así gestionar su memoria mediante paginación a pesar de estar activada la segmentación. Dado que la memoria virtual segmentada no es apenas utilizada en los sistemas operativos modernos, las versiones actuales de ARM con una Unidad de Gestión de Memoria (*Memory Management Unit*, MMU), que permite utilizar paginación, pero no segmentación. Además de no permitir la segmentación, la MMU de ARM es un dispositivo externo controlado como un coprocesador, mientras que la MMU de la arquitectura x86 está integrada dentro del procesador, en forma de un juego de registros de segmentación y registros de paginación.

3.3.2. Cambio de contexto *hardware* frente a cambio de contexto *software*

La arquitectura x86 ofrece ciertas facilidades implementadas a nivel *hardware* para el cambio de contexto. Usando un segmento especial datos, denominado *Task State Segment* o TSS, la CPU puede cargar el nuevo estado con el contenido del TSS cuando se ejecuta una instrucción `CALL` o cuando se produce una interrupción o excepción. El cambio de contexto realizado no es completo ya que no se salvan los registros de punto flotante. En la arquitectura ARM el cambio de contexto no está soportado por *hardware* y su tratamiento debe codificarse en ensamblador. Parte del cambio de contexto puede ahorrarse cuando se produce una interrupción FIQ, ya que en este modo de ejecución los registros R8 a R13 de la arquitectura son privados, así como el PC, el LR y los registros de estado actual y previo (CPSR y SPSR).

3.4. Nuestra elección

En principio el desarrollo se iba a realizar sobre la CPU ARM7TDMI, omnipresente en dispositivos móviles, pero esta opción fue descartada al carecer de MMU. Teniendo en cuenta lo anterior, la siguiente consideración fue emplear un Cortex-A8 para nuestro desarrollo. En el momento de tomar la decisión era la CPU más completa y potente de la familia ARM. A pesar de que el desarrollo no iba a hacer uso de ciertas tecnologías incorporadas en la CPU, como Jazelle, TrustZone o NEON, el disponer de MMU era un factor clave. Otras alternativas con MMU fueron descartadas en principio debido a la existencia de la placa de desarrollo Beagleboard que por un precio muy asequible permitía un desarrollo sobre el dispositivo físico con la CPU elegida. La placa de desarrollo Beagleboard ofrece alto rendimiento con un consumo mínimo, la posibilidad de conectar a dispositivos comunes de PC y un bajo coste. El procesador empleado es un SoC OMAP3530 de Texas Instruments, con núcleo Cortex-A8. Además, existe una comunidad implicada en el desarrollo de *software* para esta placa, y en dar soporte a otros desarrolladores, algo que no ofrecían otras placas. Como se comentará posteriormente, la imposibilidad de desarrollar con herramientas libres sobre este dispositivo, sumado a problemas ajenos, nos forzó a considerar las alternativas que nos ofrecía el *software* de emulación Qemu, decantándonos finalmente por desarrollar para una placa Integrator CP con chip ARM926EJS. Integrator CP es una plataforma compacta de desarrollo para dispositivos basados en ARM. Esta placa consta de un *core module* que proporciona la CPU, SDRAM, SSRAM, controladores de memoria, relojes para el núcleo y una serie de dispositivos implementados en una FPGA, la placa en sí proporciona la alimentación, memoria flash para realizar el arranque y las interfaces para los dispositivos del *core module*. La otra posibilidad que ofrecía Qemu era virtualizar una placa de desarrollo Versatile AB, pero fue descartada por ser mucho más compleja que la Integrator. Por tanto, el enfoque de nuestro desarrollo pasó de estar orientado a un dispositivo físico a un dispositivo emulado bajo Qemu.

Capítulo 4

Memoria virtual

A la hora de trasladar Minix 3 a una plataforma ARM desde la arquitectura Intel x86, no sólo es necesario entender los cambios del repertorio de instrucciones o de los juegos de registros, si no también las distintas maneras de gestionar la memoria virtual de ambas arquitecturas, ya que Minix 3 está bastante ligado a algunos de los mecanismos que provee la plataforma x86. Para ello, en este apartado se hará una breve presentación de algunos de los conceptos básicos de la memoria virtual, describiendo brevemente las diferentes alternativas *hardware* para su implementación. Una vez planteados los diversos sistemas de gestión de memoria virtual, se analizan las implementaciones realizadas en la arquitectura x86, en concreto el sistema de memoria virtual de los procesadores Intel 80386 para los que fue diseñada la primera versión de Minix, y a continuación se estudia el sistema de memoria virtual de la arquitectura ARM.

4.1. Pequeña introducción a la memoria virtual

La memoria virtual es un mecanismo que permite a un procesador disponer de una memoria principal mayor de la que realmente posee, dar a los procesos un espacio de direcciones propio, independientemente de su ubicación en la memoria física y también puede utilizarse como mecanismo de protección de memoria, permitiendo aislar unas regiones de memoria de otras, de una forma completamente transparente al programador de las aplicaciones. Durante su ejecución, un proceso genera direcciones virtuales para acceder a datos o instrucciones dentro de su espacio de direcciones contiguas, que tienen que ser traducidas a direcciones de la memoria física. De esta forma, al compilar y enlazar el código del proceso pueden generarse direcciones que sean independientes del lugar en el que el proceso vaya a ser ubicado en la memoria física. Existen distintos mecanismos de gestión de memoria, utilizando cada uno de

ellos un método distinto de traducción de direcciones.

4.1.1. Paginación

En un sistema con memoria virtual paginada, el espacio de direcciones (virtuales y físicas) es dividido en bloques, llamados páginas. Cada una de las páginas del espacio de direcciones virtuales se corresponde con una página en el espacio de direcciones físicas. Para poder hacer la traducción entre direcciones virtuales y físicas, se utiliza una tabla de páginas. Cada dirección virtual se divide en dos campos: el identificador de página virtual y el desplazamiento dentro de la página. El identificador de página virtual se utiliza como índice para leer una entrada en la tabla de páginas, donde, si la página está activa y presente en memoria, se encuentra el identificador de página física. A partir del identificador de página física y el desplazamiento se genera la dirección física correspondiente a la dirección virtual. Si la página no está activa, se produce un fallo de página.

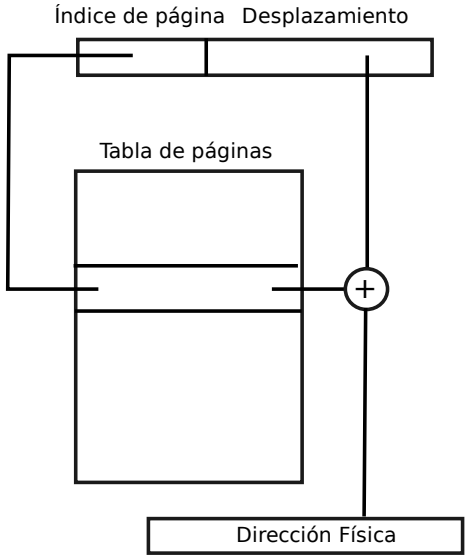

FIGURA 4.1: Traducción de direcciones de un nivel

Para evitar la sobrecarga de memoria que generan estas tablas, puede utilizarse un mecanismo de traducción multinivel. Al tener un único nivel de tablas de páginas, necesitamos una entrada por cada página. Si tenemos un sistema con direcciones virtuales de 32 bits y un tamaño de página de 4 KB, la tabla de páginas tendría 2^20 entradas, ocupando varios megabytes en memoria, y

teniendo un subconjunto reducido de esas páginas activas en la tabla. Con un sistema multinivel, la tabla de páginas de primer nivel divide la memoria en grandes regiones, compuestas por muchas páginas. En un sistema de dos niveles, cada una de estas grandes regiones es direccionada por su propia tabla de páginas, y en la tabla de páginas de primer nivel, en lugar de direcciones de páginas físicas habrá direcciones de tablas de nivel 2. Este mecanismo se puede extender a *n* niveles, teniendo en cada tabla de páginas de nivel *i* direcciones de tablas de páginas de nivel *i+1*, hasta llegar al nivel *n*, donde las tablas de páginas contienen las direcciones de las páginas físicas.

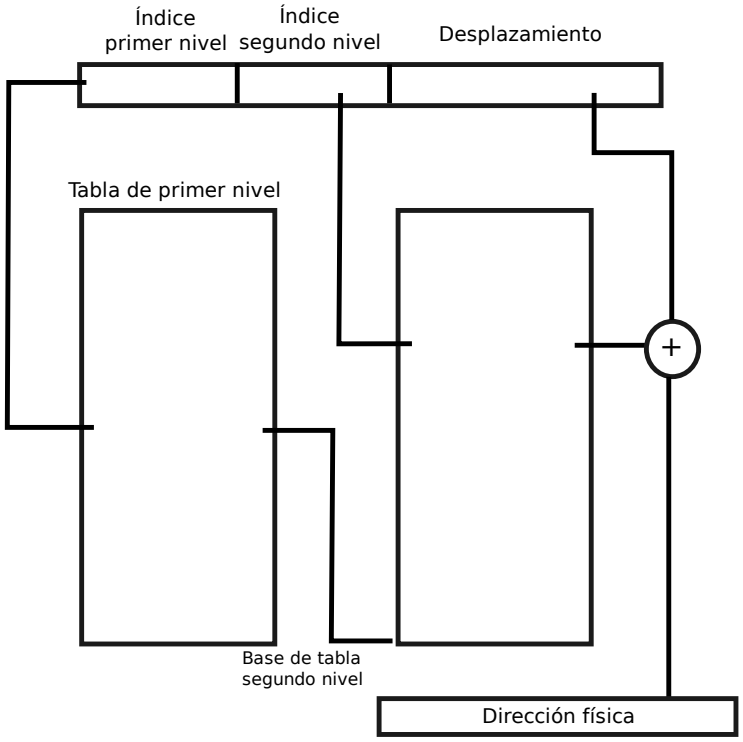

FIGURA 4.2: Traducción de direcciones de dos niveles

4.1.2. Segmentación

Otro mecanismo para gestionar la memoria virtual consiste en crear espacios de memoria independientes, llamados segmentos. Cada uno de estos segmentos es una región de direcciones de memoria consecutivas, desde la región virtual cero hasta una dirección marcada como límite del segmento. Esta dirección límite puede cambiar en el tiempo, permitiendo a los segmentos crecer o

reducirse.

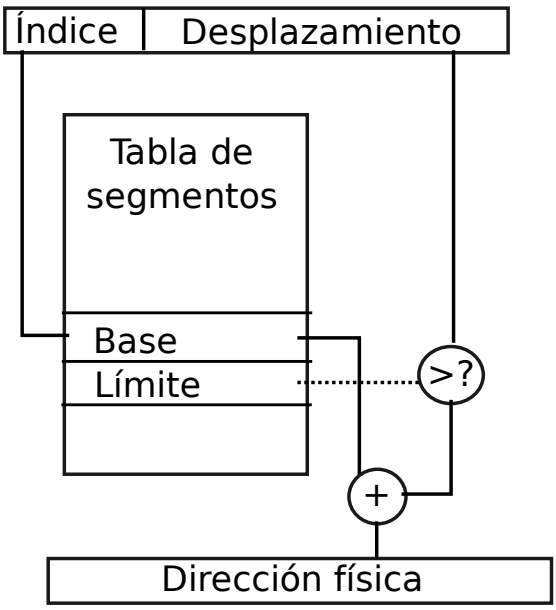

FIGURA 4.3: Traducción mediante segmentos

Las direcciones virtuales generadas por un proceso se dividen en dos campos: un identificador de segmento y un desplazamiento dentro del segmento. Para efectuar la traducción, se necesita una tabla de descriptores de segmentos, en la que se almacena la información sobre cada uno de los segmentos activos. A partir de una dirección virtual, se utiliza el identificador del segmento para acceder a la posición deseada en la tabla, y con la dirección de comienzo en la memoria física y el campo desplazamiento se crea la dirección física.

4.1.3. Segmentación con paginación

Un sistema de memoria virtual segmentada y paginada es similar a un sistema segmentado, al que se añade la posibilidad de dividir los segmentos en páginas. De esta forma, la traducción de una dirección virtual a una dirección física es un proceso en dos etapas: en primer lugar, se realiza una traducción de la dirección física del segmento accedido, y a continuación se utiliza la tabla de páginas del segmento para obtener la dirección física a partir del desplazamiento dentro del segmento.

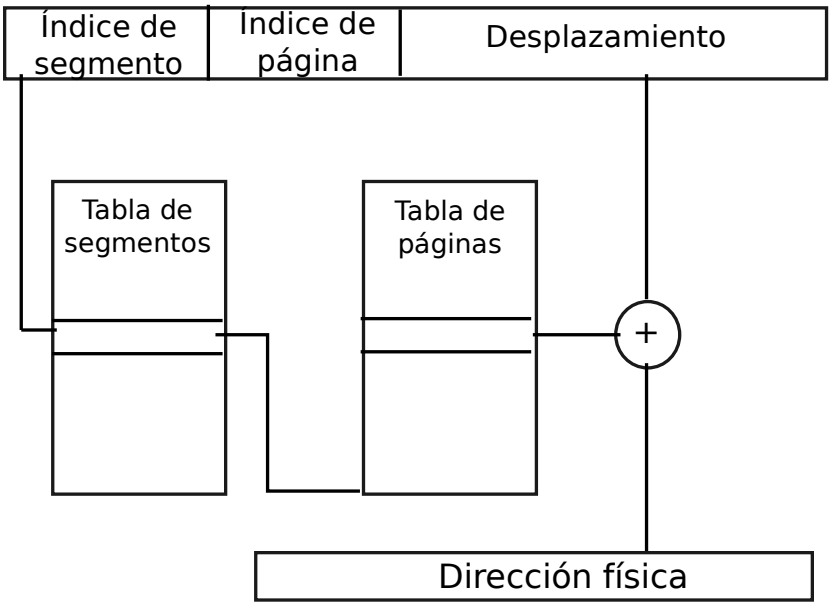

FIGURA 4.4: Traducción mediante segmentación y paginación

4.2. Memoria virtual en Intel x86

El direccionamiento de memoria en la arquitectura x86 depende del modo de ejecución en que se encuentra el procesador, pudiendo distinguirse el direccionamiento en modo real y el direccionamiento en modo protegido. A efectos de este trabajo, describiremos el funcionamiento en modo protegido, ya que es el modo de ejecución utilizado por Minix. El modo real es un modo compatible con anteriores versiones de la arquitectura que utilizaban un direccionamiento de 20 bits, pudiendo acceder únicamente a 1 MB. de memoria.

La arquitectura x86 implementa memoria virtual segmentada paginada. El tamaño de las direcciones físicas y virtuales es variable, dependiendo de la versión de la arquitectura, y, por simplicidad, en este apartado se describirá el funcionamiento de los mecanismos de traducción para el Intel 386 (Intel 80386), ya que la primera implementación de Minix fue diseñada para este procesador. El sistema operativo ha cambiado sustancialmente desde entonces, pero su gestión de memoria virtual sigue utilizando el mismo diseño de la primera versión. En esta implementación de la arquitectura se utilizan direcciones virtuales de 46 bits, que son traducidas a direcciones físicas de 32 bits en un proceso de traducción de varias etapas.

Una dirección virtual está compuesta de un selector de segmento de 16 bits, de los cuales 14 son de dirección, y un desplazamiento de 32 bits. Los selecto-

res de segmento están almacenados en una serie de registros de segmento del procesador. Hay seis de estos registros, llamados CS, DS, ES, SS, FS y GS. El registro CS almacena el selector del segmento de datos del proceso en ejecución, el registro DS el selector para el segmento de datos, SS guarda el selector para el segmento de pila, ES apunta al descriptor de un segmento de datos adicional utilizado por algunas instrucciones, y FS y GS contienen los selectores de dos segmentos adicionales a disposición de los procesos, sin tener definido un propósito concreto. Las direcciones generadas por los programas son los desplazamientos de 32 bits, que se combinan con los 14 bits de dirección del selector para formar la dirección virtual de 46 bits.

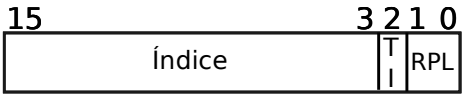

FIGURA 4.5: Selector de segmento

Como se puede ver en la Figura 4.5, el registro selector de segmento se divide en tres campos: 13 bits para identificar uno de los 8192 descriptores posibles, un bit, TI, que indica, si su valor es cero, si el selector pertenece a la Tabla de Descriptores Globales (*Global Descriptor Table*, GDT) o a la Tabla de Descriptores Locales (*Local Descriptor Table*, LDT) si su valor es uno. Además, para resolver los permisos de acceso se utilizan los dos bits RPL (*Requested Privilege Level*), que indican el nivel de privilegio de la dirección solicitada. Con el selector de segmento podemos acceder a una de las dos tablas de descriptores de segmentos, dependiendo del valor del bit TI. La tabla de descriptores globales contiene descriptores de segmentos comunes a todas las aplicaciones y segmentos propios del sistema, mientras que la Tabla de Descriptores Locales contiene descriptores propios del proceso en ejecución en un momento dado.

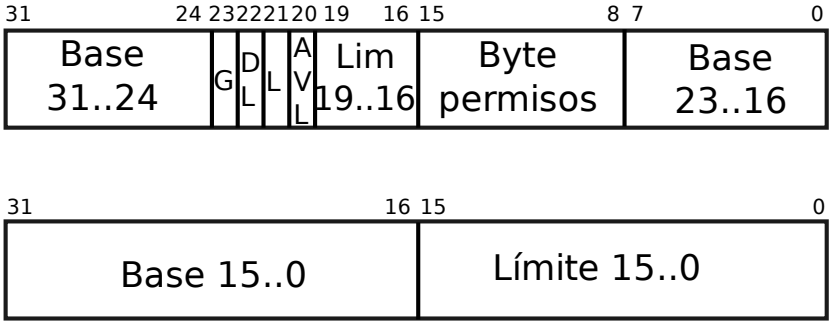

FIGURA 4.6: Descriptor de segmento

Cada uno de estos descriptores es un dato de 64 bits que contiene la dirección base y la longitud del segmento, así como otros bits de control. La dirección base está divida en tres partes, ocupando los bits del 63 al 36, del 49 al 32 y del 31 al 15, con un total de 32 bits. La longitud del segmento está dividida en dos partes, entre los bits 51 y 48 y los bits 15 y 0, con un total de 20 bits. Los bits 55 y 54 permiten cambiar el tamaño máximo del segmento. El bit 54 (DL) indica si el segmento es de datos de 32 bits, si vale uno, o 16 bits si vale cero. El bit 55 (G) indica la granularidad con la que se mide la longitud. Si G vale uno, la longitud está expresada en páginas, si no, está expresada en bytes. El bit 52 (AVL) indica si el segmento está disponible.

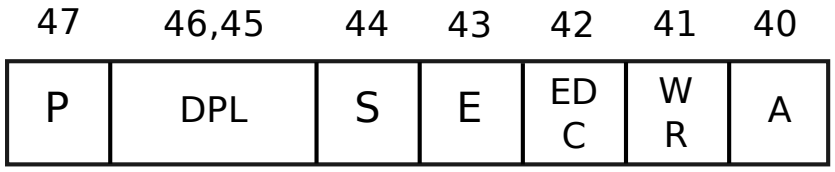

FIGURA 4.7: Byte de permisos de acceso

Los bits del 47 al 40 se llaman byte de derechos de acceso. Su funcionamiento es el siguiente

- Bit 47: Presencia (P), indica si el segmento está presente en memoria principal
- Bits 46 y 45: Codifican el nivel de privilegios o DLP del segmento
- Bit 44: Indica si el segmento pertenece al sistema o a un proceso de usuario
- Bit 43: Se llama bit E, e indica si el segmento es un segmento de datos (si vale cero) o de aplicación (si vale uno). De su valor depende la interpretación de los bits 42 y 41
- Bit 42: Si E=0, se llama bit ED, si su valor es uno, es un segmento de datos y crece hacia posiciones altas de memoria. Si su valor es cero, es un segmento de pila y crece hacia posiciones bajas de memoria. Si E=1, se llama bit C, y si su valor es cero, indica al procesador que ignore el mecanismo de nivel de privilegios al acceder al segmento
- Bit 41: También depende del valor del bit E. Si E=0, se llama bit W y controla los permisos de escritura sobre este segmento. Si E=1, se llama bit R y controla los permisos de lectura sobre este segmento

- Bit 40: Bit A o bit de acceso. Si el segmento ha sido accedido su valor es uno

Para poder acceder a las tablas de descriptores de segmento el procesador tiene varios registros, invisibles al programador, que contienen un puntero a la primera posición de cada una de las tablas de descriptores necesarios. Estos registros son GDTR, que apunta a la tabla GDT, LDTR, que apunta a la tabla LDT e IDTR, que apunta a una tabla llamada IDT utilizada para el manejo del vector de interrupciones.

Una vez obtenido el descriptor de segmento, se comprueba que la dirección a la que queremos acceder no sobrepasa el tamaño del segmento, que se dispone de los permisos de acceso necesarios y se crea una nueva dirección a partir de la dirección base del segmento y el desplazamiento de la dirección virtual, llamada dirección lineal. Si se está utilizando memoria segmentada paginada, la dirección lineal se traduce a una dirección física utilizando tablas de páginas. Si la paginación está desactivada, la dirección lineal se utiliza como dirección física para el acceso a memoria.

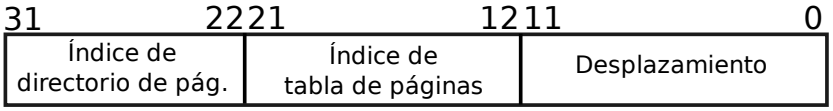

FIGURA 4.8: Formato de dirección lineal

La traducción de direcciones se realiza mediante tablas de páginas de dos niveles. El primer nivel se conoce como directorio de páginas, que divide la memoria en bloques de 4 MB. En cada una de las 1024 entradas del directorio de páginas hay un puntero a una tabla de páginas de segundo nivel. Cada una de estas tablas de páginas de segundo nivel tiene 1024 entradas a páginas de 4 KB.

Para acceder al directorio de páginas, la arquitectura x86 cuenta con un juego de registros de control con información acerca del sistema de paginación, llamados CR0, CR1, CR2 y CR3. El registro CR0 contiene una serie de bits de control, como el bit PG (bit 31), utilizado para activar o desactivar el mecanismo de paginación. Los bits 30 y 29 (CD y NW) sirven para deshabilitar la caché o deshabilitar la escritura write through en memoria respectivamente. En los veinte bits más significativos de CR3 se encuentra la dirección base del directorio de páginas para el proceso de traducción. Si durante el acceso a memoria se produce un fallo de página, la dirección que ha provocado el fallo se almacena en CR2.

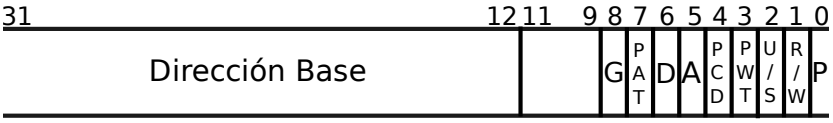

FIGURA 4.9: Formato de entrada en una tabla de páginas

Utilizando la dirección base del directorio de páginas y el desplazamiento indicado en los bits 31 a 22 de la dirección lineal obtenemos la entrada del directorio. El formato de las entradas es idéntico tanto en el directorio de páginas como en las tablas de páginas de segundo nivel y contiene la dirección de memoria en la que comienza la tabla de páginas de segundo nivel y siete bits de control: un bit P de presencia en memoria, un bit W con el permiso de escritura, bits para definir el comportamiento de la caché y los bits A y D, que indican si la región que traduce esta entrada ha sido accedida y modificada, respectivamente. A partir de la entrada en el directorio obtenemos la dirección de la tabla de segundo nivel. Utilizando los bits 21 a 12 de la dirección lineal obtenemos la entrada correspondiente a la página a la que queremos acceder. Con la dirección base y los once bits menos significativos de la dirección lineal, obtenemos la dirección física correspondiente y podemos acceder al dato o la instrucción solicitada.

4.3. Memoria virtual en ARM

Mientras que los procesadores basados en la arquitectura Intel x86 implementan una memoria virtual segmentada paginada, la mayoría de las implementaciones de ARM disponibles sólo hacen uso de la paginación. Para manejar el sistema de memoria virtual, el núcleo ARM delega en el coprocesador CP15, llamado *System Control Coprocessor*, encargado de gestionar la Unidad de Gestión de Memoria (*Memory Management Unit*, MMU), configurar el comportamiento de las cachés del procesador, controlar el TLB (*Translation Lookahead Buffer*), control del Acceso Directo a Memoria (*Direct Memory Access*, DMA) para los dispositivos, y otras tareas de configuración y control del sistema de memoria.

El sistema de memoria virtual de ARM utiliza un mecanismo de traducción de direcciones de uno o dos niveles. En la traducción de primer nivel, la memoria se divide en zonas de 1 MB, llamadas secciones. De esta forma, el espacio de 4 GB formado por direcciones de 32 bits queda dividido en 4096 secciones. Cada una de estas secciones es direccionada por una entrada en la tabla primaria, llamada Tabla Maestra (*Master Table*). Estas entradas (*Page Table Entry*, PTE) son datos de 32 bits, y pueden ser de varios tipos. El tipo de cada entrada se

identifica mediante sus dos bits menos significativos: 00, para las entradas inválidas (*Fault*), 01, para las entradas que apuntan a una tabla de segundo nivel de tipo disperso (*Coarse*), 11 para las que apuntan a una tabla de segundo nivel de tipo fino (*Fine*) y 10 para las entradas de sección.

Las entradas marcadas como inválidas sirven para indicar que un rango de direcciones virtuales no es accesibles en la memoria física.

Las entradas de sección (*section entry*) realizan una traducción de un único nivel de una sección de 1 MB. Al traducir un rango de posiciones de memoria de 1 MB, las direcciones traducidas mediante entradas de sección se componen de un campo de 20 bits de desplazamiento dentro de la sección y 12 bits para identificar la sección. Así, una entrada de sección se compone de los siguientes campos:

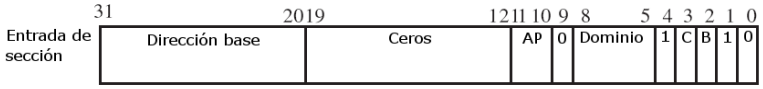

FIGURA 4.10: Entrada de sección

Los once bits más significativos indican la dirección física base de la sección, los bits 10 y 11 codifican los permisos de acceso a la misma, los bits de 8 a 5 indican el dominio al que pertenece la sección, los bits 3 y 2 indican la política de caché y *write buffer* a emplear para esta sección y los dos bits menos significativos contienen el valor "10", indicando el tipo de entrada. El bit 4 siempre vale uno, el bit 9 vale cero y los bits de 19 a 12 contienen ceros.

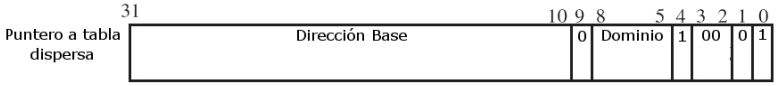

FIGURA 4.11: Entrada apuntando a una tabla dispersa

Si la entrada apunta a una tabla de página dispersa (*coarse*), los bits del 31 al 10 contienen la dirección en la que se encuentra el comienzo de la página de segundo nivel, que tiene que estar alineada en una dirección múltiplo de 1 KB y en los bits del 8 al 5 se codifica el dominio al que pertenece la región descrita por la tabla de páginas. La tabla de páginas dispersa consta de 256 entradas de 32 bytes, que pueden apuntar a páginas de 64 KB (*large pages*) ó 4 KB (*small pages*), aunque no deben intercalarse páginas de distinto tamaño, y ocupa 1 KB en memoria. Cada una de las entradas tiene el siguiente formato:

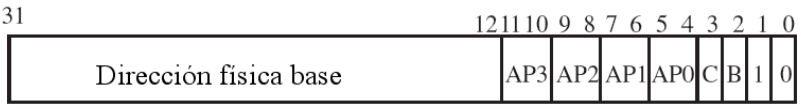

FIGURA 4.12: Entrada de una tabla dispersa

Si la entrada apunta a una página de 4 KB la dirección física de comienzo de la página se especifica en los veinte bits más significativos, y si apunta a una página de 64 KB, en los dieciséis bits más significativos, siendo ceros los cuatro restantes. Los bits 11 a 4 comprenden cuatro campos de dos bits que permiten asignar permisos de acceso con una granularidad menor que el tamaño de página. Si la página es de 64 KB se pueden especificar permisos a subpáginas de 16 KB y si la página es de 4 KB se divide en cuatro subpáginas de 1 KB.

Este sistema permite especificar permisos de acceso distintos en función del modo de ejecución del procesador y de los bits *System* (S) y *ROM control* (R) en el registro CPSR. De esta manera se puede crear un sistema de protección de memoria a nivel de página. La siguiente tabla especifica los niveles de permisos.

AP	S	R	Acceso en modo Sistema	Acceso en modo Usuario
00	0	0	Sin acceso	Sin acceso
00	1	0	Sólo lectura	Sin acceso
00	0	1	Sólo lectura	Sólo lectura
00	1	1	Impredecible	Impredecible
01	x	x	Lectura/Escritura	Sin acceso
10	x	x	Lectura/Escritura	Sólo lectura
11	x	x	Lectura/Escritura	Lectura/Escritura

El cuarto tipo de entrada en la tabla maestra es un apuntador a una tabla fina (*fine*). Este tipo de tablas de páginas han ido desapareciendo de las implementaciones más modernas de la arquitectura ARM. En estas tablas se pueden direccionar 1024 páginas de 1 KB (*tiny pages*). Los 22 bits más significativos de la entrada contienen la dirección de comienzo de la página, y en los bits 5 y 4 están codificados los permisos de acceso.

4.3.1. Memory Management Unit

Mientras en el Intel 80386 el control y la configuración de la memoria virtual se realizaba con una serie de registros del procesador, en la arquitectura ARM se delega esta tarea a una unidad externa, la Unidad de Gestión de Memoria

(*Memory Management Unit*, MMU). Esta unidad se encarga del mecanismo de paginación y del proceso de traducción de direcciones virtuales a direcciones físicas, gestiona el TLB, las cachés de datos e instrucciones y el buffer de escritura a memoria.

Esta unidad está conectada al núcleo ARM como un coprocesador, llamado coprocesador CP15 o Coprocesador de Control del Sistema (*System Control Coprocessor*). Para comunicarse con esta unidad, es necesario utilizar las instrucciones de transferencia de registros `MRC` y `MCR`.

- `MCR p, o, Rd, CRn, CRm, q` Lee el contenido del registro Rd del núcleo ARM y lo escribe en el registro CRn del coprocesador indicado por `p`. Si el registro Rd es R15, el resultado de la operación es impredecible. El campo `o` es el código de operación primario, el campo `q` es el código de operación secundario y CRm es un operando fuente o destino adicional.

- `MRC p, o, Rd, CRn, CRm, q` Escribe el contenido del registro CRn del coprocesador `p` en el registro Rd del núcleo ARM. El resto de campos tienen la misma función que en la instrucción `MCR`.

Los registros accesibles del coprocesador CP15 son quince, sin embargo, algunos están duplicados y dependiendo del código de operación secundario, especificado en la instrucción `MCR` o `MRC`, se accede a distintos registros con el mismo identificador. El funcionamiento de cada registro es el siguiente [13]:

- c0: Contiene información sobre la implementación de la MMU, como un código de fabricante, número de versión, etc.

- c1: Hay dos registros identificados como c1: *Control Register*, al que se accede con el código de operación primario y secundario a cero (`MCR p15, 0, Rd, c1, c0, 0`) y *Non-Secure Access Control Register*, al que se accede con un dos como código de operación secundario (`MCR p15, 0, Rd, c1, c0, 2`). Este segundo registro no resulta de relevancia para este trabajo, y no será explicado en detalle.

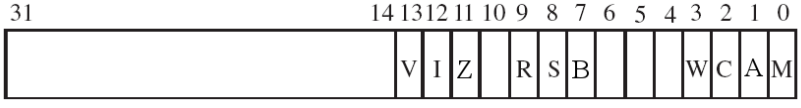

FIGURA 4.13: Registro de control

El *Control Register* contiene diversos *flags* para configurar el comportamiento de la MMU. De los 32 bits del registro, resultan de interés para este trabajo el bit M (bit 0), activa o desactiva la gestión de memoria virtual, el bit A (bit 1) que activa la comprobación de alineamiento en los accesos a memoria, el bit C (bit 2) activa la caché de datos, el bit W (bit 3) controla el buffer de escritura, el bit B (bit 7) indica si la memoria es *big endian* (B=1) o *little endian* (B=0), el bit Z (bit 11) activa el predictor de saltos, el bit I (bit 12) activa la caché de instrucciones y el bit V (bit 13) indica la posición de los vectores de interrupción (si V=0 están situados de 0x0 a 0x1c, si V=1 de 0xffff0000 a 0xffff001c)

- c2: En este registro se guarda la dirección base física de la tabla de páginas maestra. Este registro está duplicado para poder tener dos tablas primarias. Se puede acceder a este registro con el código de operación 0 ó 1.

- c3: Mediante este registro pueden activarse o desactivarse los distintos dominios en la memoria. El registro se divide en 15 campos de dos bits, uno por cada dominio. Si el campo de un dominio concreto se escribe a 00 ó 10, este dominio es inaccesible, si se escribe a 01 sólo es accesible desde el propio dominio, y si se escribe a 11 se desactiva la protección de ese dominio.

- c4: Registro sin uso.

- c5: Este registro se utiliza para almacenar un código de error en caso de fallo durante la traducción de direcciones virtuales a físicas. Se utiliza un juego de dos registros: uno para almacenar el código de error en caso de fallo en el acceso a datos (*Data Fault Status Register*) y otro para almacenar el código en caso de acceder a una instrucción (*Instruction Fault Status Register*). Para leer el registro *Data Fault*, utilizamos un cero como código de operación, y para leer el *Instruction Fault* utilizamos un uno. Los códigos de error son los siguientes:

  - 0000 = No hay errores
  - 0001 = Acceso no alineado
  - 0010 = Fallo en evento de depuración
  - 0011 = Error en el bit de acceso (bit A de AP) en traducción de un nivel
  - 0100 = Fallo en la gestión de la caché de instrucciones
  - 0101 = Fallo en acceso a sección (traducción de un nivel)

  - 0110 = Error en el bit de acceso (bit A AP) en traducción de dos niveles
  - 0111 = Fallo en traducción de páginas (traducción de dos niveles)
  - 1000 = *Precise external abort*
  - 1001 = Fallo de dominio en sección (traducción de un nivel)
  - 1010 = No utilizado
  - 1011 = Fallo de dominio en página (traducción de dos niveles)
  - 1100 = External abort en traducción de un nivel
  - 1101 = Error en el bit de permisos (bit P de AP) en traducción de un nivel
  - 1110 = External abort en traducción de dos niveles
  - 1111 = Error en el bit de permisos (bit P de AP) en traducción de dos niveles

- c6: En este registro se almacena la dirección del dato (*Fault Address Register*) o la instrucción (*Instruction Fault Address Register*) a la que se intentaba acceder cuando se produjo el fallo. Para acceder al *Fault Address Register* se utiliza un cero como código de operación, y un dos para acceder al *Instruction Fault Address Register*.
- c7: No utilizado.
- c8: Registro para el manejo del TLB. Algunas de las operaciones posibles son:
  - `MCR p15, 0, <Rd>, c8, c7, 0`. Invalida todas las entradas del TLB (datos e instrucciones) que no estén bloqueadas.
  - `MCR p15, 0, <Rd>, c8, c6, 0`. Invalida todas las entradas del TLB de datos.
  - `MCR p15, 0, <Rd>, c8, c5, 0`. Invalida todas las entradas del TLB de instrucciones.
- c9: No utilizado.
- c10: Permite marcar ciertas entradas del TLB como bloqueadas y no son sustituidas con nuevas traducciones.
- c11: No utilizado.
- c12: No utilizado.
- c13: Utilizado para depuración.

- c14: No utilizado.
- c15: Permite cambiar las entradas bloqueadas del TLB.

Capítulo 5

Minix@ARM

Minix 3 es un sistema operativo del que se han llevado a cabo pocos *ports* a otras arquitecturas que no fueran Intel. En septiembre de 2006 se publicó una tesis [1] sobre un *port* de Minix 3 para Power PC (PPC), llevado a cabo por un alumno de la Vrije Universiteit (VU) y dirigido por el propio Andrew S. Tanenbaum. El sistema quedó casi completo, y es posible arrancar sobre un PPC la copia de Minix 3 y utilizar la mayor parte de los servicios. Sin embargo, la mayor contribución de esta tesis fue la separación del código fuente de Minix 3 en *hot spots* dependientes de la arquitectura, lo que facilita la tarea de localizar el código dependiente en futuros proyectos como el nuestro.

El soporte de Minix 3 para ARM es uno de los *milestones* de la comunidad Minix. Anteriormente han realizado intentos en este sentido, existiendo una persona [29] designada para realizar esta tarea desde hace años, y que sin embargo no ha dado aún frutos. En la convocatoria del GSoC [28] (*Google Summer of Code*) 2008, se abrió un proyecto [30]para llevar a cabo un *port* parcial de Minix 3 a plataformas ARM. Ninguno de estos proyectos llegó a liberar ninguna versión del código, y la mayoría de proyectos relacionados con Minix 3 en los GSoC han sido abandonados [32]sin completar. Sin embargo, en la planificación dentro del wiki de Minix 3, se considera un proyecto a medio plazo tener disponible una versión del sistema operativo que se ejecute sobre plataformas ARM.

El hecho de que gente seguramente más preparada, y en ocasiones guiada por los desarrolladores originales de Minix 3, haya sido incapaz de completar esta iniciativa debería dar una idea aproximada de la dificultad intrínseca del proyecto. Es necesario, además, tomar en consideración el tiempo real invertido en la realización del proyecto, que si bien comenzaba con el inicio del curso

universitario, debido a las dificultades técnicas con los entornos de desarrollo comentadas en el Capítulo 7 no comenzó realmente hasta muy avanzado el curso.

5.1. Objetivos iniciales

Los objetivos iniciales del proyecto incluían *portar* Minix 3 a la arquitectura ARM, así como escribir controladores para algunos dispositivos existentes en la placa de desarrollo. En un principio la placa era una BeagleBoard, y los controladores a implementar eran para el puerto serie y, si daba tiempo, para la pantalla táctil que contiene la placa. Más adelante, cuando se cambió de plataforma por problemas con el entorno de desarrollo, se propuso desarrollar únicamente el controlador del puerto serie. Desde el primer momento se era plenamente consciente de que los objetivos eran demasiado ambiciosos y que muy probablemente no fueran alcanzables, pero pareció más motivador plantear grandes objetivos y no llegar a completarlos que objetivos más simples que pudieran completarse a mitad de camino.

5.2. Objetivos alcanzados

A pesar del tardío comienzo del proyecto, se ha logrado que las capas de bajo nivel de Minix 3 funcionen correctamente, lo que nos ha permitido desarrollar el *microkernel* al completo, con la excepción del paso de mensajes -implementado pero no completamente probado-, que se ha quedado a pocas semanas de ser funcional. Además, se ha añadido soporte para memoria virtual, que es algo que no se utiliza en Minix 3 -sobre Intel- con páginas, sino que se lleva a cabo aprovechando la facilidad arquitectónica de los segmentos. Por último, se han implementado controladores para los temporizadores y el dispositivo serie de la placa, lo que nos permite introducir y recibir datos por pantalla, y de esta forma tener una noción más visual de lo que está ocurriendo en el sistema. El temporizador ha sido imprescindible para poder implementar completamente el planificador de Minix 3, que se comporta de manera idéntica al original.

Capítulo 6

Implementación

Para comenzar con el desarrollo en sí de Minix sobre la arquitectura ARM son necesarios ciertos pasos posteriores a la obtención de un entorno de desarrollo y previos a la fase de implementación de funcionalidad contenida en Minix 3. El primero de ellos es realizar la correcta inicialización de la placa de desarrollo y de sus dispositivos. Una vez logramos esto será necesario crear un entorno multiprogramado.

En este punto comienza la adaptación de la parte más baja de Minix. En la versión para procesadores Intel, esta parte se encarga del intercambio de procesos, el paso de mensajes, las rutinas de tratamiento de las interrupciones y de la preparación parcial del entorno para la ejecución del símbolo `main()` del *microkernel*. Fundamentalmente esta capa es la encargada de cada transición al *microkernel*. En nuestro desarrollo comenzamos esta etapa dando soporte al intercambio de procesos empleando el scheduler de Minix sobre el sistema multiprogramado que programamos. La gestión de las interrupciones se había abordado durante la inicialización de la placa, pero las rutinas de tratamiento de las mismas se fue refinando a medida que requería funcionalidad. La implementación del paso de mensajes se fundamenta en los puntos anteriormente expuestos y por esta razón se comenzó su implementación en fechas cercanas a la conclusión del proyecto. Su estado es incompleto aunque las dependencias a nivel ensamblador están implementadas.

Uno de los principales escollos a salvar a la hora de implementar Minix 3 sobre ARM era la sustancial diferencia entre los sistemas de gestión de memoria. A pesar de ser diseñado para x86, que permite utilizar segmentación y paginación, Minix 3 prescinde de este último mecanismo, realizando toda la gestión de memoria mediante segmentación pura. Sin embargo, la Unidad de

Gestión de Memoria de ARM no permite la segmentación, teniendo que adaptar la gestión de memoria de Minix a un esquema paginado.

La implementación del sistema de memoria virtual hizo necesaria la creación de las utilidades básicas para la gestión de memoria dinámica. Tomando como punto de partida código de las librerías de Minix, se reescribió la parte dependiente de la arquitectura.

6.1. Multiprogramación

Se denomina multiprogramación a la técnica que permite la ejecución de múltiples tareas compartiendo los recursos de un mismo computador. Se trata de una evolución del procesamiento por lotes. Con la multiprogramación se explota el paralelismo real entre CPU y E/S y se optimiza el uso de los recursos disponibles.

El uso de la multiprogramación se ve mejorado con la memoria virtual ya que permite, a efectos prácticos, que los procesos se ejecuten como si los demás procesos no existieran.

6.2. Scheduler

Para que el planificador de Minix 3, analizado en la Sección 2.3 funcione en ARM, requiere un soporte básico en ensamblador que fue necesario implementar desde cero. Una vez preparado el soporte básico, se fueron ampliando las funcionalidades del planificador. En un principio, todos los procesos tenían un único *tick* de *quanto*, usaban un número restringido de las dieciséis colas de planificación y se omitieron todos los chequeos de privilegios sobre los procesos (si un proceso es facturable, si es interrumpible...). Poco a poco se fueron integrando nuevos ficheros de Minix 3 que cubrían estas necesidades hasta finalmente consqguir que el planificador de Minix@ARM tenga la misma funcionalidad que el de Minix 3.

6.3. Memoria virtual

La aproximación utilizada ha sido emular los segmentos lógicos de Minix mediante segmentos formados por una o más páginas consecutivas en memoria. Este sistema está aún en fase de mejoras, ya que, aunque permite simular satisfactoriamente los segmentos de Minix, no se aprovecha toda la potencia

de un sistema paginado.

Para poder gestionar los segmentos lógicos de Minix, fue necesario crear un tipo de datos que pudiera comportarse de la misma manera, pero basándose en paginación. El tipo de datos definido fue el siguiente:

```
struct Region {
unsigned int vAddress; //first position in virtual memory
unsigned int length; //size in number of pages or clicks
unsigned int AP; //access permission
unsigned int CB; //cache and write buffer policies
unsigned int pAddress; //physical address
struct Pagetable *PT; //pointer to father page table
};
```

Se ha utilizado el nombre `Region` para sustituir los segmentos de Minix, para recalcar el hecho de que no se basan en mecanismos *hardware* de segmentación. En lugar de ello, definimos regiones de direcciones contiguas formadas por varias páginas. Esta estructura de datos resulta equivalente a la estructura `mem_map` de Minix: tiene una dirección virtual de comienzo, su traducción a memoria física y el tamaño, medido en *clicks* o páginas, ya que de entre los diversos tamaños de página permitidos por ARM, elegimos las páginas de 4 KB, que permiten una gestión más fácil de las tablas de páginas y coinciden con el tamaño de *click* en Minix 3. Además de estos tres atributos compartidos con la estructura original de Minix 3, hay otros atributos dependientes de la Unidad de Gestión de Memoria (MMU) de ARM, como los permisos de acceso `pagetable.AP` y la política de gestión de cachés y *write-buffer* especificada en `pagetable.CB` que se utilizará en esa región de memoria. El último atributo de esta estructura es un puntero a la primera tabla de páginas que direcciona esta región de memoria.

Para poder utilizar la memoria virtual de ARM es necesario gestionar no sólo estas regiones, si no también las tablas de páginas que permiten realizar las traducciones entre las direcciones virtuales y las direcciones físicas. El sistema de memoria virtual de ARM puede utilizar traducciones tanto de un nivel como de dos, y se necesita crear dos tipos de tablas de páginas: una tabla maestra para el primer nivel de traducción y tablas de páginas de segundo nivel. Mediante la siguiente estructura se pueden definir los dos tipos de tablas:

```
struct pagetable {
unsigned int vAddress; //first position in virtual memory
unsigned int ptAddress; //location in physical memory
unsigned int masterPtAddress; //pointer to the master table
unsigned int type; //type of table
unsigned int dom; // domain
struct pagetable *next; // pointer to the next pagetable
};
```

El principal atributo de una tabla de páginas es la primera dirección virtual que traducen. En caso de tratarse de la tabla maestra, la dirección de comienzo es 0, ya que permite direccionar los 4 GB de memoria virtual. Si la tabla es una tabla de nivel dos permite direccionar 1 MB de espacio de memoria virtual. El tipo de tabla definido por esta estructura se guarda en el campo `Pagetable.type`. ARM permite varios tipos de páginas, sin embargo, en este trabajo sólo se utilizan dos tipos: la tabla maestra y tablas dispersas de segundo nivel, que permiten direccionar 1 MB con 256 páginas de 4 KB. El campo `pagetable.dom` especifica el dominio al que pertenecerán las páginas direccionadas mediante esta tabla, y el campo `pagetable.PtAddress` es un puntero a la tabla de primer nivel. De esta forma, para activar o desactivar las traducciones de esta tabla de páginas se tiene un acceso directo a la tabla maestra, sin pasar por la Unidad de Gestión de Memoria. El último campo es un puntero a la siguiente tabla de páginas, ya que una tabla de páginas sólo permite direccionar 1 MB mediante 256 entradas a páginas de 4 KB. Para poder direccionar regiones de un tamaño mayor, es necesario utilizar listas, que contengan una tabla de páginas por cada megabyte de la región.

Mediante estas estructuras de datos se consigue que cada proceso tenga su propio espacio de direcciones virtuales aislado del resto. Para ello, en el cambio de contexto únicamente es necesario desactivar las entradas de la tabla de primer nivel correspondientes al proceso saliente, se limpia la caché de traducciones del TLB y se escriben en la tabla de páginas maestra las entradas correspondientes a las tablas de páginas del proceso entrante. Para poder llevar a cabo el cambio de páginas, cada proceso tiene asignadas sus propias regiones en la estructura proc, en lugar de los segmentos de la arquitectura x86. Así, cuando un proceso entra en ejecución por primera vez, es necesario crear sus tablas de páginas en memoria. Estas tablas estarán siempre presentes en una región de la memoria destinada a tal fin, evitando la sobrecarga de tener que crear las tablas en cada cambio de contexto.

6.4. Paso de mensajes

El paso de mensajes requiere la copia eficiente de bloques de memoria. La versión para arquitectura x86 de Minix realiza esta copia con una función *ad hoc* implementada en ensamblador que efectúa la copia del mensaje que se quiere enviar y la escritura del remitente sobre el mensaje copiado al mismo tiempo. Esta forma de proceder hace que interpretar el paso de mensajes sea difícil y poco metódico, además de tener un diseño poco modular. La solución adoptada en nuestro caso se apoya en `memcpy` quedando el código más estructurado y accesible.

Sobre esta base se comenzó a colocar el código encargado de la gestión del paso de mensajes. Esta parte no está finalizada y su código se encuentra separado de la rama principal por los problemas que generaba.

6.5. Gestión de memoria dinámica

El uso de listas enlazadas para gestionar las páginas de la memoria virtual hizo necesaria la implementación de un sistema de gestión de memoria dinámica.

El soporte fundamental de la memoria dinámica es el *heap* o montículo. Esta región de memoria comienza en el *break*(primera dirección tras la zona de datos no inicializados) y crece hacia la pila. Puede darse el caso de que estas regiones de memoria lleguen a solaparse por su crecimiento; esto no es deseable y se debe tener en cuenta en la implementación.

Podemos obtener la información necesaria para definir el *break* gracias al linker a través de la etiqueta `_end`, que apunta precisamente al valor que debe tener inicialmente *break*.

Ampliar la sección de datos de un proceso implica desplazar *break*, de esto se encarga la función `sbrk(size_t incr)`. Además `sbrk` comprueba que el heap no ha invadido la pila, con un cierto margen (en nuestro caso 1024 bytes).

La implementación de `sbrk` depende también del diseño escogido para pila y *heap*. La posición de una región de memoria con respecto a la otra y el sentido en que crece el heap son aspectos fundamentales a tener en cuenta.

El código que realiza la ampliación de la zona de datos es el siguiente:

```
.global _sbrk
break:
.word _end
_sbrk:
 stmfd sp!,{r1,r2}
    ldr r1,=break    @load old end of heap zone
    ldr r1,[r1]
    stmfd sp!,{r1}   @save old end of heap zone
    add r1,r1,r0     @break = break + size
    ldr r2,=break    @load break address
    str r1,[r2]      @store the new limit for heap zone
    mov r2,sp        @load effective address of stack pointer
    sub r2,r2,#1024  @safety  area for checking stack invasión
    cmp r1,r2
    bhi crash
    ldmfd sp!,{r1}
    mov r0,r1        @place the return value
    ldmfd sp!,{r1,r2}
    mov pc,lr
```

La librería de Minix que decidimos emplear para dar soporte a memoria virtual requería a su vez la existencia de `memcpy`, de la cual se han mantenido dos versiones, una programada en C para dar soporte a la función `realloc()` en un corto plazo de tiempo y una rutina en ensamblador que se completó posteriormente para hacer las copias de forma más eficiente.

Una vez terminadas las partes dependientes de la arquitectura se usó el código C incluido en Minix para la gestión de la memoria dinámica.

6.6. Inicialización

Si bien parte de la inicialización la delegamos a U-Boot(que en el caso de Integrator CP a su vez, delega en ARM Boot), existen dispositivos que deben ser configurados para conseguir el estado deseado en el entorno de ejecución. U-Boot realiza la configuración mínima para que pueda ser cargado en memoria un kernel que continúe o sobrescriba la configuración realizada previamente y es por esto que la imagen que cargamos en RAM debe realizar unos pasos extra.

6.6.1. Placa

La inicialización básica de la placa es realizada por ARM Boot (invocado por U-Boot). Se activan los bancos de memoria y se colocan en la dirección 0x00000000 del mapa, donde se copiarán los vectores de interrupción de U-Boot (las rutinas de tratamiento proporcionadas por U-Boot solo informan del tipo de fallo y muestran el estado del procesador en ese instante) y se configuran los dispositivos que va a emplear como la UART, el *display* para obtener datos y presentar información y el dispositivo de red, ya que se puede copiar información a la RAM mediante *tftpboot* o NFS.

6.6.2. Vectores de interrupción

Las rutinas de tratamiento de las interrupciones que vamos a necesitar difieren enormemente de las que emplea U-Boot. La gestión de las interrupciones *hardware* es el pilar básico del cambio del contexto, necesario para disponer de multiprogramación. La gestión de memoria virtual emplea las rutinas de tratamiento de *Data Abort* y *Pefetch Abort* para dar soporte a la depuración. El resto de interrupciones se tratan con un bucle infinito para ayudar a la hora de encontrar los posibles errores de programación. Esta es la tabla de vectores de interrupción:

```
b reset
ldr pc, _undefined_instruction
ldr pc, _software_interrupt
ldr pc, _prefetch_abort
ldr pc, _data_abort
ldr pc, _not_used
ldr pc, _irq
ldr pc, _fiq
```

Tanto los vectores como las rutinas de tratamiento de las interrupciones se encuentran en la imagen del *kernel* y son copiadas a la dirección 0x00000000:

```
.global _copy_vector_table
_copy_vector_table:
adr r0, _vectors_start /* pc relative */
ldr r1, =0 /* destination address */
ldr r2,=_vectors_start
ldr r3,=_vectors_end
```

```
copy_loop:
ldmia r0!, {r4-r10} /* copy from source address [r0]*/
stmia r1!, {r4-r10} /* copy to   target address [r1] */
cmp r0, r3 /* until source end address [r2] */
ble copy_loop
mov pc, lr
```

donde el núcleo del procesador espera que estén, aunque virtualmente se *mapean* a 0xffff0000 (*high vectors*), para que los procesos puedan tener direcciones virtuales comenzando en 0x00000000. Debido a que U-Boot también se copia, es posible volver a su *prompt* en las condiciones que lo dejó restaurando su contexto y sus vectores de interrupción.

6.7. Dispositivos

6.7.1. Temporizadores

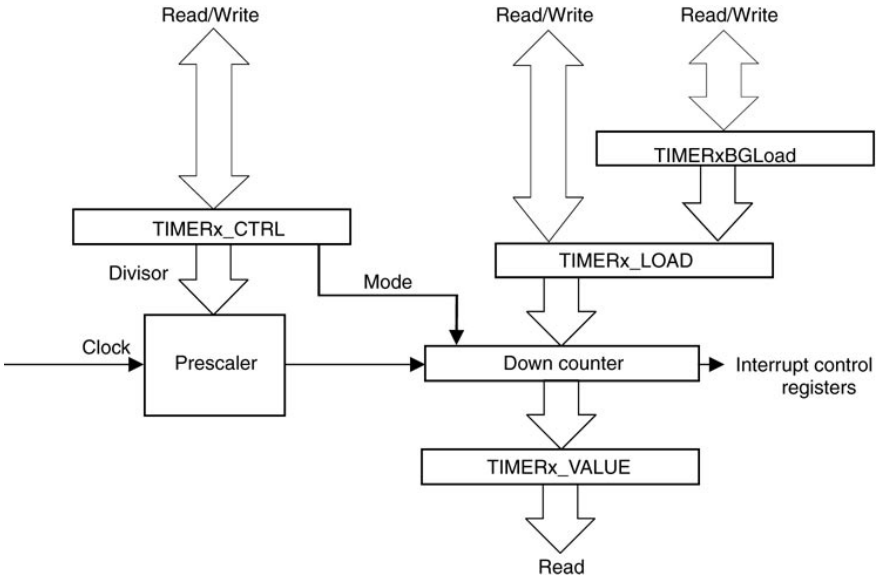

FIGURA 6.1: Estructura de los temporizadores

La placa empleada dispone de dos temporizadores programables de 32 bits. Timer0 tiene su señal de reloj conectada al bus del sistema, mientras que Timer1 y Timer2 están conectados a un reloj que oscila a 1MHz. Además cada uno de los dispositivos está compuesto por:

Dirección	Tipo	Ancho	Nombre	Descripción
0x13000200	Read/Write	32	Timer2Load	Valor a cargar en el temporizador
0x13000204	Read	32	Timer2Value	Valor actual del temporizador
0x13000208	Read/write	8	Timer2Control	Registro de control
0x1300020C	Write	-	Timer2IntClr	*interrupt clear*
0x13000210	Read	1	Timer2RIS	*raw interrupt status*
0x13000214	Read	1	Timer2MIS	*masked interrupt status*
0x13000218	Read/write	32	Timer2BGLoad	Valor de carga en segundo plano

TABLA 6.1: Estructura temporizador 2

- Un contador descendente de 32 bits con divisor (también puede operar como contador de 16 bits)
- Dos registros de carga
- Registro de control
- Registro de control de interrupciones

Los contadores tienen tres modos de operación:

- Cuenta libre: cuenta hasta 0, carga su valor máximo y continúa
- Periódico: cuenta hasta 0, carga el valor presente en el registro de carga y continúa
- Un disparo: cuenta hasta 0 y se detiene

En nuestra implementación usamos el temporizador 2, cuya funcionalidad y estructura detallamos a continuación:

- Timer2 load register: El registro de carga del temporizador empleado es un registro de 32 bits que contiene el valor que va a ser decrementado por el contador. Este es el valor que se va a recargar en el contador cuando éste opere en modo periódico y la cuenta llegue a cero. Si se escribe directamente en este registro, el valor se carga al llegar el siguiente flanco ascendente de la señal de reloj y la cuenta prosigue desde el nuevo valor.

El valor de este registro se sobrescribe si se efectúa una escritura al registro Timer2BGLoad, pero la cuenta actual no se ve afectada inmediatamente. Si se escriben valores a los registros Timer2Load y Timer2BGLoad pasa lo siguiente:

- En el siguiente pulso de reloj, el valor escrito a Timer2Load reemplaza la cuenta actual
- Cada vez que el contador llegue a cero, el valor del contador se actualiza al valor escrito en Timer2BGLoad

El valor leído de Timer2Load es siempre el valor que se cargará en modo periódico cuando la cuenta llegue a cero.

- Timer 2 background load: Este registro de 32 bits contiene el valor que se va a cargar en el contador cuando el modo periódico esté activado, y la cuenta actual llegue a cero. Con este registro se proporciona un método alternativo para acceder a Timer2 Load Register. La diferencia estriba en que las escrituras a Timer2BGLoad no provocan que el contador se reinicie inmediatamente. Las lecturas de este registro devuelven el mismo valor que las lecturas de Timer2Load
- Timer 2 current value register: El contenido de este registro es el valor actual de la cuenta, el valor del temporizador
- Timer 2 control register: Los registros de control del temporizador se usan para configurar las operaciones de los contadores/temporizadores, su formato es el indicado por la Figura 6.2 y detallado en la Tabla 6.7.1
- Timer 2 clear register: Este registro de sólo lectura no dispone de almacenamiento, escribir cualquier valor en su dirección anula el *flag* de la interrupción asociada al contador/temporizador
- Timer 2 raw interrupt status register: El bit 0 de este registro indica el estado de las interrupciones lanzadas por el contador. Si este bit vale 1, el contador ha lanzado una interrupción. El valor que toma el pin de salida de interrupciones del temporizador es el resultado de efectuar la AND lógica al valor de este registro y al bit 5 del registro de control (bit que habilita/inhabilita las interrupciones)
- Timer 2 interrupt status register: El bit 0 de este registro contiene el estado de la máscara de interrupciones del contador. Su valor coincide con el valor del pin de salida de interrupciones anteriormente comentado

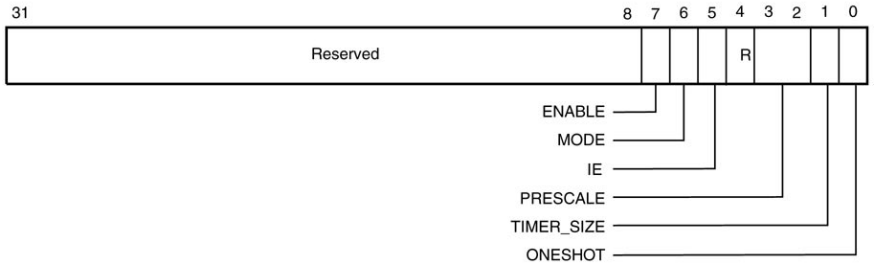

FIGURA 6.2: Registro de control de cada temporizador

Bit	Nombre	Función
[7]	ENABLE	Timer enable: 0 = inhabilitado 1 = habilitado
[6]	MODE	Timer mode: 0 = cuenta libre, al llegar a 0 recarga su máximo valor 1 = periódico, al llegar a 0 se recarga el contenido de Load Register
[5]	IE	Habilitación de interrupciones
[4]	R	No usado
[3:2]	PREESCALE	divisor de preescalado
[1]	TimerSize	Selecciona el modo de operación de 16 bits (0) o 32 bits (1)
[0]	OneShotCount	Selecciona el modo de un disparo (0) o el continuo (1)

TABLA 6.2: Control del Temporizador 2

Con el fin de repartir el tiempo de uso de la CPU emplearemos el temporizador 2 de IntegratorCP para producir una interrupción cada 100ms.

La configuración inicial del temporizador establece el modo de operación periódico, con contador de 32 bits, habilita las interrupciones(se lanzan cuando el contador llega a 0) y por último se carga el valor inicial en el contador y se habilita para que comience la cuenta.

Para hacer más flexible el empleo del temporizador programamos un API que nos permitiese configurar buena parte de los aspectos del temporizador desde código C. Los únicos aspectos fijos son el modo de operación periódico y la anchura de 32 bits del contador, ambas características se configuran en la inicialización del temporizador y no se proporcionan funciones para alterarlos.

Con este API se puede cargar un valor en el contador, valor que se especifica en milisegundos por comodidad. La gestión de las interrupciones también se realiza por medio del API, ya que va más allá de habilitarlas e inhabilitarlas. Cuando se produce una interrupción se pone a uno el bit 0 del *Raw Interrupt Status Register*, al estar habilitadas las interrupciones el pin de salida de las interrupciones valdrá uno. Este valor se anula escribiendo en *Interrupt Clear Register*, acción que hay que realizar cuando se identifique que este dispositivo lanzó la interrupción para evitar que interrumpa permanentemente.

6.7.2. PIC

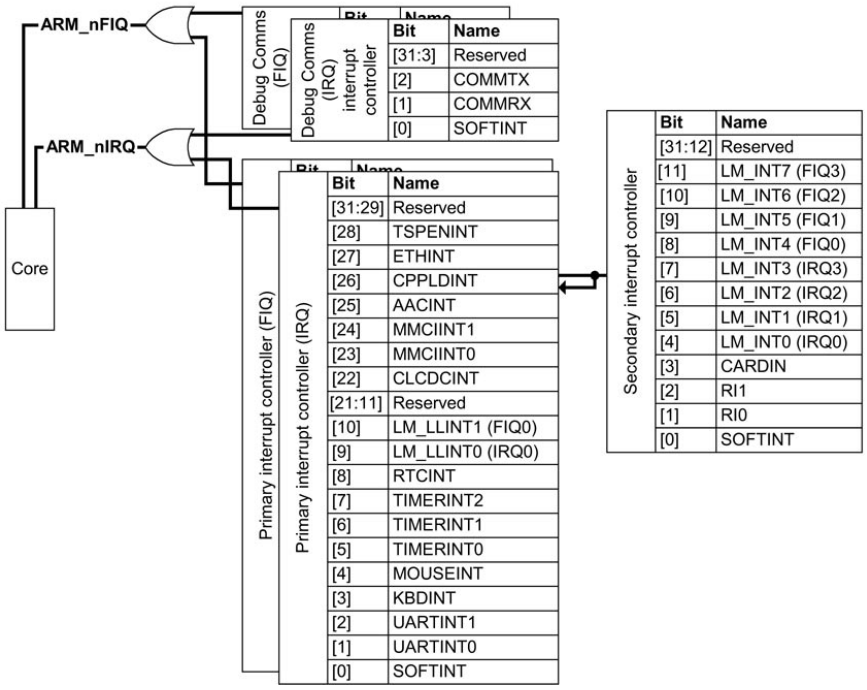

FIGURA 6.3: Estructura del controlador primario de interrupciones

Para que las interrupciones periódicas generadas por el temporizador lleguen al núcleo de la CPU, es preciso configurar el controlador de interrupciones al que está conectado. Los temporizadores de IntegratorCP están conectados al PIC (*Primary Interrupt Controller* o controlador primario de interrupciones).

La estructura de gestión del PIC permite un control absoluto sobre las interrupciones. Gracias a ella podemos hacer que una interrupción generada por un dispositivo se envíe al núcleo como una IRQ, FIQ o interrupción *softwa-*

re. La decisión tomada fue enviar directamente interrupciones *hardware* al núcleo (IRQ). Para ello activamos el bit 7 del registro PIC_IRQ_ENABLESET. Al igual que en el caso del temporizador, es preciso escribir en el bit adecuado de PIC_IRQ_ENABLECLR una vez identificada la fuente de la interrupción para que no se interrumpa de forma continua a la CPU. Es por esto que se implementó un API para poder gestionar este dispositivo directamente desde C al igual que se hizo con el temporizador.

Por último, las interrupciones tienen que ser activadas en el núcleo o la configuración llevada a cabo previamente no tendrá efecto. Al emplear únicamente interrupciones IRQ, tan sólo activaremos este tipo de interrupciones en el registro de control del procesador, escribiendo un 0 en el bit I, que se corresponde con el bit 7 del CPSR.

6.7.3. MMU

La Unidad de Gestión de Memoria no se encuentra dentro del núcleo ARM, si no que se conecta como el coprocesador 15 (System Control Co-processor). Gran parte de la inicialización y gestión de la memoria virtual se realiza a través de las tablas de páginas presentes en memoria, sin embargo, para conseguir un funcionamiento correcto del sistema es necesario configurar algunos parámetros de la MMU.

Para la implementación de Minix@ARM hemos realizado una configuración básica de esta unidad, ya que permite un uso muy flexible de todas sus características, de las cuales no todas resultan necesarias o útiles para el *microkernel* de Minix.

Para un funcionamiento mínimo de un sistema con memoria virtual, es necesario, en primer lugar, crear las tablas de páginas y, en segundo lugar, configurar el puntero a la tabla maestra (Translation Table Base, TTB), ya que las tablas de páginas pueden estar ubicadas en cualquier zona de la memoria. El siguiente elemento a configurar es la caché de traducciones, o TLB: ha de inicializarse vacía, y eliminar todas las entradas que tenga en cada cambio de contexto.

Además de activar estos mecanismos, hemos optado por desactivar las cachés de datos e instrucciones ya que en ARM, aunque permiten un alto grado de configuración, las cachés de datos requieren bastante gestión por parte del sistema operativo.

6.8. Definiciones

6.8.1. Mapa de memoria

Una de las primeras tareas a la hora de desarrollar las capas inferiores de un sistema operativo es el diseño y especificación de un mapa de memoria adecuado. Esto es especialmente relevante en el mundo de los dispositivos empotrados, donde muchos dispositivos están colocados directamente en regiones de memoria fijas. En nuestro caso, resultó aún más complicado, ya que queríamos mantener U-Boot funcionando, para lo que debíamos respetar su espacio de memoria. Sin tener un conocimiento profundo de las necesidades que habríamos de afrontar, se propuso un mapa de memoria inicial, sin siquiera tener en cuenta la memoria virtual. Posteriormente, según se fue adquiriendo mayor comprensión del problema y de las ventajas e inconvenientes que aportaba una estructura u otra, se llegó al mapa de memoria que utilizamos actualmente en el proyecto.

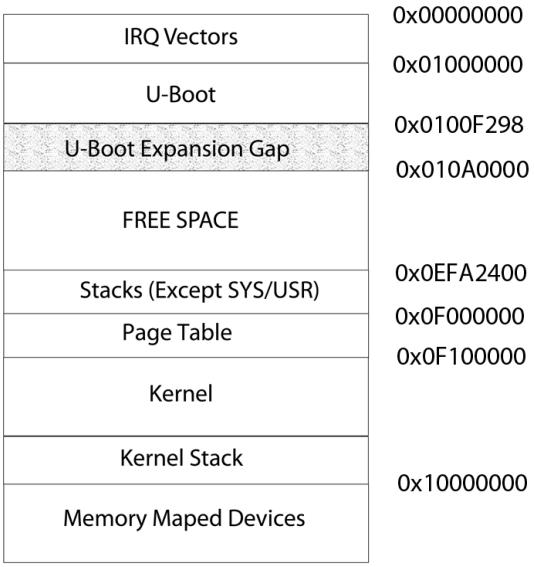

FIGURA 6.4: Mapa de memoria

La imagen es bastante intuitiva, pero hay un par de detalles que merece la pena resaltar. En primer lugar, el hueco denominado *U-Boot Expansion Gap* es un pequeño espacio sin utilizar, con la idea de que si la versión de U-Boot

utilizada es diferente y ocupa un espacio mayor en la memoria, no haga falta cambiar toda la distribución. En segundo lugar, la región *Stacks*, contiene las pilas de todos los modos menos del modo Usuario y del modo Sistema. La pila del modo Sistema se encuentra en la región *Kernel Stack*, situada justo antes de los dispositivos y creciendo hacia posiciones bajas de memoria. Esta región (*Stacks*), ocupa 372KB, y todas sus pilas crecen hacia posiciones bajas de memoria. Tanto *Page Table*, como *Kernel Stack*, y *Kernel*, pertenecen al espacio de memoria del *kernel*.

La distribución de la memoria física (Figura 6.4) no es la representación que se muestra a los procesos. Todos los procesos del sistema, ven un mapa de memoria virtual como el que se muestra a continuación.

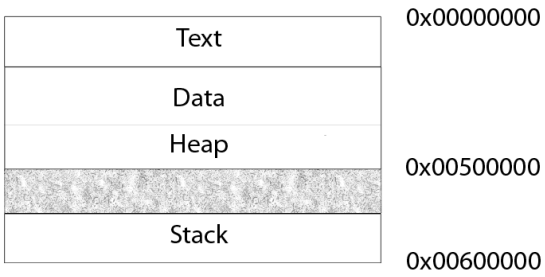

FIGURA 6.5: Imagen de memoria virtual del proceso

Este espacio, uno por cada proceso, será colocado dentro de la región denominada *FREE SPACE* en el mapa de memoria física. De esta forma, cada proceso verá que dispone de un espacio de memoria de tamaño igual al del hueco *FREE SPACE* -unos 220MB-, independientemente de que existan más procesos en el sistema. Actualmente, como muestra la imagen, cada proceso cuenta con un espacio de memoria virtual de 6MB, para facilitar las pruebas y la comprensión del trabajo. Con el mecanismo de memoria virtual nos aseguramos de que ningún proceso de usuario acceda a las regiones de memoria que no le pertenecen, incluyendo tanto las de otros procesos como las regiones del *kernel*, de dispositivos o de U-Boot.

6.8.2. Stackframe

El marco de pila (o *stackframe*) de Minix 3 era utilizado para almacenar la información contenida en los registros del procesador en la tabla de procesos

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter
r14		LR	Link Address / Scratch Register
r13		SP	The Stack Pointer. Lower end of current stack frame
r12		SB	Scratch reg. / new sb in inter-link-unit-calls
r11	v8		Frame Pointer
r10	sl/v7		Stack limit / Variable-register 7
r9		v6/SB/TR	Static base Register / Register Variable 6
r8	v5		Variable-register 5
r7	v4		Variable register 4
r6	v3		Variable register 3
r5	v2		Variable register 2
r4	v1		Variable register 1
r3	a4		Argument 4 / scratch register 4
r2	a3		Argument 3 / scratch register 3
r1	a2		Argument 2 / result / scratch register 2
r0	a1		Argument 1 / result / scratch register 1

TABLA 6.3: Minix@ARM Stackframe

durante una interrupción o cambio de contexto. En Minix@ARM tiene la misma utilidad, pero dado que depende directamente de los registros del procesador, comentados en detalle en la Sección **??**, es completamente diferente al que utiliza Minix 3 en Intel.

Hemos diseñado un *stackframe* respetando el formato EABI y el ARM Procedure Call Standard (PCS). El formato definido por el PCS, junto a la información que EABI sugiere que debe haber en cada registro se muestra en la Tabla 6.8.2.

Basándonos en esta información, utilizamos una estructura como la siguiente para almacenar en memoria el estado del procesador durante el cambio de contexto.

```
struct stackframe_s { /* proc_ptr points here */
reg_t  a1;
reg_t  a2;
```

```
reg_t  a3;
reg_t  a4;
reg_t  v1;
reg_t  v2;
reg_t  v3;
reg_t  v4;
reg_t  v5;
reg_t  sb;
reg_t  sl;
reg_t  fp;
reg_t  ip;
reg_t  sp;
reg_t  lr;
reg_t  pc;
reg_t  cpsr;
reg_t  lr_irq; // for eficiency
};
```

Esta estructura ha sustituido a la homónima de Minix 3 en la tabla de procesos. El motivo de incluir el valor del registro de enlace LR (lr_irq), en el momento de entrar en la rutina de tratamiento de IRQ, es únicamente por simplicidad en la rutina de cambio de contexto y eficiencia. Para retornar después del cambio de contexto, se debe copiar el CPSR almacenado en el *stackframe* del nuevo proceso a ejecutar en el SPSR del modo IRQ. Además, debemos colocar en LR_IRQ el valor que debe tener el PC del proceso interrumpido cuando sea reanudado. Puesto que se accede al CPSR del *stackframe* con un *offset*, no cuesta mucho traerse también el siguiente valor en memoria, que corresponde al LR, y con una única instrucción se puede cargar el SPSR y el LR_IRQ, para que se pueda retornar de la interrupción con la instrucción `movs` por defecto.

6.9. Minix

El proceso de adaptación del código fuente presenta otros inconvenientes además de adaptar el código dependiente de la arquitectura. Dentro del *kernel* se integran características que, por razones de desarrollo, no habían sido portadas junto a las que si lo habían sido. Estas partes debían ser aisladas con sumo cuidado para no alterar la funcionalidad. Este proceso, laborioso y propenso a errores, ha estado muy presente durante el desarrollo, por ser este un proceso gradual. Cada nueva característica incorporada desde el código de Minix, ha pasado por un proceso en el cual se pretendía traer al *port* la mayor parte de

funcionalidad sin introducir errores ni más partes dependientes que ralentizaran el desarrollo y que, a su vez, no entraran en conflicto con la base de código ya existente. Es por esto que el paso de mensajes no se considera finalizado, ya que este proceso no pudo ser concluido. El sistema completo establece unas dependencias internas que es preciso minimizar y aislar para poder usar el código fuente de Minix.

La escasa documentación de Minix y la existencia de numerosas partes implementadas con el fin de obtener un mayor rendimiento, dificultan en muchos casos la labor de entender qué se está haciendo en cada parte.

Capítulo 7

Entorno de Desarrollo

7.1. Qemu

Qemu [11] es un emulador de procesadores basado en la traducción dinámica de binarios -realiza la conversión del código binario de la arquitectura anfitriona a código ejecutable en la arquitectura huésped-. Esta forma de trabajar le permite adquirir un buen rendimiento en la emulación. Qemu dispone de dos modos de funcionamiento:

- Emulación en modo usuario (*User mode emulation*): En este modo, Qemu puede lanzar procesos compilados para una CPU en otra distinta. Esto es útil, por ejemplo, para lanzar el API de WINE [14], para facilitar la compilación cruzada (cross-compilation) y la depuración cruzada (cross-debugging).
- Emulación en modo sistema (*Full system emulation*): En este modo, Qemu emula un sistema completo (por ejemplo, un PC), incluyendo uno o varios procesadores y dispositivos. Se puede utilizar para arrancar diferentes sistemas operativos sin reiniciar el PC o para depurar código de *kernel*.

Para emulación en modo usuario, tan solo se encuentran soportadas algunas CPUs (x86, PPC, ARM, Mips32, Sparc32/64 y ColdFire), pero ningún sistema completo. En modo sistema, sin embargo, existen varias plataformas completas disponibles (ARM IntegratorCP, ARM VersatileAB, MIPS Magnum, ISA PC ...).

Además, Qemu dispone de facilidades de virtualización, que es de hecho la forma más común de utilizarlo. Esta máquina virtual puede ejecutarse en

cualquier tipo de microprocesador o arquitectura. Esta programa no dispone de interfaz gráfica (GUI), pero existen paquetes para Windows, GNU/Linux y MAC OS X, que dotan de una GUI a Qemu.

Qemu es *Open Source* y su código se encuentra licenciado bajo LGPL y GPL.

Para el desarrollo de nuestro proyecto, hemos utilizado Qemu en modo *system emulation* para obtener una placa IntegratorCP emulada con sus periféricos. Además, hemos utilizado su servidor GDB para depurar nuestro código y el código de U-Boot vía Ethernet, utilidad indispensable para llevar a cabo cualquier desarrollo sobre plataformas de este tipo.

7.2. U-Boot

U-Boot [15] es un *bootloader* universal que provee *firmware* para un nutrido grupo de arquitecturas y placas, con todo su código licenciado bajo licencia GPL. Es ampliamente utilizado como cargador de arranque por varios fabricantes de dispositivos empotrados -por ejemplo, es el que trae la BeagleBoard [16]-.

Pese a las similitudes, en un dispositivo empotrado, el rol del *bootloader* es más complicado, puesto que estos sistemas no disponen de una BIOS para realizar la configuración inicial del sistema. La inicialización a bajo nivel de los microprocesadores, controladores de memoria y otro *hardware* específico de la placa varia de placa a placa y de CPU a CPU. Todas estas inicializaciones deben llevarse a cabo antes de que ninguna imagen de *kernel* pueda ejecutarse.

Como mínimo, un cargador para un sistema empotrado ha de proveer las siguientes características:

- Inicialización del *hardware*, especialmente el controlador de memoria.
- Facilidades para paso de parámetros a *kernels* Linux.

Además, muchos *bootloaders* proveen características orientadas a simplificar el desarrollo:

- Lectura y escritura de posiciones arbitrarias de memoria
- Carga de nuevas imágenes binarias a la RAM de la placa vía puerto serie o Ethernet

- Copia de binarios de memoria RAM a memoria FLASH

U-Boot incorpora todas estas características, además de algunas otras ventajas. Sin embargo, todas las funcionalidades orientadas a *kernels* Linux nos han sido indiferentes para el desarrollo del proyecto. Además, la inicialización para IntegratorCP no es completa.

En cuanto a la memoria, U-Boot comienza en la FLASH, activa la RAM y la *mapea* en la posición cero de memoria, para posteriormente copiarse a esa dirección de RAM y desactivar la FLASH. No obstante, la inicialización de la MMU no se realiza y se mantiene desactivada. Ha sido por tanto necesario inicializar y activar la MMU para poder dar soporte a la memoria virtual.

Respecto al resto de dispositivos, puesto que se pretendía que U-Boot siguiera funcionando tras la ejecución de Minix, hubo ciertos aspectos a tener en cuenta. Si bien U-Boot inicializaba un temporizador, lo utilizaba internamente para contar el tiempo de ejecución y para llevar a cabo ciertas tareas. De este modo, nos vimos forzados a inicializar un segundo temporizador, para utilizarlo dentro de Minix.

Por parte de las interrupciones, U-Boot configura unos vectores de interrupción básicos -la mayoría son bucles infinitos-, ya que es indispensable para el correcto funcionamiento de las placas ARM. Sin embargo, la inicialización que realiza de las interrupciones es similar a la de la MMU; simplemente las inabilita en el *Core*, lo que evita que las posibles interrupciones lanzadas por los dispositivos alcancen la CPU y sean tratadas. Por lo tanto, tuvimos que escribir nuestros propios vectores de interrupción -y moverlos en memoria, desde la posición de nuestro *kernel* hasta las posiciones reservadas a los vectores de interrupción-, configurar las interrupciones en el *Core* y en el PIC y programar el temporizador para contar a la frecuencia deseada.

Tras la breve introducción, hay que decir que U-Boot nos ha facilitado mucho el desarrollo. Lo hemos utilizado asiduamente para cargar nuestros binarios vía TFTP, examinar posiciones de memoria, saltar a los puntos de entrada de los binarios y muchos otros pequeños detalles. Es también importante reseñar, que disponer del código de U-Boot nos ha sido de mucha utilidad para comprender el funcionamiento de algunos dispositivos y para encontrar errores en el código de bajo nivel de nuestro *kernel* de Minix.

7.3. *Toolchain*

Un *toolchain* (cadena de herramientas) es un término que hace referencia a una serie de herramientas de programación que forman un sistema integrado, utilizado normalmente para crear un determinado producto -normalmente otro programa o sistema informático-. Existe un amplio espectro de entornos de desarrollo para arquitecturas ARM, si bien la mayoría de ellos tienen unos precios excesivos para las licencias de uso. Desde nuestro punto de vista, las facilidades que ofrecen compensa el precio, ya que aceleran el desarrollo de la aplicación objetivo y sobretodo facilitan la depuración y seguimiento de errores. Debido al coste de estos entornos se optó por una opción basada en *software* libre. El *toolchain* compuesto por las herramientas desarrolladas por el proyecto GNU es comúnmente conocido con GNU *toolchain*, indispensable para el desarrollo del *kernel* Linux, BSD y *software* para sistemas empotrados. Este *toolchain* está disponible para diferentes arquitecturas, la versión para ARM se denomina GNUARM [17].

El proceso de compilación de un *toolchain* es complejo y requiere de varias etapas o fases en las que se van compilando las diferentes piezas de *software* que integran el mismo. Es por eso que es frecuente utilizar versiones precompiladas para la arquitectura objetivo.

Durante las fases tempranas del proyecto utilizamos la última versión compilada del *toolchain* GNUARM. Debido a que esta versión es bastante antigua, existían diferentes fallos -principalmente en el depurador, y alguna sutileza del enlazador- que dificultaban o imposibilitaban el desarrollo. Investigando alternativas gratuitas llegamos al *toolchain* de CodeSourcery [18]. CodeSourcery desarrolla en colaboración con ARM, Ltd. [19] mejoras sobre el *toolchain* GNU para procesadores ARM y provee con regularidad versiones revisadas del mismo. Incluye las últimas versiones de GCC, binutils, glibc... Este es el *toolchain* que se ha utilizado finalmente para desarrollar este proyecto.

7.4. Desarrollo Colaborativo: Launchpad, Bazaar y Dokuwiki

En cualquier proyecto medianamente extenso, llevado a cabo por más de una persona, se requiere algún método de sincronización y compartición de información entre los desarrolladores. Para llevar a cabo nuestro proyecto, decidimos utilizar Launchpad. Launchpad [20] es un sitio web que permite el

alojamiento de código y una plataforma de desarrollo de *software* colaborativa. Está desarrollada y mantenida por Canonical [21], compañía responsable del desarrollo de Ubuntu [22], y apoya el desarrollo de *software* libre. Incluye gran variedad de funciones, entre las que se destacan repositorios de código, seguimiento de errores, gestión de documentación, *roadmap* del proyecto y sus hitos importantes...

Para la gestión del código, hemos utilizado Bazaar [23]. Bazaar es una herramienta de control de versiones distribuida (DCVS) que realiza un seguimiento de los cambios que tú y otros desarrolladores hacéis sobre una serie de ficheros, para ofrecerte *snapshots* de cada etapa de su evolución. Usando esta información, Bazaar puede mezclar tu trabajo con el de otra gente de manera eficiente. Además de las ventajas inherentes a su modo distribuido -frente a CVS centralizado como subversion-, Bazaar es el sistema de control de versiones utilizado por Launchpad, es *software* libre bajo GPLv2, multiplataforma y muy sencillo de utilizar.

Dokuwiki [24] ha sido la alternativa escogida para gestionar la documentación del proyecto, así como las ideas y notas que se han ido tomando a lo largo de su desarrollo. Dokuwiki es un wiki sencillo de usar, cuyo objetivo principal es la creación de documentos de cualquier tipo. Está orientado a equipos de desarrollo, grupos de trabajo y pequeñas empresas. Tiene una sintaxis simple pero completa, que permite garantizar que los ficheros de datos serán legibles fuera de la *wiki* y facilita la creación de textos estructurados. Toda la información se almacena en ficheros de texto plano, por lo que no necesita de ninguna base de datos para funcionar. Nos ha sido especialmente útil, su característica de control de revisiones sobre las páginas, permitiéndote ver versiones antiguas de las mismas, cambios recientes... Para nuestro proyecto, hemos configurado un Dokuwiki en un servidor privado, donde se ha alojado la documentación relevante para el proyecto. El enlace puede encontrarse en la sección de bibliografía [31].

Capítulo 8

Dificultades

Desde las primeras reuniones y planificación del proyecto quedaba claro que se trataba de una idea ambiciosa y compleja. Sin embargo, el grupo estaba bastante motivado ya que el proyecto nos daba una gran oportunidad para reforzar conceptos que durante la carrera no habían sido estudiados en profundidad y aprender de la experiencia de enfrentarnos a un sistema operativo de cierta complejidad, por lo que nos pusimos manos a la obra.
La primera idea consistía en desarrollar sobre la placa utilizada en la asignatura de Laboratorio de Estructura de Computadores (LEC). Comenzamos a realizar algunas pruebas de concepto, a cargar algunos códigos y a leer la documentación de la plataforma. Hacia el final de la fase de pruebas nos dimos cuenta de que la placa que había sido sugerida como plataforma de desarrollo carecía de MMU, por lo que la implementación de los mecanismos *hardware* de protección de memoria y memoria virtual habrían de emularse mediante *software*[1], lo que hubiera incrementado notablemente la complejidad del proyecto.

Descubierto el problema tuvimos que buscar una nueva plataforma, que cumpliera con los requisitos necesarios para el desarrollo, y que además fuera económica, ya que cada fabricante implementa sus propias herramientas de desarrollo y depuración, comercializándolos a precios prohibitivos dado el presupuesto muy limitado que estaba a nuestra disposición. Uno de nuestros objetivos secundarios era poder trabajar en un entorno de desarrollo basado en *software* libre. Tras pasar un tiempo buscando una plataforma *hardware* que cumpliese con estas especificaciones, dimos con la plataforma BeagleBoard [16], una placa de bajo coste (alrededor de 150$), basada en la implementación Cortex A-8 de ARM, integrada en una placa con diversos dispositivos. Se informó a nuestros directores de proyecto, que procedieron a realizar el pedido. La Bea-

[1] al igual que hace uclinux en dispositivos que carecen de este *hardware*

gleBoard tardó más de un mes en llegar, por lo que no pudimos ponerle las manos encima hasta diciembre. Durante ese tiempo, no obstante, aprovechamos para leer la documentación y aprender los nuevos dispositivos y las características dependientes de la implementación, ya que ciertos componentes, como los temporizadores, por ejemplo, de cada placa son distintos, así como las características del procesador, los coprocesadores... Cuando recibimos el material, nos informaron de que el dispositivo JTAG -necesario para depuración *hardware*- tardaría aún en llegar. En esta situación, aprovechamos para instalar una implementación Debian adaptada para esta plataforma en la placa y comprobar las posibilidades que brindaba el *hardware*, ya que las primeras etapas de desarrollo están tan ligadas al *hardware* que sin JTAG no era posible avanzar. El Flyswater [25] (JTAG), llegó a finales de enero. Previamente habíamos comprobado que el *software* de depuración a través de JTAG (OpenOCD [26]) fuera compatible con el dispositivo Flyswater. Con todo el material disponible, comenzamos a montar el entorno de desarrollo. Conseguimos conectarnos con OpenOCD a través del Flyswater a la placa, sin embargo, cada vez que tratábamos de activar el procesador o depurar alguna instrucción, se producía un fallo de segmentación en OpenOCD que nos impedía continuar. Tratando de solucionar este error, pudimos comprobar que la documentación disponible era bastante pobre, y a través de la lista de correo de los programadores de OpenOCD y otros foros, nos informaron de que el soporte para el Cortex-A8 estaba aún en fase de desarrollo; era posible conectar y ver el procesador, pero no activarlo ni iniciar la depuración. Después de averiguar esto, los desarrolladores de OpenOCD finalmente actualizaron la documentación para informar del primitivo estado de desarrollo del *software* para la conexión con BeagleBoard, hasta comienzos de septiembre de 2009, cuando fue publicada una versión con una funcionalidad básica, pero que hubiera sido suficiente para desarrollar este trabajo.
Tras esta serie de inconvenientes, habíamos llegado a principios de abril y no teníamos nada. Volvimos a buscar plataformas y entornos de desarrollo. Encontramos varios IDE muy prometedores, pero con precios bastante elevados que tuvimos que descartar. Asimismo localizamos otras placas que podían servir para el proyecto, pero no había mucho más dinero que invertir. Nos dimos cuenta de que no quedaba otra opción que utilizar algún entorno de simulación para tratar de llevar a término el proyecto. Pasamos por tanto a recopilar información acerca de candidatos de emulación de *hardware*. Pasado un tiempo valorando ventajas e inconvenientes de cada una de las opciones, finalmente optamos por Qemu, un *software* de emulación de procesadores.

Las opciones de Qemu para emular plataformas ARM completas incluían

principalmente VersatileAB e IntegratorCP. Investigamos las dos, leímos documentación de todas ellas y probamos a emularlas en Qemu con diferentes procesadores. Dentro de los procesadores que Qemu permitía emular, no todos disponían de MMU, por lo que hubo que seleccionar y buscar la combinación adecuada. Más adelante, nos dimos cuenta de que también se debía tener en consideración U-Boot, ya que las fuentes no compilaban correctamente para todas las combinaciones de placa y procesador. Iniciamos las emulaciones con VersatileAB, ya que disponía a priori de mayor número de dispositivos. Sin embargo, encontramos mayor documentación y ejemplos de código sobre IntegratorCP, por lo que acabamos cambiando a esta última.

Como se puede comprobar leyendo el Capítulo 7 sobre el entorno de desarrollo y el Capítulo A, sobre su configuración, no es asunto sencillo poner en marcha todo el equipo necesario. Puede parecer que los pasos a seguir son simples y que funciona correctamente al primer intento, pero eso es sólo porque nosotros ya hemos probado todas las combinaciones erróneas y hemos descubierto los parámetros de configuración adecuados que hacen de pegamento entre todas las capas necesarias. Al final, obtuvimos un entorno de desarrollo plenamente operativo para mediados de mayo, momento en el que empezamos a programar el proyecto.

La conclusión que se puede extraer de todo esto es que la falta de planificación y de presupuesto han llevado a un desperdicio del tiempo disponible para desarrollar el proyecto. También hay que mencionar que la lectura de documentación para los diferentes entornos nos ha proporcionado un mayor conocimiento de los dispositivos ARM, sus variantes y características. A pesar de haber conseguido bastante en los pocos meses disponibles, sentimos que se podría haber avanzado mucho más si se hubiera dispuesto de un entorno funcional desde el principio del curso y no hubiera sido necesaria investigación sobre las opciones además de sobre el material del proyecto.

Capítulo 9

Trabajo futuro

A pesar del avance significativo que se ha llevado a cabo durante el proyecto, aún queda mucho camino por recorrer para tener un sistema operativo completamente funcional. Es recomendable comenzar por las secciones que han quedado inconclusas o cuyo soporte es precario. En primer lugar, el paso de mensajes es una funcionalidad crítica para el *microkernel* que, pese a estar prácticamente implementada, no ha sido completamente probada y puede que falte alguna sección por integrar. Por lo tanto, el primer paso a dar en la continuación del proyecto es la finalización del paso de mensajes.

El paso de mensajes es el mecanismo de comunicación del *microkernel* Minix. Existen dos tareas dentro del *kernel* que son necesarias para el mismo: *Clock Task*, y *System Task*. La primera se encarga de las interrupciones del temporizador y de los cambios de contexto. La segunda se encarga principalmente proveer a los controladores y servidores por encima de ella una serie de llamadas al sistema privilegiadas. Actualmente, *Clock Task* está funcionando, si bien hemos tenido que saltarnos el mecanismo de envío y recepción de mensajes por no tenerlo finalizado. *System Task*, por el contrario, no está funcionando. Su código está integrado en el *kernel* y compila correctamente, pero no se ha dado soporte a todas las funcionalidades que requiere. Debido a estos aspectos, es conveniente implementar el paso de mensajes, que junto con las dos tareas del *kernel* finalizarían lo que Minix denomina capa uno.
La gestión de interrupciones es funcional y sencilla. Sin embargo, está programada a medida. Sería conveniente, por tanto, acondicionar el código ensamblador que gestiona las interrupciones, principalmente para poder aceptar otras fuentes de IRQ además del temporizador. Esta tarea resulta de especial sencillez, ya que el código está comentado y organizado de manera comprensible.

Estos pasos completarían la adaptación del *microkernel* a la arquitectura ARM. Aún así, el problema de las dependencias *hardware* aparecerá más adelante durante el desarrollo por varios motivos. El primero es que cada placa tiene una serie de dispositivos diferentes, cuyos controladores deben ser programados de nuevo, así como un mapa de memoria distinto, que debe ser adaptado. El problema del mapa de memoria es menor, ya que está parametrizado mediante constantes de fácil modificación. El problema de los dispositivos es de mayor envergadura. Por lo tanto, recomendamos que para la continuación del proyecto se utilice la misma plataforma de desarrollo, IntegratorCP con el núcleo por defecto de Qemu. Además de estos aspectos muy dependientes del *hardware*, hay algunos aspectos relativos a la gestión de memoria virtual que pueden refinarse. En primer lugar, la emulación de segmentos mediante páginas ya implementada no permite aprovechar totalmente las ventajas de un sistema paginado, ya que se asignan páginas físicas consecutivas en memoria. Un primer paso para mejorar este aspecto es añadir la posibilidad de que los segmentos sean direcciones consecutivas en memoria virtual y puedan estar dispersos en memoria física. Las estructuras de datos para manejar las regiones (el equivalente a los segmentos de Minix) permite asignar a un segmento páginas ubicadas en cualquier espacio libre de memoria, siendo necesario modificar únicamente el algoritmo de asignación de espacio.

Con la base firmemente acabada, comenzaría el proceso de añadir capas sobre el *microkernel*. Las primeras capas a considerar serán los servidores. Dentro de los servidores, consideramos prioritario y de mayor relevancia el servidor de ficheros. Esto es debido a que la gestión de las llamadas al sistema se realiza en parte dentro del mismo. El mecanismo de llamadas al sistema es imprescindible para permitir la ampliación del proyecto. El servidor de procesos debería ser el objetivo siguiente, ya que la gestión de procesos y de memoria está entrelazada en su interior. El paso lógico siguiente será completar los servidores y comenzar el desarrollo de los diferentes controladores de dispositivo. En la realidad, el desarrollo de ambos está ligado, ya que el servidor de disco puede necesitar controladores para dispositivos de almacenamiento. Esta será la etapa más tediosa, pero no comprende una complejidad excesiva.

Llegados a este punto -aunque bien podría desarrollarse en paralelo-, es recomendable tratar de dar soporte ELF a Minix 3. Esto requiere *portar* un compilador que genere ELF o bien compilar en una máquina externa. La segunda opción es válida para el desarrollo del sistema operativo, pero una vez acabado, es necesario introducir un compilador con soporte ELF de cara a facilitar el desarrollo a los programadores. Hemos investigado ligeramente esta opción,

pero debido a la complejidad aparente, nuestra recomendación es que no se aborde hasta la última etapa del proyecto.

Es necesario hacer una última puntualización, referente al monitor de Minix. El monitor ha sido ignorado durante el desarrollo del proyecto ya que es excesivamente dependiente de la arquitectura x86. Sin embargo, se consideró la posibilidad de utilizar U-Boot para realizar en ARM las funciones del monitor en x86. La implementación actual confía la inicialización de parte de los dispositivos a U-Boot. En este sentido, es posible tomar dos decisiones, prescindir de U-Boot e implementar dicha parte de la inicialización, o bien finalizar y ampliar el soporte a U-Boot, permitiendo retornar al mismo desde Minix@ARM al igual que hace el monitor con la versión de x86. Como ya se ha comentado, esta segunda opción ha sido tenida en cuenta y existe código que proporciona cierto soporte para su continuación. El estado de U-Boot se guarda previamente al inicio del *microkernel*. De cara a poder regresar a U-Boot, es preciso restaurar sus vectores de interrupción, así como volver al modo SVC y copiar de vuelta a la pila SVC el contenido que tenía cuando se inició nuestro *kernel*. Además, es necesario inhabilitar las interrupciones en el núcleo y desactivar la MMU antes de cederle el uso completo de la CPU.

Apéndice A

Configuración del Entorno

En este anexo vamos a detallar la manera de preparar un entorno de desarrollo para que todo aquel que lo desee pueda contribuir al proyecto o hacer sus propias investigaciones. Se tomará como base un sistema basado en debian, aunque la mayor parte debería ser idéntica en cualquier distribución.

A.1. Instalación

A.1.1. Software Necesario

- Qemu
- CodeSourcery ARM Toolchain
- U-Boot
- tftpd
- bridge-utils
- uml-utilities
- dnsmasq

Qemu nos servirá para emular una IntegratorCP completa, a la que nos conectaremos vía Ethernet, para lo que necesitaremos configurar interfaces y *bridges* mediante `brigde-utils` y `uml-utilities`. Sobre la placa emulada -IntegratorCP-, se ejecutará U-Boot, que nos dará un pequeño cargador de arranque como se explica en la sección U-Boot 7.2. Para cargar nuestros binarios, U-Boot ofrece un cliente TFTP. Para utilizarlo necesitamos configurar un servidor TFTP (`tftpd`) en nuestro computador que sirva los binarios a través

de este protocolo. Como es obvio, para que el anfitrión y el huésped puedan comunicarse ambos deben tener direcciones IP asignadas; utilizaremos `dnsmasq` para asignar por DHCP una IP a Qemu+U-Boot y asignarle además otros parámetros de configuración, como por ejemplo qué imagen debe cargar por defecto.

Utilizaremos el toolchain para compilar tanto U-Boot, como nuestro código del proyecto y cualquier ejemplo que queramos cargar, así como para depurar el código cargado.

A.1.2. Qemu, tftpd, bridge-utils y uml-utilities

Para su instalación, basta con abrir una terminal y teclear lo siguiente como `root` (o con `sudo` si el usuario `root` está deshabilitado):

```
root@Orion:~# aptitude install qemu kqemu tftpd \
 bridge-utils uml-utilities
```

Necesitaremos cargar los binarios en la IntegratorCP (Qemu) desde U-Boot. Para poder hacerlo, necesitaremos acceder desde U-Boot via tftp a nuestro PC. Configuraremos la red y `tftp` de la manera adecuada. En primer lugar, editaremos el fichero `/etc/inetd.conf` para indicarle la carpeta que debe mostrar como raíz del servidor TFTP:

```
tftp dgram udp wait nobody /usr/sbin/tcpd \
    /usr/sbin/in.tftpd /opt/minix-arm/src/bin
```

Donde `/opt/minix-arm/src/bin` es el directorio que se exportará como raíz del servidor TFTP, y donde posteriormente colocaremos los binarios que queramos cargar. Debemos reiniciar el servicio para que los cambios tengan efecto:

```
root@Orion:~# /etc/init.d/openbsd-inetd restart
```

Qemu utiliza interfaces `tun/tap` para conectarse con el host, por lo que utilizaremos un *bridge* para poner la interfaz del *host* en el mismo segmento "físico" de red. Existen otras formas de hacerlo, pero esta es sencilla. Para empezar, editaremos el fichero `/etc/network/interfaces` y colocaremos la definición de nuestro nuevo *bridge*. Se muestra sólo la configuración del *bridge*.

```
auto br0
iface br0 inet static
address 192.168.3.121
netmask 255.255.255.0
```

```
network 192.168.3.0
broadcast 192.168.3.255
bridge_stp off
bridge_maxwait 0
bridge_fd 0
bridge_hello 0
```

Si quisiéramos añadir una interfaz al *bridge*, podríamos meterla en el fichero con la opción

```
bridge_ports ethX
```

o bien añadirla temporalmente mediante la orden:

```
root@Orion:~# brctl addif br0 ethX
```

Para no tener que hacer esto constantemente, Qemu ejecuta un *script* cada vez que arranca, al cual le pasa como parámetro la interfaz que va a utilizar. Este fichero se encuentra en `/etc/qemu-ifup` (si no existe debemos crearlo). El contenido del fichero debe ser el siguiente:

```
#!/bin/sh

echo "Executing /etc/qemu-ifup"
echo "Bringing up $1 for bridged mode..."
echo "Adding $1 to br0..."

/sbin/ifconfig tap0 up
/sbin/ifconfig br0 up
brctl addif br0 $1
```

El *script* debe tener los permisos adecuados:

```
-rwxr-xr-x 1 root root 202 2009-07-02 22:40 /etc/qemu-ifup
```

Si Qemu no se ejecutase como `root`, entonces el script deberá lanzar los comandos con `sudo`.

A.1.3. DNSmasq

Este paso se puede omitir, pero es más cómodo el desarrollo si se realiza la configuración como se describe en él. Para empezar, debemos instalar dnsmasq:

```
root@Orion:~/si# aptitude install dnsmasq
```

Editaremos el fichero `/etc/dnsmasq.conf` para dejar las siguientes lineas descomentadas (si no aparecen, añadidlas):

```
interface=br0
dhcp-range=192.168.3.241,192.168.3.250,12h
dhcp-boot=kernel.bin
```

Con esto U-Boot podrá obtener por DHCP los parámetros `serverip`, `ipaddr` y cargará por defecto la imagen llamada `kernel.bin` situada en el directorio raíz del servidor TFTP. U-Boot cargará la imagen `kernel.bin` en la posición que indique su parámetro de configuración (fijado en tiempo de compilación). Es necesario reiniciar el servicio para que los cambios tengan efecto:

```
root@Orion:~# /etc/init.d/dnsmasq restart
```

A.1.4. CodeSourcery *Toolchain*

Una vez descargado de la web oficial [27], basta con darle permisos de ejecución al `.bin`, ejecutarlo y seguir los pasos indicados en el instalador. Es importante añadir al `PATH` los binarios si queremos encontrarlos desde cualquier ruta del sistema, para esto editaremos el fichero `/.bashrc` y añadiremos al final la siguiente línea:

```
root@Orion:~# export PATH=$PATH:/opt/CodeSourcery/bin
```

Siendo `/opt/CodeSourcery/` la ubicación que se le ha dado al instalador para depositar el *toolchain*. Cada vez que abramos una terminal nueva, el fichero será leído y tendremos disponibles los binarios. Si no queremos cerrar la terminal actual, podemos cargar el fichero con la orden:

```
root@Orion:~# source ~/.bashrc
```

A.1.5. U-Boot

Podemos obtener la última versión estable del código de u-boot de la página web: `ftp://ftp.denx.de/pub/u-boot/`. Tras la descarga, crearemos un directorio donde irán a parar los archivos compilados de U-Boot para la plataforma que queramos (en nuestro caso IntegratorCP).

```
root@Orion:~/si# mkdir uboot-intcp
root@Orion:~/si# cd u-boot-1.3.4/
```

Necesitamos indicarle a U-Boot dónde se encuentran las herramientas de compilación para ARM. Además, hay que indicarle el directorio destino de la compilación. Lo haremos exportando las siguientes variables de entorno:

```
root@Orion:~/si# export CROSS_COMPILE=arm-none-eabi-
root@Orion:~/si# export BUILD_DIR=~/si/uboot/uboot-intcp
```

Si se ha configurado DNSmasq y se desea automatizar el proceso de carga del *kernel*, hay que modificar la configuración de U-Boot antes de proceder a la compilación. Para ello, editaremos el fichero de configuración de nuestra placa, `u-boot-1.3.4/include/configs/integratorcp.h`:

```
#define CONFIG_BOOTCOMMAND \
        "cp 0x24080000 0x0F100000 0x100000; bootm"
#define CFG_LOAD_ADDR 0x0F100000 /*default load address*/
```

Donde `0x0F1000000` es la dirección donde queremos que se cargue el *kernel*. Esta dirección es fija y no es configurable sin modificar el código fuente.

Para finalmente compilar U-Boot a nuestro gusto, realizamos los siguientes comandos:

```
root@Orion:~/si/u-boot-1.3.4# make cp_config
root@Orion:~/si/u-boot-1.3.4# make all
```

A.2. Utilizando el entorno

A.2.1. Ejecutando

Ahora la configuración del entorno está completa, y podemos proceder a ejecutarlo. En primer lugar, compilaremos el código del *kernel* y de los procesos de prueba. Para ello, debemos dirigirnos al directorio donde se encuentren las fuentes, y compilarlas:

```
root@Orion:~# cd /opt/si/minix-arm/src
root@Orion:/opt/si/minix-arm/src# make clean; make all
```

Si todo ha ido bien, se habrán compilado tanto el kernel como los cinco procesos de prueba. Los binarios deben encontrarse en `bin/` y los ejecutables (ELF) en `elf/`. Ahora procederemos a lanzar Qemu indicándole que ejecute U-Boot:

```
root@Orion:~/si/uboot-intcp# qemu-system-arm \
    -nographic -m 256 -M integratorcp -cpu arm926 -s \
    -net nic -net tap -serial stdio -kernel ./u-boot.bin
```

-nographic: Evita que se lance la ventana de qemu (por comodidad)

-m 256: Incrementa la memoria a 256MB. Es imprescindible al menos esta cantidad, ya que el kernel se carga en posiciones de memoria elevadas

-net nic -net tap: Indica a qemu que utilice un interfaz tap como tarjeta de red

-serial stdio: Redirige la entrada y salida del terminal serie de la placa emulada por Qemu a la entrada y salida estándar de GNU/Linux

-kernel ./fichero.bin: Indica a Qemu qué binario debe ejecutarse en la placa. Debe ser un binario puro (*raw binary*)

-cpu arm926: Indica qué CPU queremos emular

-M integratorCP: Indica la placa que queremos emular

-s: Arranca un servidor GDB y espera peticiones en el puerto 1234 de *localhost*

Si todo ha ido bien, tendremos delante el intérprete de U-Boot. Para ver una lista detallada de las órdenes que se pueden ejecutar, podemos utilizar *help*. En este punto, dependerá de si se ha configurado DNSmasq para usar DHCP o no.

Si se ha configurado DNSmasq:

```
Integrator-CP # dhcp
SMC91111: PHY auto-negotiate timed out
Using MAC Address 52:54:00:12:34:56
BOOTP broadcast 1
*** Unhandled DHCP Option in OFFER/ACK: 28
*** Unhandled DHCP Option in OFFER/ACK: 28
DHCP client bound to address 192.168.3.247
TFTP from server 192.168.3.240;
our IP address is 192.168.3.247
Filename 'kernel.bin'.
Load address: 0xf100000
Loading: ##########
done
Bytes transferred = 46100 (b414 hex)
```

Si no se ha configurado DNSmasq:

```
Integrator-CP # setenv ipaddr 192.168.3.247
Integrator-CP # setenv serverip 192.168.3.240
```

```
Integrator-CP # tftpboot 0x0F100000 kernel.bin
SMC91111: PHY auto-negotiate timed out
Using MAC Address 52:54:00:12:34:56
TFTP from server 192.168.3.240;
our IP address is 192.168.3.247
Filename 'kernel.bin'.
Load address: 0x0F100000
Loading: #
done
Bytes transferred = 46100 (b414 hex)
```

Donde `192.168.3.240` es la IP que le hemos dado al bridge `br0` anteriormente. `192.168.3.247` es la IP que le asignaremos al interfaz de U-Boot, y que debe estar en la misma subred que nuestro *bridge*.

Ahora, independientemente del método seguido, debemos cargar los procesos que ejecutará el *kernel*, ya que de momento no se dispone de una imagen unificada ni de un lector de ELF. Para ello, utilizaremos TFTP de la misma forma que con el *kernel*.

```
Integrator-CP # tft 0x0a000000 proc1.bin
SMC91111: PHY auto-negotiate timed out
Using MAC Address 52:54:00:12:34:56
TFTP from server 192.168.3.240;
our IP address is 192.168.3.247
Filename 'proc1.bin'.
Load address: 0xa000000
Loading: #
done
Bytes transferred = 684 (2ac hex)
Integrator-CP # tft 0x0b000000 proc2.bin
SMC91111: PHY auto-negotiate timed out
Using MAC Address 52:54:00:12:34:56
TFTP from server 192.168.3.240;
our IP address is 192.168.3.247
Filename 'proc2.bin'.
Load address: 0xb000000
Loading: #
done
Bytes transferred = 684 (2ac hex)
Integrator-CP # tft 0x0c000000 proc3.bin
SMC91111: PHY auto-negotiate timed out
```

```
Using MAC Address 52:54:00:12:34:56
TFTP from server 192.168.3.240;
our IP address is 192.168.3.247
Filename 'proc3.bin'.
Load address: 0xc000000
Loading: #
done
Bytes transferred = 684 (2ac hex)
Integrator-CP # tft 0x0d000000 proc4.bin
SMC91111: PHY auto-negotiate timed out
Using MAC Address 52:54:00:12:34:56
TFTP from server 192.168.3.240;
our IP address is 192.168.3.247
Filename 'proc4.bin'.
Load address: 0xd000000
Loading: #
done
Bytes transferred = 684 (2ac hex)
Integrator-CP # tft 0x0e000000 proc5.bin
SMC91111: PHY auto-negotiate timed out
Using MAC Address 52:54:00:12:34:56
TFTP from server 192.168.3.240;
our IP address is 192.168.3.247
Filename 'proc5.bin'.
Load address: 0xe000000
Loading: #
done
Bytes transferred = 684 (2ac hex)
Integrator-CP #
```

Una vez esté todo cargado llega el momento de decirle a U-Boot que ejecute nuestro *kernel*. La orden `go` es la que debemos utilizar. La dirección que se le pasa como argumento es la del punto de entrada del binario (la dirección de la función `main`).

```
Integrator-CP # go 0f1026e4
## Starting application at 0x0F1026E4 ...
```

```
|p_name|rts|prio|m_prio|ti_left|q_size|u_time|s_time|n_ready|
|-------------------------------------------------------------|
|  p1  | 0 | 002| 002  |  000  |  001 | 00001| 00000| NULL  |
|_____________________________________________________________|

                ****************************
                * Proceso expropiado =>  p1 *
                * Siguiente  proceso =>  p1 *
                ****************************
```

Para averiguar cual es el punto de entrada de un binario dado, podemos utilizar la orden `objdump`, como se muestra en la siguiente linea:

```
root@Orion:~/si/minix-arm/src# arm-none-eabi-objdump \
    -d elf/kernel.elf | grep -i main
f1026e4 <main>:
```

A.2.2. Depurando

Abrimos un nuevo terminal, y vamos al directorio que contiene el correspondiente fichero ELF. Una vez allí, ejecutamos gdb.

```
root@Orion:~/opt/minix-arm/src/elf# arm-none-eabi-gdb \
    ./kernel.elf
GNU gdb (Sourcery G++ Lite 2009q1-161) 6.8.50.20081022-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu
--target=arm-none-eabi".
For bug reporting instructions, please see:
<https://support.codesourcery.com/GNUToolchain/>...
(gdb)
```

El siguiente paso es indicar a gdb que queremos establecer una conexión remota (via red), con nuestro propio equipo en el puerto 1234 (puerto por defecto de qemu):

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x01002e70 in ?? ()
```

No es el objetivo de este documento explicar el funcionamiento de GDB ni sus opciones. En la wiki se encuentra un documento más extenso sobre las opciones de GDB. Baste decir que, una vez aquí, podemos hacer uso del comando list para ver las líneas del programa y la orden `br fichero:numlinea` para colocar un *breakpoint*, entre otros muchos.

```
(gdb) br main
(gdb) c
```

A.2.3. Compilando

El proyecto incluye una serie de Makefiles que permiten la sencilla compilación del mismo. Desde la carpeta `src/` se puede ejecutar la orden *make all* para obtener la compilación del *kernel* y los procesos de prueba. Para los detalles sobre la compilación y el enlazado de los diferentes componentes recomendamos la lectura del contenido de los Makefiles.

Apéndice B

Glosario

AOT : *ahead-of-time*, técnica de compilación que transforma código compilado a lenguaje intermedio en binarios dependientes del sistema. La diferencia con JIT es que AOT se realiza antes de la ejecución, en lugar de durante la ejecución, evitando la pérdida de rendimiento.

API : *application programming interface* es una interfaz que define la forma en la que una aplicación solicita los servicios de una librería o del sistema operativo. Un API determina el vocabulario y las convenciones de llamada que el programador debe emplear para usar los servicios.

barrel shifter : es un circuito digital que permite desplazar una palabra de datos un determinado número de bits en un ciclo de reloj.

BBC Micro : serie de microcomputadores y periféricos asociados y construidos por Acorn Computers para el *BBC Computer Literacy Proyect* operado por la BBC. Fue diseñado para la educación y sus principales características son su dureza, posibilidades de expansión y la calidad de su sistema operativo. Fue la plataforma empleada para el desarrollo de la arquitectura ARM

Bootloader : o cargador de arranque, es un programa que se encarga de preparar el entorno necesario para que un sistema operativo pueda comenzar a funcionar.

bytecode : es un término empleado para denotar diversas formas de repertorios de instrucciones diseñados para ser ejecutados de forma eficiente por un intérprete *software* y para poder ser compilados posteriormente a código máquina. El nombre proviene de repertorios de instrucciones que tienen *opcodes* de un byte, seguidos de parámetros opcionales. Las representaciones intermedias como el *bytecode* pueden ser producidas por

implementaciones de un lenguaje de programación para facilitar su interpretación, o pueden ser empleadas para reducir las dependencias del *hardware* y del sistema operativo, permitiendo la ejecución del mismo código en diferentes plataformas. El *bytecode* puede ser ejecutado directamente sobre una máquina virtual (por ejemplo un intérprete), o puede ser compilado a código máquina para obtener mejor rendimiento.

CAD : *Computer Asisted Desing* es el uso de un amplio rango de herramientas computacionales que asisten a ingenieros, arquitectos y a otros profesionales del diseño en sus respectivas actividades.

CVS : *Concurrent Versioning System*, aplicación informática que implementa un sistema de control de versiones: mantiene el registro de todo el trabajo y los cambios en los ficheros que forman un proyecto y permite que los desarrolladores colaboren. Tienen estructura cliente-servidor.

DCVS : *Distributed CVS* es un CVS en el que la arquitectura cliente-servidor se sustituye por una arquitectura distribuida, lo que flexibiliza el sistema, permitiendo crear versiones en local sin depender de un servidor central.

DHCP : *Dynamic Host Configuration Protocol* protocolo empleado por los dispositivos para obtener información de configuración para operar en una red IP.

DMA : *Direct memory access* capacidad presente en computadores y microprocesadores modernos que permite a ciertos subsistemas *hardware* acceder al sistema de memoria de forma independiente a la CPU. Los computadores que disponen de canales DMA pueden realizar transferencias de datos hacia o desde los dispositivos con mucha menos sobrecarga que los computadores que carecen de ellos. Al emplear E/S programada en dispositivos sin DMA, la CPU está ocupada mientras se efectúa la transferencia, incapacitándola para realizar otra operación. Con DMA, la CPU puede iniciar la transferencia, efectuar otras operaciones mientras se realiza la transmisión y recibir una interrupción del controlador DMA una vez que ésta termina.

DRM : *Digital rights management*, término genérico que se refiere a tecnologías de control de acceso empleada para imponer limitaciones al uso de contenidos y dispositivos digitales.

DSP : *Digital signal processing*, representación de señales como una secuencia de números o símbolos y el procesamiento de esas señales.

Dhystone : es un benchmark sintético que pretende ser representativo del rendimiento con enteros.

GPL : *GNU General Public License* es una licencia de *software* libre ampliamente usada, originalmente escrita por Richard Stallman para el *GNU project*. Es la licencia *copyleft* más conocida y popular. La licencia GPL garantiza a los receptores de un programa informático los derechos de la definición de *software* libre y usa el *copyleft* para asegurar que se preservan las libertades incluso cuando el programa es modificado.

GUI : *Graphical user interface*. Una interfaz gráfica de usuario es un tipo de interfaz de usuario que permite la interacción con dispositivos electrónicos. Una GUI ofrece iconos e indicadores visuales, frente a las interfaces basadas en texto, para representar totalmente la información y las acciones disponibles.

IP core : en el diseño electrónico la propiedad intelectual del semiconductor, *IP block* (bloque IP), *IP core* (núcleo IP), o *logic core* (núcleo lógico) es una unidad reusable de lógica o el diseño del *layout* de un chip, y es también la propiedad intelectual de una empresa.

ISA : *Instruction set arquitecture* es la parte de la arquitectura de un computador relacionada con la programación, incluyendo los tipos de datos nativos, instrucciones, registros, modos de direccionamiento, arquitectura de memoria, gestión de interrupciones y excepciones, y operaciones de E/S. Una ISA incluye la especificación del registro de *opcodes*, los comandos nativos implementados por un procesador particular.

Intel Atom : es el nombre comercial de una linea de CPUs x86 y x86-64 de Intel, empleada fundamentalmente en *netbooks* debido a su rendimiento y bajo consumo respecto a otras implementaciones de la arquitectura x86.

JIT : la compilación *Just-in-time* es una técnica que permite mejorar el rendimiento en ejecución de programas informáticos. JIT se apoya en ideas previas de entornos en tiempo de ejecución: compilación a *bytecode* y compilación dinámica. JIT convierte el código intermedio a código nativo en tiempo de ejecución.

microcódigo : o microprograma, es un tipo particular de firmware empleado en algunos procesadores de propósito general. El microcódigo es una secuencia de datos binarios, o microinstrucciones, que representan señales eléctricas internas de la unidad de control de un microprocesador. Una instrucción de la arquitectura se realiza ejecutando varias de estas microinstrucciones.

MIPS : *Million instructions per second* o millones de instrucciones por segundo es una escala empleada en la medida de la velocidad de un procesador.

MMU : la *Memory management unit* es un dispositivo *hardware* responsable de gestionar los accesos a memoria solicitados por la CPU. Sus funciones incluyen la traducción de direcciones virtuales a direcciones físicas (gestión de la memoria virtual), protección de memoria, control de cache y arbitraje del bus.

MMX : es un repertorio de instrucciones SIMD diseñado por Intel, introducido en 1997 en su linea de procesadores Pentium.

MOS 6502 : el MOS Technology 6502 es un microprocesador de 8 bits diseñado por Chuck Peddle y Bill Mensch para MOS Technology. En el momento de su lanzamiento se convirtió en el microprocesador más asequible del mercado, costando menos de una sexta parte de lo que costaban diseños de compañías como Motorola e Intel.

netbooks : computador portátil de bajo costo, reducidas dimensiones y rendimiento limitado. Su diseño se centró en potenciar la autonomía y la portabilidad. Estos dispositivos se usan principalmente para navegar por Internet y realizar funciones básicas como procesamiento de texto.

ODM : un *Original design manufacturer* es un OEM que diseña y fabrica productos o componentes para venderlos a segundas compañías que venden estos productos bajo su nombre. La diferencia crucial es que los ODM diseñan y fabrican, mientras que los OEM solo fabrican.

pipelining : o segmentación, es un método por el cual se consigue aumentar el rendimiento de algunos sistemas electrónicos digitales. Se aplica fundamentalmente en microprocesadores. Consiste en descomponer la ejecución de cada instrucción en varias etapas para poder empezar a procesar una instrucción diferente en cada una de ellas y trabajar con varias a la vez. Cada una de las etapas de la instrucción usa en exclusiva un *hardware* determinado del procesador, de tal forma que la ejecución de cada una de las etapas, en principio, no interfiere con la ejecución del resto. En el caso de que el procesador no pudiese ejecutar las instrucciones en etapas segmentadas, la ejecución de la siguiente instrucción sólo se podría llevar a cabo tras la finalización de la primera. En cambio en un procesador segmentado, salvo excepciones de dependencias de datos o uso de unidades funcionales, la siguiente instrucción podría iniciar su ejecución tras acabar la primera etapa de la instrucción actual.

RISC : *reduced instruction set computer*, representa una estrategia de diseño de CPUs basada en la idea de que instrucciones simplificadas pueden proporcionar un mejor rendimiento debido a que se pueden ejecutar más rápido. Las arquitecturas RISC también suelen ser denominadas arquitecturas *load/store*.

SIMD : la técnica *Single Instruction Multilpe Data* se emplea para lograr paralelismo a nivel de datos.

smartphone : (teléfono inteligente en español) es un dispositivo electrónico que funciona como un teléfono móvil con características similares a las de un ordenador personal. Casi todos los teléfonos inteligentes son móviles que soportan completamente un cliente de correo electrónico con la funcionalidad completa de un organizador personal. Una característica importante de casi todos los teléfonos inteligentes es que permiten la instalación de programas para incrementar el procesamiento de datos y la conectividad.

SoC : *System on Chip*, se refiere a la integración de todos los componentes de un computador u otro sistema electrónico en un solo circuito integrado (chip). El área de aplicación típica es la de los sistemas empotrados.

TFTP : Son las siglas de *Trivial file transfer Protocol*, o Protocolo de transferencia de archivos trivial. Es un protocolo de transferencia muy simple semejante a una versión básica de FTP.

TLB : siglas de *Translation Lookaside Buffer*, es una memoria caché administrada por la MMU, que contiene partes de la tabla de paginación, es decir, relaciones entre direcciones virtuales y físicas. Posee un número fijo de entradas y se utiliza para obtener la traducción rápida de direcciones. Si no existe una entrada buscada, se deberá revisar la tabla de paginación y tardará varios ciclos más, sobre todo si la página que contiene la dirección buscada no está en memoria primaria. Si en la tabla de paginación no se encuentra la dirección buscada, se producirá un fallo de página.

Bibliografía

[1] Ingmar Alting. A port of the minix os to the powerpc platform. Master's thesis, Vrije Universiteit, Amsterdam, September 2006.

[2] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, 2006.

[3] Barry B. Brey. *Los microprocesadores Intel. Arquitectura, programación e interfaz de los procesadores 8086/8088, 80186/80188, 80286, 80386, 80496, Pentium, Pentium Pro y Pentium II (Quinta Edición)*. Prentice Hall.

[4] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel (3rd Edition)*. O'Reilly, 2006.

[5] Robert Love. *Linux Kernel Development (2nd Edition)*. Novell Press, 2005.

[6] David Seal. *ARM Architecture Reference Manual (2nd Edition)*. Addison-Wesley.

[7] Andrew N. Sloss, Dominic Symes, and Chris Wright. *ARM System Developer's Guide. Designing and optimizing system software*. Morgan Kaufmann.

[8] Steve Furber. *ARM System-on-chip architecture (2nd Edition)*. Addison-Wesley.

[9] Karim Yaghmour. *Building Embedded Linux Systems*. O'Reilly.

[10] Kernel Log. More than 10 million lines of linux source files, October 2008.

[11] Fabrice Bellard. QEMU: Open Source Processor Emulator. `http://www.qemu.org/`, Julio 2008.

[12] Minix 3. The minix 3 operating system. `http://www.minix3.org/`, 2004. [Online; accessed 1-September-2009].

[13] ARM Info Center. Arm mmu software accesible registers. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0333g/I1031142.html.

[14] WINE. http://www.winehq.org. [Online; accessed 12-September-2009].

[15] U-Boot. http://www.denx.de/wiki/U-Boot/. [Online; accessed 12-September-2009].

[16] BeagleBoard. http://beagleboard.org/. [Online; accessed 12-September-2009].

[17] GNUARM. http://www.gnuarm.com/. [Online; accessed 12-September-2009].

[18] Code Sourcery. http://www.codesourcery.com/sgpp/lite/arm. [Online; accessed 12-September-2009].

[19] ARM LTD. http://www.arm.com/. [Online; accessed 12-September-2009].

[20] Launchpad. http://www.launchpad.net/. [Online; accessed 12-September-2009].

[21] Canonical. http://www.canonical.com/. [Online; accessed 12-September-2009].

[22] Ubuntu. http://www.ubuntu.com/. [Online; accessed 12-September-2009].

[23] Bazaar. http://bazaar-vcs.org/. [Online; accessed 12-September-2009].

[24] Dokuwiki. http://www.dokuwiki.org/. [Online; accessed 12-September-2009].

[25] Flyswater. http://www.tincantools.com/product.php?productid=16134. [Online; accesed 12-September-2009].

[26] OpenOCD. http://openocd.berlios.de/web/. [Online; accesed 12-September-2009].

[27] Descargar CodeSourcery. http://www.codesourcery.com/sgpp/lite/arm/portal/subscription?@template=lite. [Online; accessed 12-September-2009].

[28] *Google Summer of Code*. http://code.google.com/soc/. [Online; accessed 12-September-2009].

[29] Minix 3: *Who is doing what?* http://www.minix3.org/who_doing_what.html. [Online; accessed 12-September-2009].

[30] Proyecto Minix to ARM en el GSoC 2008. http://minix-to-arm.blogspot.com/. [Online; accessed 12-September-2009].

[31] Wiki Minix@ARM. http://www.nosomoslibres.com/wiki. [Online; accessed 12-September-2009].

[32] Proyectos del GSoC sobre Minix 3 abandonados. http://groups.google.com/group/minix3/browse_thread/thread/a74a64f68ba37a05?pli=1. [Online; accessed 12-September-2009].

Índice de figuras