

UNIVERSIDAD DE GRANADA

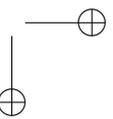
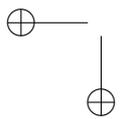
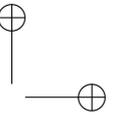
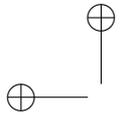


PROYECTO DE FIN DE CARRERA

Technical report nº 2:
Acerca del kernel Linux

Autor:
Ricardo CAÑUELO NAVARRO

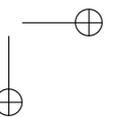
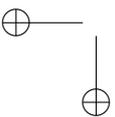
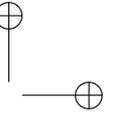
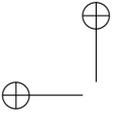
Tutor:
Jesús GONZÁLEZ PEÑALVER



Licencia

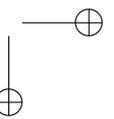
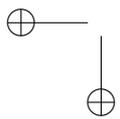
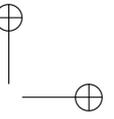
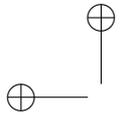
Copyright © 2009 Ricardo Cañuelo Navarro <ricardo.canuelo@gmail.com>.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the file named FDL.



Índice general

1. Introducción	1
2. La construcción del kernel	2
2.1. ¿Por qué necesitamos construir el kernel?	3
2.2. Las herramientas	3
2.3. La construcción	4
3. El kernel y la imagen del kernel	10
3.1. El kernel y sus componentes	10
3.2. La imagen del kernel y sus componentes	14
4. Inicialización del kernel	19
5. Inicialización del sistema	24
5.1. El sistema de archivos raíz	24
5.2. El proceso init	26
5.3. Disco RAM inicial	28
5.3.1. initrd	28
5.3.2. initramfs	29
A. Scripts y código fuente	31
B. Archivos del código del kernel mencionados	34
Bibliografía	36



1

Introducción

Debido a la naturaleza del proyecto, se hace necesario contar con unos conocimientos bien afianzados y en cierta profundidad acerca de algunos detalles del kernel de Linux. En este informe trataremos varios puntos acerca de la arquitectura, la organización, la configuración y el arranque del kernel, entrando en suficiente detalle como para conocer a fondo al menos los aspectos más relevantes.

Previamente a esto se explicará paso a paso el procedimiento de construcción del kernel a nivel general a modo de introducción para acercar más el mundo del desarrollo en Linux a quien no tenga experiencia en este campo. De este modo, aunque en el resto del informe se tratan temas avanzados se intentará que la curva de aprendizaje sea suave para alguien con una experiencia básica en Linux.

Cuando entremos en detalles de implementación a bajo nivel nos centraremos en la arquitectura ARM, que es la que nos interesa para el proyecto, tratando en el resto de casos de ofrecer una visión lo más global posible.

2

La construcción del kernel

Linux forma parte de la amplia familia de sistemas operativos de tipo Unix. Fue desarrollado por Linus Torvalds en 1991 como un sistema operativo para ordenadores personales IBM y compatibles basados en el 80386 de Intel. Desde entonces ha sufrido un crecimiento más que enorme, especialmente desde finales de los 90, y actualmente está portado a un gran número de arquitecturas y funcionando en la mayoría de servidores de Internet del mundo y, cada vez más, en máquinas de escritorio.

Técnicamente hablando, Linux es un kernel Unix, aunque no es realmente un sistema operativo completo porque no contiene aplicaciones. Sin embargo, como la mayoría de ellas están disponibles bajo licencia GPL, pueden instalarse en cualquier sistema basado en Linux. Lo más común es encontrar *distribuciones* del kernel Linux junto con una colección de paquetes de software que forman lo que se viene llamando GNU/Linux.

La versión actual de Linux es la 2.6, y es la que usaremos a lo largo del proyecto. La versión estable anterior fue la 2.4, y difería notablemente con respecto a la 2.6. Entre otras cosas cambió la numeración de las versiones. Actualmente, la numeración consiste en una cifra después del 2.6 que indica el número de revisión, y otra cifra después del número de revisión que indica la versión del parche de esa revisión. Esto se hace así porque una revisión puede cambiar bastante con respecto a la inmediatamente anterior o posterior, y cuando aparece una revisión nueva es probable que contenga fallos que se van solventando con los lanzamientos de las versiones parcheadas. Así podemos encontrar un kernel con un número de versión 2.6.11.12 (versión 2.6, revisión 11, parche 12).

El kernel de Linux se distribuye bajo la licencia GPL, lo que significa entre otras cosas que su código fuente está disponible gratuitamente y que podemos obtenerlo,

compilarlo, observarlo y modificarlo con total libertad. Nosotros haremos uso de estas libertades a lo largo del proyecto, y especialmente en este informe.

2.1. ¿Por qué necesitamos construir el kernel?

Cuando instalamos Linux en un ordenador de sobremesa, la mayoría de las veces pasamos por alto el aspecto de la instalación del kernel (la parte más importante del sistema). Esto es así porque la inmensa mayoría de las distribuciones de Linux para máquinas de escritorio ya vienen preparadas para instalar un kernel por defecto que satisfaga las necesidades de casi todo el mundo. Por eso no es de extrañar que casi la totalidad de los usuarios ocasionales, y también muchos usuarios habituales de Ubuntu, Mandriva, Suse o Fedora desconozcan o no se sientan interesados por estas posibilidades que tienen a mano. Sencillamente estas distribuciones vienen preparadas para funcionar sin problemas nada más instalarlas.

Sin embargo hay muchas otras ocasiones en las que, bien porque trabajamos en una máquina con unos requisitos muy específicos, o bien porque queremos construir un sistema que se ajuste mejor a nuestras necesidades, necesitamos construir nuestra propia versión del kernel. En este proyecto, las razones para hacerlo son que necesitamos que el kernel se adapte para funcionar específicamente en nuestro hardware, y que dicho hardware tiene unas restricciones de memoria y de arquitectura que hacen necesario quitar del kernel todo aquello que no nos vaya a hacer falta.

2.2. Las herramientas

Como acabamos de ver, el código fuente del kernel (que no es más que un programa muy complejo y muy grande) está a nuestra entera disposición. Por supuesto, lo necesitamos para construir el kernel. El sitio oficial y el primero en el que hay que mirar para obtenerlo es www.kernel.org, sin embargo también lo podemos encontrar en los repositorios de la distribución que utilicemos. Por ejemplo, en Debian podemos instalar con *aptitude* o *Synaptic* el paquete “kernel-source-2.6.xx”¹, que nos dejará un archivo comprimido “linux-source-2.6.xx.tar.bz2” con todo el código fuente en el directorio `/usr/src`.

 En posteriores informes hablaremos en detalle de algunas de estas herramientas

¹Las xx son un número cualquiera de versión dentro de la versión 2.6.

Además del propio código fuente necesitaremos algunas herramientas básicas para compilarlo. Entre ellas están el compilador de C de GNU (gcc), Gnu Make, las binutils, udev y utilidades para la gestión de los sistemas de archivos como e2fsprogs, jfsutils o reiserfsprogs (dependiendo del sistema de archivos que escojamos). Dentro del árbol del código fuente del kernel, en el archivo Documents/Changes se encuentra una lista de todo el software necesario y de las versiones mínimas requeridas. Para conseguir estas herramientas basta con utilizar la utilidad de instalación de software de nuestra distribución e instalarlos. Son herramientas básicas y fundamentales, así que no debería haber ningún problema a la hora de encontrarlas ni de instalarlas. Además, las herramientas actuales de gestión de paquetes como apt-get realizan todo el trabajo del mantenimiento de dependencias y versiones por nosotros.

2.3. La construcción

Ahora que tenemos todo lo necesario para construir el kernel vamos a ponernos manos a la obra.

Empezaremos descomprimiendo el código fuente del kernel (si no lo tenemos ya descomprimido) en un directorio en el que tengamos permisos de escritura. Esto lo podemos hacer con:

```
$ tar zxvf linux-source-2.6.xx
```

Si el archivo está comprimido con gzip (extensión tar.gz), y con:

```
$ tar jxvf linux-source-2.6.xx
```

Si está comprimido con bzip2 (extensión tar.bz2). En ambos casos el archivo deberá estar dentro del directorio donde queremos extraerlo.

 Los módulos del kernel son componentes que aportan funcionalidades adicionales sin tener que estar integrados en la imagen del kernel. Se pueden cargar dinámicamente cuando sean necesarios, lo que da mayor flexibilidad al sistema.

Dentro del árbol de directorios que se ha extraído se encuentra el código fuente para construir el kernel para multitud de arquitecturas distintas junto con todos los módulos del kernel disponibles. El archivo README contiene una documentación básica muy valiosa a la hora de construir el kernel, incluyendo todo el procedimiento que se detalla aquí.

Si hemos hecho alguna compilación anterior de este código fuente, antes de seguir hay que asegurarse de que no quedan archivos objeto perdidos que puedan estorbar a la compilación nueva que vamos a hacer. Para limpiar el código fuente hacemos:

```
$ make mrproper
```

Este paso no hay que hacerlo si acabamos de descomprimir el código.

Ahora empieza el verdadero trabajo, la configuración. El kernel de Linux está preparado para funcionar en una infinidad de máquinas distintas con una enorme cantidad de dispositivos de todo tipo y con características de lo más dispares. Desde sistemas empotrados hasta clusters de ordenadores y desde “dumb terminals” hasta potentes equipos de escritorio. Es por eso que la configuración del kernel es una tarea larga y, en cierto modo, delicada (dependiendo de cuánto queramos refinar nuestro sistema, claro está). Para que este paso no se nos haga tan cuesta arriba conviene que tengamos en mente antes de empezar las ideas generales acerca de qué necesitamos para nuestro sistema. Ahora mismo vamos a suponer que queremos construir un kernel para un ordenador de sobremesa y que lo que queremos hacer mantener un sistema funcional como lo tenemos ahora, y además activar una funcionalidad del kernel que no viene activada por defecto. Bien, dicho esto comenzamos la configuración haciendo:

```
$ make menuconfig
```

Con esto iniciamos la configuración basada en menú en una terminal en modo texto (los menús están hechos con ncurses). Otras opciones son usar `make xconfig` o `make gconfig` para lanzar las herramientas de configuración en modo gráfico (con las bibliotecas Qt y Gtk respectivamente). La interfaz en modo texto puro (`make config`) no se recomienda porque, además de ser menos amigable, tendríamos que empezar desde el principio cada vez que nos equivocáramos en algo.

La herramienta de configuración nos mostrará una serie de secciones de configuración, cada uno de los cuales nos lleva hacia otras opciones y otras subsecciones. Cuando seleccionemos una opción podemos hacer tres cosas: marcarla (con “Y”), desmarcarla (con “N”) o marcarla como módulo (con “M”). Si la opción aparece precedida de “[]” las opciones son marcar o desmarcar (se puede marcar o desmarcar con Espacio). Si por el contrario aparece precedida de “<” se puede marcar también como módulo.

La estrategia a seguir es marcar como módulo aquellas funcionalidades que no vayan a ser imprescindibles pero que sepamos que nos van a hacer falta en algún momento, como por ejemplo el soporte para sistemas de archivos que no sean los nativos de nuestro sistema, pero que nos podamos encontrar en un pen drive, en un cd o en otro disco duro. Las funcionalidades que sí sean imprescindibles conviene que las marquemos para que

 No es necesario que hagamos toda la configuración de una sentada. Podemos guardar la configuración que llevemos hecha para continuarla en otro momento

queden integradas en el kernel.

Si construimos el kernel para un pc moderno, sabemos que tenemos una máquina con una gran cantidad de memoria y una cpu rapidísima, así que no pasa nada si incluimos más cosas de la cuenta. De hecho, los kernels que vienen contruidos por defecto en la mayoría de las distribuciones están configurados con todas las funcionalidades que cualquiera pueda necesitar. Así que en caso de duda conviene marcar aquellas opciones que no sepamos a ciencia cierta si llegaremos a usar o no. Así nos aseguraremos que no tendremos problemas a la hora de hacer funcionar el sistema, con la contrapartida de que tendremos una imagen del kernel más grande. Pero como hemos dicho, en un pc actual ni nos daremos cuenta.

Como ejemplo vamos a suponer que queremos activar una funcionalidad que por defecto nos viene desactivada: El uso de la tecla “Pet Sis”. Veremos a modo de curiosidad un uso bastante desconocido de esta tecla. Esta opción se encuentra dentro de la sección “Kernel hacking”, lo que la hace aún más interesante, y es la opción “Magic SysRq key”. Lo que se consigue activando esta opción es poder mandarle mensajes al kernel utilizando esta tecla en combinación con otras. Esto es útil en momentos en los que el sistema se quede aparentemente congelado y no responda ni siquiera a los cambios de terminal ni a las combinaciones Ctrl+Z, Ctrl+C y Ctrl+Alt+Supr (nuestro último recurso). Ahora dispondremos de otra combinación de emergencia: Alt+PetSis+R,S,E,I,U,B². Cada letra realiza una acción de lo que es básicamente un apagado de emergencia: sincronizar los dispositivos, parar los discos, etc. hasta llegar a PetSis+B, que es como pulsar el botón de reset.

Para el resto de opciones seguiremos la estrategia que mencionamos antes, y en caso de duda sobre si necesitaremos una determinada funcionalidad o no, lo mejor es añadirla. Por supuesto, hay partes que casi nunca necesitaremos y las podemos desactivar, como los drivers de dispositivos antiguos como los de cinta (si estamos seguros de que no las utilizaremos), los sistemas de archivos obsoletos, el soporte para hardware que no tenemos, etc.

Bien, una vez hechos los cambios en la configuración seleccionamos “Exit” para salir y después “Yes” para guardar la configuración que hemos hecho en el archivo .config. Así si algo va mal al arrancar el nuevo kernel podemos volver a editar esta configuración y cambiar aquello que creamos que nos da problemas, sin tener que rehacer toda la configuración desde el principio. El siguiente paso es compilar el kernel:

 Lamentablemente la configuración del kernel es en muchos casos un proceso de prueba y error

```
$ make
```

²Regla nemotécnica: Raising Skinny Elephants Is Utterly Boring

Esta orden crea una imagen comprimida del kernel y compila los módulos que seleccionamos en la fase de configuración. Si preparamos las herramientas correctamente y tenemos suficiente espacio en disco, todo debería ir bien. Eso sí, este proceso puede llegar a tardar bastante y que mientras se lleva a cabo no tiene mucho sentido quedarse mirando todo lo que aparece en la pantalla (a no ser que sepas realmente lo que estás mirando), así que da tiempo de sobra de tomarse un café mientras.

Una vez terminado, se habrán construido tanto el kernel como los módulos. Para instalarlo todo tenemos que ser root, así que este es el momento de cambiar de usuario. Todos los pasos anteriores se pueden hacer con cualquier usuario normal (de hecho es lo recomendado).

Para instalar los módulos hacemos:

```
% make modules_install
```

Y para instalar el kernel:

```
% make install
```

Esta última orden lo que hace es copiar dos archivos importantísimos (bzImage y System.map) al directorio /boot. Esto también lo podemos hacer a mano:

```
% cp arch/nuestra_arquitectura/boot/bzImage /boot/vmlinuz-2.6.xx  
% cp System.map /boot/System.map-2.6.xx
```

Más adelante veremos qué son y cómo se construyen estos archivos. De momento basta con saber que bzImage es el kernel y System.map es una lista de símbolos del kernel. Se copian al directorio boot porque serán utilizados por nuestro cargador de arranque para iniciar el sistema, y se les cambia el nombre porque es bastante frecuente tener dentro de /boot varias versiones del kernel, cada una con su tabla de símbolos. De este modo podemos nombrar cada versión de una forma distinta y evitamos conflictos. Es muy recomendable no borrar nuestra versión actual del kernel cuando añadamos una nueva, porque si la borramos y la versión nueva no consigue arrancar el sistema lo tendremos muy difícil para salir del problema en el que nos hemos metido. Si conservamos nuestra versión actual del kernel siempre podremos volver a como estábamos antes.

Otro aspecto a tener en cuenta es que si hemos construido los drivers principales para el arranque como módulos, necesitaremos crear un disco ram. De nuevo, veremos qué es esto más adelante. El por qué lo necesitamos es fácil de ver: Si los drivers no están

integrados en el kernel no es posible montar el sistema de archivos raíz. Es necesario cargar los drivers para utilizar el sistema de archivos, pero no podemos porque los drivers se encuentran en el sistema de archivos y este no está montado. En otras palabras, el problema del huevo y la gallina. Para solucionarlo el kernel necesita un disco ram inicial que contiene los drivers, y una vez cargados estos ya se puede montar el sistema de archivos y continuar con el arranque. Para crear un disco ram hacemos lo siguiente:

```
% update-initramfs -c -k 2.6.xx
```

Ya sólo queda decirle al cargador de arranque que estemos utilizando cómo arrancar la nueva imagen que hemos puesto en boot. Si usamos LILO, lo podemos hacer añadiendo unas líneas como las siguientes al archivo `/etc/lilo.conf`:

```
image=/boot/vmlinuz-2.6.xx
label=Nuevo kernel
root=/dev/sda5
read-only
```

Con esto creamos una nueva entrada en el menú y si la seleccionamos arrancará la imagen del kernel que hay en `/boot/vmlinuz-2.6.xx`. Además se especifica que el sistema de archivos raíz será `/dev/sda5`. Más adelante veremos qué es el sistema de archivos raíz. Por ahora vamos a conformarnos con poner como parámetro `root` el mismo que utiliza nuestro kernel actual. Para que los cambios hagan efecto tenemos que hacer como root:

```
% /sbin/lilo
```

En grub la configuración de `/boot/grub/menu.lst` sería algo así:

```
title=Nuevo kernel
root (hd0,1)
kernel /boot/vmlinuz-2.6.xx root=/dev/sda5 ro
initrd /boot/initrd.img-2.6.xx
```

Como en el caso anterior, lo mejor es que pongamos en los parámetros `root` lo mismo que para nuestro kernel actual.

Hecho todo esto ya podemos reiniciar la máquina, y si todo ha salido bien veremos arrancar nuestro kernel configurado a nuestra medida. Una forma de ver rápidamente si está arrancando el kernel correcto es modificar la variable `linux_banner` del archivo `init/version.c` del código del kernel para que al arrancar escriba el mensaje que nosotros queramos en lugar de “Linux version ...”. Si algo falla, tendremos que volver a arrancar con el kernel que usábamos antes y volver a realizar los pasos anteriores variando la configuración hasta dar con una que funcione.

La construcción del kernel

9

Hasta aquí todo lo referente a cómo se compila el kernel. En los siguientes capítulos se verán temas avanzados de implementación que son de interés general. A partir de este punto vamos a considerar la arquitectura del microprocesador de nuestra placa de desarrollo en lugar del pc de sobremesa.

3

El kernel y la imagen del kernel

El código fuente de Linux para todas las arquitecturas soportadas alberga más de 14000 archivos en ensamblador y C, que conjuntamente llegan a los 6 millones de líneas de código. Su complejidad hace que su estudio no sea sencillo pese a disponer del código fuente completo.

Si nuestro interés es conocerlo en profundidad no es por lo general una buena idea comenzar a trazar el código desde cero para ver qué está ocurriendo, ya que el tamaño del programa, unido al multihebrado y la apropiatividad convierten esta tarea en casi imposible para alguien con poca experiencia.

Una forma más práctica de entender la estructura de una gran imagen binaria como es el kernel es examinar los componentes con los que se ha construido. De esta forma podemos ver qué hace cada componente aislado y qué papel juega cada uno en la imagen global. Teniendo a mano el código fuente del kernel y las binutils podemos inspeccionar en detalle las partes del kernel que nos interesen.

3.1. El kernel y sus componentes

Cuando compilamos y construimos el kernel se generan varios archivos comunes, además de uno o más *módulos binarios* específicos de la arquitectura para la que se ha construido el núcleo. Los archivos comunes se construyen siempre, independientemente de la arquitectura objetivo. Dos de ellos son `System.map` y `vmunix`. El primero de ellos lo vimos por encima en el capítulo anterior y consiste en una lista de los símbolos del kernel así como de sus direcciones (similar a la salida de la orden `nm vmunix`) en un archivo plano que se puede visualizar como cualquier archivo de texto corriente. Este archivo nos puede servir a la hora de depurar para seguir la pista dentro del kernel de

los símbolos que nos interesen. El segundo es el kernel propiamente dicho en formato ELF¹ propio de la arquitectura de tipo ejecutable, sin embargo, es importante notar aquí que normalmente este archivo *nunca* se arranca directamente. Una de las razones es porque casi siempre se encuentra comprimido, por lo que se hace imprescindible un cargador de arranque que ponga en marcha el proceso de descompresión de la imagen. Más adelante veremos este proceso en más profundidad.

 Cuando hablamos de la “imagen del kernel” nos estamos refiriendo al archivo binario que contiene el código objeto del kernel

Código 3.1 Parte de la salida de la construcción del kernel

```
LD      vmlinux
SYSMAP  System.map
SYSMAP  .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy.gz
AS      arch/arm/boot/compressed/piggy.o
CC      arch/arm/boot/compressed/misc.o
AS      arch/arm/boot/compressed/head-xxx.o
AS      arch/arm/boot/compressed/big-endian.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

Este archivo (`vmlinux`) es el kernel propiamente dicho, una imagen monolítica y completamente independiente, sin ninguna referencia externa. Cuando se ejecuta en el contexto apropiado (por un cargador de arranque) arranca la placa en la que se está ejecutando y, como resultado, la placa se queda con un núcleo completamente funcional y lista para realizar el resto de la inicialización del sistema.

¹El formato ELF (Executable Linker Format) es un formato estándar de archivos objeto. Más información sobre ELF en [3]

Código 3.2 Extracto de System.map

```
010bce70 t processor_modes
010bcef0 t isa_modes
010bcf00 T cpuinfo_op
010bcf10 t hwcaps_str
010bcf40 t proc_arch
010bcf84 T sigreturn_codes
010bcfa0 t handler
010bcfb0 t cpu_arch_name
010bcfb7 t cpu_elf_name
010bcfba t cpu_arm7tdmi_name
010bcfc3 t cpu_triscenda7_name
010bcfd0 t cpu_at91_name
010bcfe1 t cpu_s3c3410_name
010bcff1 t cpu_s3c44b0x_name
```

 Cuando aparezca una elipsis (...) al principio de un path nos estaremos refiriendo al directorio base del código del kernel.

Como se comentó en la introducción, la imagen vmlinux está compuesta de varias imágenes binarias (figura 3.1), así que podemos estudiar las que más nos interesen. Claro está, no entraremos en detalle acerca de qué es cada una y para qué sirve porque queremos estudiar las cuestiones de implementación de una arquitectura concreta y eso entra más en el campo del diseño general del kernel². Por ejemplo, para nosotros es interesante el primer objeto, head.o. Este binario se ensambla a partir de `.../arch/arm/-kernel/head.S`, que es un archivo fuente en ensamblador de la arquitectura que estamos tratando y que realiza algunas inicializaciones del kernel a muy bajo nivel.

Este archivo contiene las primeras líneas ejecutables del kernel propiamente dicho. Más adelante veremos el proceso de arranque del núcleo con mayor detalle.

Dando un vistazo a la figura 3.1, vemos que las mayores porciones de la imagen pertenecen al código del sistema de archivos, la red y los drivers. Dentro del código del módulo kernel y los módulos propios de la arquitectura se encuentra el planificador de procesos, la gestión de procesos y hebras y otras funcionalidades principales del núcleo.

²En la bibliografía se recomiendan algunos libros que tratan este aspecto

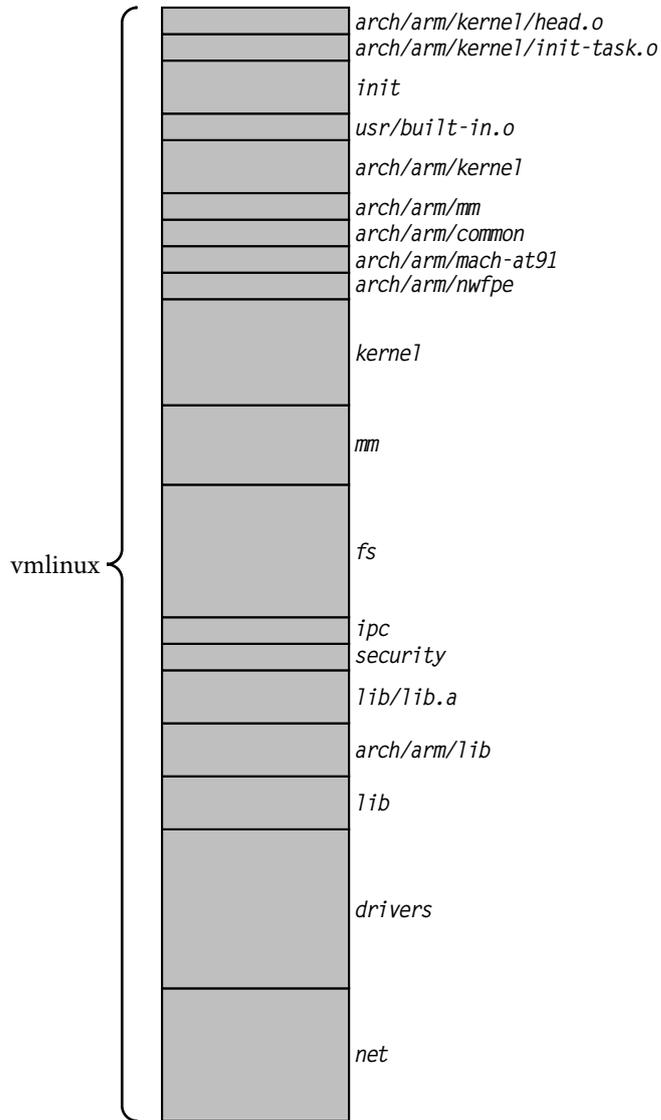


Figura 3.1: Componentes de vmlinux

3.2. La imagen del kernel y sus componentes

En la sección anterior vimos que `vmlinux` es el kernel propiamente dicho, un archivo monolítico en formato ELF binario específico de la arquitectura. También se comentó que no se puede arrancar directamente en la mayoría de las circunstancias, debido entre otras cosas a que está comprimido. A continuación veremos el por qué y cómo se construye la imagen del kernel, que es la que se arranca realmente.

Algunos cargadores de arranque y algunas arquitecturas pueden arrancar la imagen `vmlinux` directamente tras convertirla de ELF a binario `raw`³ (partiendo de la imagen sin comprimir). Sin embargo, en la mayoría de ocasiones es necesario incluir algunas funcionalidades adicionales que proporcionen un contexto apropiado y el código necesario para cargar y arrancar el kernel. Para ello sirven los últimos pasos en la construcción del kernel, en los cuales se crea una nueva imagen formada por el kernel propiamente dicho en estado comprimido junto con una serie de añadidos que permiten su arranque. En la tabla 3.1 se detallan brevemente cada uno de los componentes de la imagen.

Después de construir el archivo ELF `vmlinux`, el sistema de construcción del kernel procesa los objetivos de la tabla 3.1. El objeto `Image` se crea a partir de `vmlinux`. Básicamente, `Image` es el archivo ELF `vmlinux` sin las secciones supérfluas (notas y comentarios) y sin los símbolos de depuración que pueda haber. Para ello se utiliza la orden `objcopy`:

Código 3.3 Transformación de `vmlinux` en `Image`

```
$ objcopy -O binary -R .note -R .comment -S vmlinux arch/arm/boot/Image
```

³El término *raw binary* aparece en la bibliografía para describir una imagen binaria formada por una secuencia de instrucciones máquina listas para ser ejecutadas. Se puede traducir como “binario crudo”, aunque aquí se utilizará la palabra inglesa “raw”.

Componente	Descripción
vmlinux	El kernel propiamente dicho en formato ELF, incluyendo símbolos, comentarios y componentes específicos de la arquitectura
System.map	Tabla de símbolos del kernel vmlinux en texto plano
Image	Imagen binaria del kernel sin símbolos ni comentarios
head.o	Código de inicialización específico de la arquitectura. El cargador de arranque pasa el control a este objeto
piggy.gz	El archivo Image comprimido con gzip
piggy.o	El archivo piggy.gz en formato ensamblador para que pueda ser enlazado con misc.o (a continuación)
misc.o	Rutinas para descomprimir la imagen del kernel (piggy.gz)
head-xxx.o	Código de inicialización del procesador específico de la arquitectura
big-endian.o	Pequeña rutina en ensamblador para configurar el procesador en modo big-endian
vmlinux	Imagen compuesta del kernel. No confundir con el kernel propiamente dicho, que tiene el mismo nombre. Esta imagen es el resultado de enlazar el kernel propiamente dicho con todos los módulos de esta tabla
zImage	Imagen compuesta final que será cargada por un cargador de arranque

Tabla 3.1: Componentes de la imagen del kernel para la arquitectura ARM

Acto seguido se ensamblan una serie de pequeños módulos que realizan varias tareas de bajo nivel específicas de la arquitectura y el procesador (`head.o`, `head_XXX.o`⁴, etc). Es especialmente interesante la construcción de `piggy.o`:

En primer lugar el binario Image se comprime con `gzip` en el archivo `piggy.gz`, y a continuación se ensambla un pequeño archivo llamado `piggy.S` que referencia al archivo `piggy.gz`.

⁴Las xxx indican una variante concreta de la arquitectura ARM, como *xscale*. Este archivo funciona como añadido a `head.o` y existe sólo para algunas arquitecturas

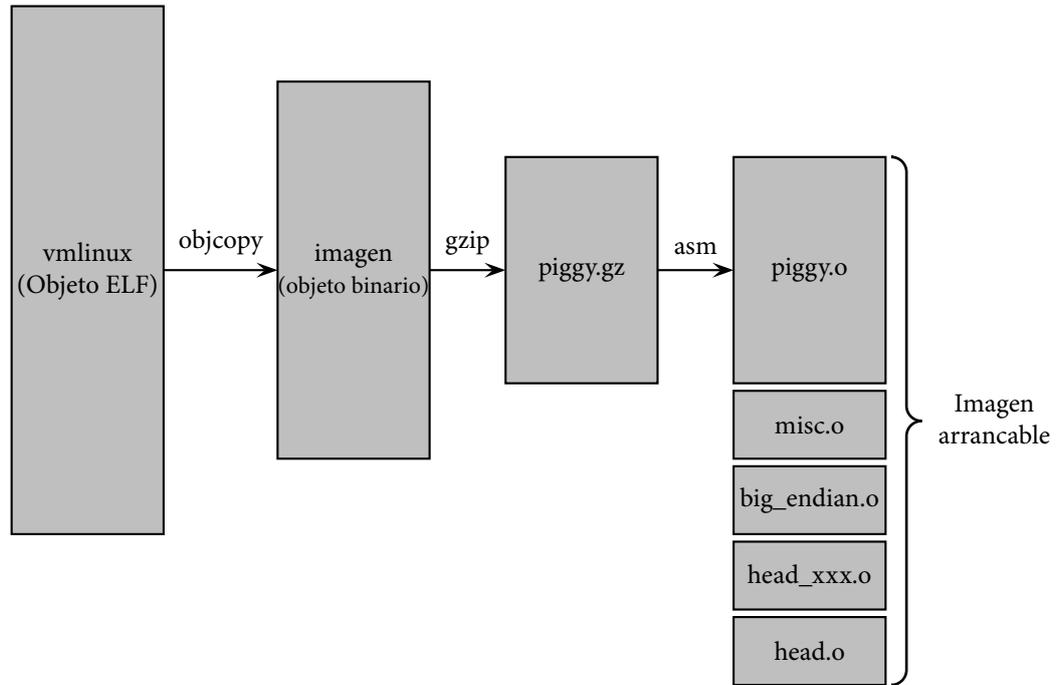


Figura 3.2: Construcción de la imagen compuesta

Código 3.4 piggy.S

```

1  .section .piggydata, #alloc
   .globl input_data
   input_data:
4  .incbin "arch/arm/boot/compressed/piggy.gz"
   .globl input_data_end
   input_data_end:
  
```

Lo que hace el código de piggy.S es decirle al ensamblador que emita la imagen comprimida del kernel (piggy.gz) a una sección del archivo objeto llamada .piggydata, con dos etiquetas input_data y input_data_end que indican el principio y el final del

kernel. Con este sencillo paso lo que se consigue es empaquetar el kernel junto a un pequeño cargador de inicialización⁵ de bajo nivel que es el encargado de inicializar el procesador y las regiones de memoria pertinentes, descomprimir la imagen del kernel, cargarla en la dirección de memoria adecuada y pasarle el control. Para conocer mejor cómo funciona el enlazado de archivos y el formato ELF recomiendo la lectura de [3].

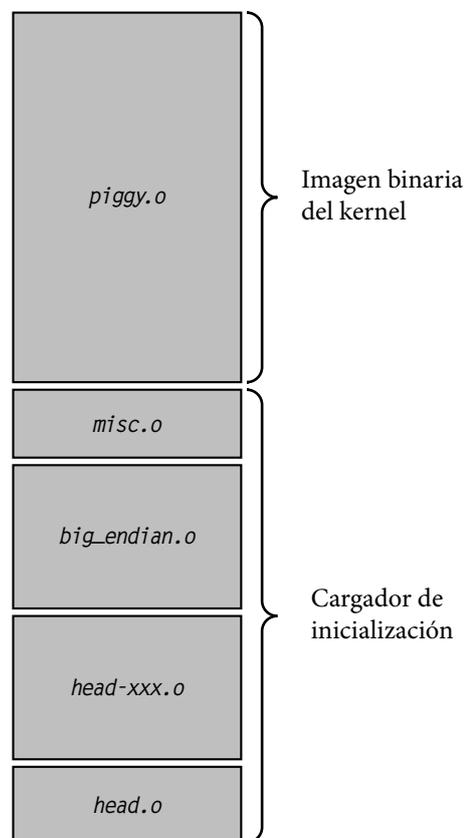


Figura 3.3: Imagen compuesta del kernel

Es importante no confundir el *bootstrap loader* con el cargador de arranque. El

⁵A este tipo de cargadores se los denomina *bootstrap loaders* en la bibliografía

cargador de arranque es el que se encarga de controlar la placa cuando esta se pone en funcionamiento y no depende en absoluto del kernel y es el que nos suena a todos (Lilo, Grub). El cargador de inicialización, por otra parte, actúa como pegamento entre el cargador de arranque y el kernel, y es el responsable de proporcionar un contexto apropiado para que el kernel se ejecute, así como de realizar los pasos necesarios para la descompresión y colocación en memoria de la imagen del kernel. Paso a paso, las funciones que realiza son las siguientes:

- Inicialización del procesador a bajo nivel, incluyendo la activación de las caches de instrucciones y de datos, desactivación de interrupciones y la configuración de un entorno de ejecución para código C. Los módulos que se encargan de esta tarea son `head.o` y `head-xxx.o`.
- Código para la descompresión y colocación del kernel, realizado por `misc.o`.
- Otras inicializaciones específicas del procesador, como `big-endian.o`, que activa el modo big-endian para este procesador en concreto⁶

⁶La arquitectura ARM es little-endian por defecto, aunque puede funcionar también en modo big-endian.

4

Inicialización del kernel

En esta sección examinaremos el flujo de control del arranque completo, que comienza en el cargador de arranque y acaba en la ejecución del kernel.

Como ya sabemos, el cargador de arranque se aloja en la memoria no volátil (una memoria flash, por ejemplo) de la placa y toma el control en cuanto se arranca la placa. Este cargador consiste en un pequeño conjunto de rutinas diseñadas para inicializar la placa y cargar una imagen de arranque, así como otras rutinas para realizar comprobaciones de memoria y dispositivos. Por último, el cargador de arranque debe poder cargar y pasar el control a otro programa, que normalmente es un sistema operativo. A continuación se explicarán los pasos que se llevan a cabo en esta última tarea.

Cuando el cargador de arranque pasa el control a la imagen del sistema operativo, lo que está haciendo es pasar el control al módulo `head.o` del *bootstrap loader*. Este se encuentra a partir de la etiqueta `Start` del mismo.

Como se explicó antes, este cargador de inicialización tiene una sola responsabilidad: crear el entorno apropiado para descomprimir y reubicar el kernel y pasarle el control. El control se pasa desde el cargador de inicialización directamente al kernel propiamente dicho, a un módulo llamado `head.o`¹, y de aquí a la función `start_kernel()` en `main.c`²

El módulo `head.o` del kernel realiza varias inicializaciones específicas de la arquitectura y el procesador para preparar el cuerpo principal del kernel, aunque las tareas específicas del procesador se intentan mantener tan genéricas como sea posible. Este módulo realiza, entre otras, las siguientes tareas:

 Ojo, no hay que confundir el archivo `head.o` del *bootstrap loader* con el del kernel. Aunque se llaman igual no son iguales y realizan tareas distintas

¹Ensamblado a partir de `.../arch/arm/kernel/head-common.S`

²El archivo se encuentra en `.../init/main.c`

- Comprueba el procesador y la arquitectura
- Crea las entradas iniciales de la tabla de páginas
- Activa o desactiva la MMU del procesador
- Salta al principio del kernel (el programa), `main.c`

A este nivel aparecen ciertas complejidades que interesa conocer. Cuando se pasa el control a `head.o`, el procesador está funcionando en *modo real*. Esto significa, entre otras cosas, que las direcciones que manejan los registros del procesador son las verdaderas direcciones físicas a las que accede este. En el caso de contar con un procesador con una unidad MMU, en el momento en el que se inicializan los registros y las estructuras de datos para permitir las traducciones de memoria, se activa dicha unidad, lo cual causa que de repente el espacio de memoria que veía el procesador sea remplazado por un esquema de direccionamiento virtual. Esto hace que en esta etapa sea imposible depurar el código paso a paso.

Por otra parte hay que tener en cuenta las limitaciones en el mapeado de memoria en esta etapa. Por ejemplo, supongamos que tenemos un dispositivo que necesita cargar el firmware en una de las primeras etapas del arranque. Para ello, una posible solución podría ser compilar estáticamente dicho firmware dentro de la imagen del kernel y después referenciarlo mediante un puntero para descargarlo al dispositivo. Sin embargo, debido al limitado mapeado de memoria, es bastante posible que la imagen se salga del rango que ha sido mapeado en esta etapa del arranque. Cuando el código se ejecute provocará un fallo de página porque se ha intentado acceder a una dirección de memoria para la cual no se ha creado un mapa válido dentro del procesador. Pero además se dará el caso de que no se ha instalado todavía un manejador de fallos de página, así que el sistema se colgará sin más. A estas alturas del arranque es probable que no se obtengan ni siquiera mensajes de error.

En lugar de esto, la mejor solución es retrasar cualquier inicialización específica del hardware hasta que el kernel haya arrancado. Así se podrá usar el sistema de drivers de Linux para acceder al hardware en lugar de tener que modificar un código de inicialización en ensamblador.

La última tarea que realiza `head.o` es pasar el control a la función principal de inicialización del kernel, que es código C (en el archivo `main.c`). Para la arquitectura ARM esto se realiza en una línea de `head-common.S`

Código 4.1 Salto a `start_kernel`

```
55 ldmia r3, {r4, r5, r6, r7, sp}
   str r9, [r4] @ Save processor ID
   str r1, [r5] @ Save machine type
   str r2, [r6] @ Save atags pointer
   bic r4, r0, #CR_A @ Clear 'A' bit
60 stmia r7, {r0, r4} @ Save control register values
   b start_kernel
```

A partir de aquí la ejecución pasa al archivo `main.c`, que hace todo el trabajo de inicialización del kernel, desde inicializar la primera hebra kernel hasta montar el sistema de archivos raíz y arrancar la primera hebra en modo usuario. La mayor parte de la inicialización del kernel se realiza dentro de la función `start_kernel`. A continuación veremos los aspectos básicos de estas tareas.

Una de las primeras cosas que ocurren dentro de `start_kernel` es una llamada a la función `setup_arch`. Esta función es la responsable de las tareas de inicialización específicas de la arquitectura. `setup_arch` llama a funciones que identifican la CPU y proporciona un mecanismo para llamar a rutinas de inicialización de la CPU de alto nivel. Una de ellas es `setup_processor`, que se encuentra en `.../arch/arm/kernel/setup.c`. Esta función verifica la identificación de la CPU y la revisión, llama a otras funciones de inicialización de la CPU que ha identificado y muestra varias líneas de información en la consola durante el arranque.

Una de las últimas acciones que llevan a cabo estas rutinas de configuración es realizar las inicializaciones de la máquina. Para nuestra placa, el código específico se encuentra en `.../arch/arm/mach_at91`.

El siguiente paso importante es el procesamiento de la *línea de órdenes del kernel*. Esto seguramente nos sonará si hemos visto alguna vez los mensajes de arranque de Linux. La línea de órdenes nos da la posibilidad de pasar ciertos parámetros al kernel para alterar o configurar alguna funcionalidad específica. Esto proporciona mucha flexibilidad al sistema ya que, en teoría, podemos utilizar una configuración diferente cada vez que arrancamos la máquina sin tener que modificar el kernel. Para localizar la línea durante el arranque miramos cómo después de los mensajes sobre la información del procesador aparece una línea similar a la siguiente:

```
Kernel command line: console=ttyS0.115200 ip=bootp root=/dev/nfs
```

En este ejemplo se le está diciendo al kernel durante el arranque que abra una consola en el puerto serie `ttys0`, con una tasa de 115Kbps. Además se le ordena que obtenga su IP inicial de un servidor bootp y que monte el sistema de archivos raíz a través del protocolo NFS.

Normalmente Linux se arranca mediante un cargador de arranque con una serie de parámetros que forman la *línea de órdenes* del kernel. En realidad no es una línea de órdenes como las que estamos acostumbrados a usar, ya que no invocamos al kernel desde una shell, sino que son los cargadores de arranque los que la utilizan para pasar los parámetros al kernel. En algunas plataformas que no tienen un cargador de arranque capaz de realizar esta operación, se pueden definir los parámetros de arranque en tiempo de compilación de forma que queden fijados en el código como parte de la imagen del kernel. En otras plataformas (lo más común) la línea de órdenes la puede modificar el usuario sin tener que recompilar el kernel. El cargador de arranque que haya instalado construye la línea de órdenes a partir de un archivo de configuración (por ejemplo, `menu.lst` en el caso de usar Grub) o bien puede permitir al usuario crear una línea de órdenes antes del arranque.

Después de que `start_kernel()` realice sus tareas de inicialización, se crea la primera hebra kernel. Dicha hebra es la que al final acabará siendo la conocida como “proceso `init()`”, con PID 1. A cualquiera que conozca las generalidades de Linux le sonará este proceso, ya que es muy importante para el sistema. Por encima de todo, este proceso es conocido por ser el padre de todos los procesos de usuario.

En este momento del arranque habrá dos hebras vivas: la que está ejecutando la función `start_kernel()` y la hebra `init()`. La primera acabará convirtiéndose en el proceso ocioso (`idle`). La función que realiza estos pasos es `rest_init()`³. En ella, primero se crea la hebra `init` con una llamada a `kernel_thread`, a partir de aquí `init` completa el resto de la inicialización mientras la hebra que comenzó a ejecutar `start_kernel` llama por primera vez al planificador y finalmente se queda en un bucle infinito en la llamada `cpu_idle`.

El motivo de que la bifurcación se haga en una función fuera de `start_kernel` es que dicha función tiene un tamaño considerable, y merece la pena poder reutilizar la memoria que está utilizando, así que se marca para que al final del arranque se reclame

³Ver el listado A.1 en el apéndice

esa memoria. Sin embargo, para poder recuperarla, antes es necesario salir de la función y del espacio de direcciones que ocupa. Esto se soluciona llamando a una función más pequeña (`rest_init()`) que será la que quedará como el proceso ocioso.

Los últimos pasos del arranque, realizados ya por la hebra `init`, incluyen la liberación de la memoria usada por las funciones y los datos de inicialización, abrir una consola e iniciar el primer proceso de usuario. Esto se hace pasando un ejecutable a la llamada `run_init_process`, que habitualmente suele ser el binario `/sbin/init`, lo cual provoca la creación del proceso de usuario `/sbin/init` que se encarga de los pasos antes mencionados. Como vemos, estos programas se buscan en un `path`, por lo que de alguna manera tendremos que tener un sistema de archivos raíz que contiene a alguno de esos archivos en las rutas especificadas, y dicho sistema de archivos debe haber sido montado por el kernel en algún momento anterior dentro del proceso de arranque que hemos visto. En el siguiente capítulo hablaremos del sistema de archivos raíz.

Si en la línea de órdenes del kernel se encontró el parámetro `init`, se intentará lanzar a este como proceso `init`. Si esto falla o no se utilizó el parámetro, se prueba una serie de ejecutables por defecto⁴. Si ninguno de ellos tiene éxito se termina la ejecución de `main.c` con un *kernel panic*.

La ventaja de poder modificar el parámetro `init` en el arranque es que podemos decirle al kernel que utilice el programa de inicialización que nosotros queramos, por ejemplo, uno hecho a la medida de nuestras necesidades. O incluso se puede lanzar una shell en caso de emergencia con `init=/bin/sh` para realizar tareas de recuperación.

⁴Ver el código A.2 en el apéndice

5

Inicialización del sistema

En este capítulo continuaremos la secuencia de inicialización donde la dejamos en el anterior. Después de que el kernel quede inicializado, tiene que montar un sistema de archivos raíz y ejecutar una serie de rutinas de inicialización del sistema. A continuación estudiaremos los detalles de este proceso.

5.1. El sistema de archivos raíz

Unas de las principales características de Unix (y, por lo tanto, de Linux) es que basa la mayor parte de su funcionalidad en el sistema de archivos. Además de los propios archivos, un gran número de aspectos del sistema, como los dispositivos y los procesos, están modelados como archivos. No debe de extrañar, por tanto, que Linux *necesite* un sistema de archivos raíz.

El sistema de archivos raíz se refiere al sistema de archivos montado en la base de la jerarquía del sistema de archivos (*/*). Es de una importancia fundamental para el sistema y hay ciertos requisitos que debe cumplir, lo que lo diferencia del resto de sistemas de archivos de bloques. Linux espera que el sistema de archivos raíz contenga los programas necesarios para arrancar el sistema y configurar los servicios necesarios, así que tendremos que construir el sistema de archivos de una forma predeterminada para que Linux pueda encontrar en él los archivos que necesita.

Para organizar la estructura del sistema de archivos raíz está el estándar FHS. Este establece un mínimo de compatibilidad entre las distintas distribuciones de Linux y los programas mediante la definición de los directorios principales y sus contenidos. En el estándar FHS todos los archivos y directorios aparecen bajo el directorio raíz “*/*”,

incluso aunque se encuentren almacenados en dispositivos físicos distintos, y hay una serie de directorios principales bien definidos que siempre deberían encontrarse en el sistema y contener ciertos archivos imprescindibles [11]. Por ejemplo, un sistema de archivos raíz común contendría los directorios `bin`, `dev`, `etc`, `lib`, `sbin`, `usr`, `var` y `tmp`. Aunque podríamos tener un sistema de archivos raíz tan sencillo como este:

```
.
|--bin
| |--busybox
| '---sh -> busybox
|--dev
| '---console
|--etc
| '---init.d
| '---rcS
'-- lib
   |--ld-2.3.2.so
   |--ld-linux.so.2 -> ld-2.3.2.so
   |--libc-2.3.2.so
   '---libc.so.6 -> libc-2.3.2.so
```

que con sólo 8 archivos nos permitiría arrancar el sistema con una consola serie a través del dispositivo `/dev/console` y tener una *shell*, que estaría implementada dentro de *busybox*. Puede que de momento no entendamos qué son los otros archivos que aparecen en este árbol de directorios, pero lo tendremos claro a medida que avancemos en el proyecto. De momento basta con saber que Busybox es un paquete de aplicaciones especialmente diseñado para su uso en sistemas empotrados y que implementa un gran número de utilidades a través de un solo ejecutable. En este caso lo utilizaremos para ejecutar una shell. Los archivos que hay dentro del directorio `lib` son las bibliotecas dinámicas necesarias para que los programas en C (particularmente *busybox* en este ejemplo) puedan ejecutarse.

En un sistema de escritorio la construcción del sistema de archivos raíz –suponiendo que no esté automatizada– no plantea muchos problemas, ya que consistiría en ir formando la estructura de directorios e ir añadiendo los programas y utilidades necesarios para el arranque. Sin embargo, en un sistema empotrado, debido a las restricciones de almacenamiento esta cuestión se complica al tener el compromiso de construir un sistema de archivos raíz que nos ofrezca el máximo de funcionalidad con el menor tamaño posible.

Aparte de la complicación del diseño del sistema de archivos (qué metemos y qué dejamos fuera), están los quebraderos de cabeza a la hora de añadir y quitar aplicaciones, ya que muchas consisten en paquetes con un gran número de archivos, además de archivos de configuración y documentación, bases de datos, etc. A esto hay que sumar el esfuerzo de llevar la cuenta de las dependencias de los programas, tanto de bibliotecas como de otros programas, algo que puede ocurrir en cascada.

Con paciencia podemos realizar esta tarea a mano a base de prueba y error, aunque recientemente han surgido herramientas que automatizan la construcción de sistemas de archivos, facilitando enormemente la tarea. De este modo podemos escoger fácilmente qué paquetes queremos incluir y la propia herramienta se encargará de las dependencias y de eliminar los archivos que no vayamos a necesitar, ahorrando espacio.

Entre estas herramientas se encuentra *buildroot*, que está especialmente diseñada para construir, entre otras cosas, sistemas de archivos raíz para su uso en sistemas empotrados que utilicen la biblioteca *uclibc* en lugar de la típica *libc*. Buildroot tiene un sistema de construcción basado en menús similar al del kernel, y tiene muchas opciones para generar tanto el sistema de archivos raíz como un entorno de desarrollo utilizando un gran número de herramientas y versiones distintas de las mismas. De momento nos interesa sólo construir el sistema de archivos raíz. En los siguientes informes veremos cómo construir un sistema de archivos raíz a nivel práctico.

5.2. El proceso init

Ya sabemos que *init* es el primer proceso de usuario, que lo crea el kernel al completar la secuencia de arranque y que es el primero en la jerarquía de procesos del sistema. De hecho, *init* proporciona unos parámetros de entorno por defecto que son heredados por el resto de procesos.

Salvo que queramos hacer cosas muy poco usuales, nunca necesitaremos crear nuestro propio proceso inicial porque *init* es muy flexible. Realiza su tarea junto con una familia de scripts de inicialización, implementando de esta forma lo que se conoce como el *esquema de inicialización de System V*. *Init* se encuentra siempre en un nivel de ejecución o *runlevel* (ver la tabla 5.1) y dependiendo de en cuál se encuentre lanza unos scripts u otros que son los que definen las acciones que se llevan a cabo. De este modo podemos configurar un determinado *runlevel* sin más que añadir, borrar o modificar los scripts que están asociados a él.

Runlevel	Propósito
0	Apagado del sistema
1	Configuración monousuario para mantenimiento
2	Definido por el usuario
3	Configuración multiusuario de propósito general
4	Definido por el usuario
5	Multiusuario con GUI al inicio
6	Reinicio del sistema
S	Usado para inicializar el sistema en el arranque

Tabla 5.1: Componentes de la imagen del kernel para la arquitectura ARM

Los scripts se encuentran normalmente en el directorio `/etc/rc.d/init.d` o en `/etc/init.d`, dependiendo de la distribución. Estos scripts permiten iniciar y detener los servicios del sistema y se pueden invocar manual e individualmente. Por ejemplo, si quisiéramos arrancar el servidor web Apache haríamos:

```
% /etc/init.d/apache2 start
```

Un runlevel está definido por los servicios que están activados en él. Normalmente hay una estructura de directorios en `/etc/rc.d` (o en `/etc`, dependiendo de la distribución) que contiene enlaces simbólicos a los scripts de `/etc/init.d`. Cada uno de los subdirectorios de `/etc/rc.d` contiene enlaces simbólicos a los scripts que deberán ser ejecutados cuando se entre en el runlevel asociado a ese subdirectorio. Por ejemplo, en Debian los subdirectorios asociados a los runlevels son `/etc/rc0.d`, `/etc/rc1.d`, `/etc/rc2.d`, etc. Dentro de cada uno hay una serie de enlaces simbólicos de nombre “Knumservicio” o “Snumservicio” que apuntan a scripts de `/etc/init.d` y que provocan el inicio (si el enlace empieza por S) o la parada (si empieza por K) de un determinado servicio con la prioridad que indica “num”. De este modo, si dentro de `/etc/rc0.d` está el enlace `K01gdm`, cuando pasemos al runlevel 0, `init` parará el servicio `gdm`.

Para saber qué runlevel corresponde a qué directorio, `init` mira el archivo `/etc/inittab`.

`/etc/inittab` contiene directivas que se aplican a todos los runlevels además de directivas propias de cada runlevel por separado. El listado A.3 del apéndice A es un

 Para más información mirar **man init** y **man inittab**. Se pueden encontrar ejemplos de `inittab` en cualquier distribución de Linux en uso.

ejemplo de un `/etc/inittab`. En él se define el runlevel por defecto, el primer script que será ejecutado (para realizar la inicialización y configuración del sistema), los scripts que serán ejecutados para cada runlevel, y también hace que `init` lance una shell en la consola.

5.3. Disco RAM inicial

El kernel de Linux soporta un mecanismo para montar un sistema de archivos raíz primario para realizar ciertas tareas de inicialización durante el arranque. Este mecanismo se conoce como disco RAM inicial o *initrd* (de *initial ram disk*) y lo vimos de pasada en el capítulo 2.

5.3.1. `initrd`

El `initrd` es un pequeño sistema raíz autocontenido que normalmente contiene lo necesario para cargar algunos drivers antes de que termine el ciclo de arranque. Estos drivers suelen ser los necesarios para montar un dispositivo en el que se encuentra el verdadero sistema de archivos raíz.

Para poder usar esta funcionalidad, se debe marcar en las opciones de configuración del kernel cuando lo construimos. Una vez que disponemos de un kernel preparado para arrancar con `initrd`, necesitamos un cargador de arranque que pase la imagen del `initrd` al kernel. Una forma común de hacerlo consiste en que el cargador de arranque carga la imagen del kernel en memoria y después carga un `initrd` en otra sección de la memoria, tras lo cual informa al kernel de la dirección donde ha cargado el `initrd` antes de que el kernel tome el control. Si el sistema con el que trabajamos no dispone de un cargador de arranque con esta capacidad, podemos construir una imagen compuesta del kernel concatenado con el `initrd`. Sea cual sea el método utilizado, el kernel debe conocer siempre dónde se encuentra cargado el `initrd`.

Para indicar al kernel dónde se encuentra el `initrd` se utiliza la opción `initrd` en la línea de órdenes del kernel. Por ejemplo, la opción `initrd=0x10800000,0x14af47` indica al kernel que debe cargar el `initrd` que se encuentra a partir de la dirección física `0x10800000`, el cuál tiene un tamaño de `0x14af47` (1355591 bytes).

Cuando el kernel arranca de esta manera, detecta la presencia de la imagen del `initrd` y copia el binario de la dirección física especificada a un disco ram creado por

el kernel y monta en este el sistema de archivos raíz. Dentro de la imagen del `initrd` se encuentra un archivo llamado `linuxrc`. Cuando el kernel monta el disco `ram` busca en él este archivo y lo trata como un script, ejecutando todo lo que hay en él. Esto significa que el comportamiento y la función del `initrd` viene definido por lo que pongamos en el archivo `linuxrc`. Por ejemplo, podríamos incluir en él las instrucciones necesarias para cargar los drivers para leer una tarjeta de memoria donde podríamos tener un sistema de archivos raíz funcional.

Cuando `linuxrc` se termina de ejecutar, el kernel desmonta el `initrd` y continúa con las últimas etapas de la inicialización del sistema. Si el verdadero sistema de archivos raíz contiene un directorio llamado `/initrd`, Linux monta el `initrd` en ese punto de montaje. En caso contrario, la imagen `initrd` simplemente se descarta.

Si el parámetro `root` de la línea de órdenes del kernel especifica un disco `ram` (por ejemplo, `root=/dev/ram0`), el procedimiento es diferente. En primer lugar se omite la ejecución de `linuxrc`. En segundo lugar, no se intenta montar ningún otro sistema de archivos raíz distinto al `initrd`, con lo que tendríamos un sistema Linux funcionando con el `initrd` como único sistema de archivos. Esto es útil para cuando queramos tener una configuración mínima con un disco `ram` como sistema de archivos raíz.

5.3.2. `initramfs`

`Initramfs` es un mecanismo utilizado a partir del kernel 2.6 para ejecutar programas en espacio de usuario en las primeras etapas de funcionamiento del sistema. Su propósito es similar al de `initrd`: permitir la carga de los drivers que pudieran ser necesarios para montar el verdadero sistema de archivos raíz. Sin embargo, la implementación es muy diferente a la de `initrd`.

Desde el punto de vista práctico, `initramfs` es más fácil de usar. `Initramfs` está integrado en el árbol de directorios de los fuentes del kernel y se construye automáticamente cuando construimos el kernel. De este modo, hacer cambios en `initramfs` es más sencillo que construir y cargar una nueva imagen `initrd` cada vez que queremos cambiar algo.

En el directorio `.../usr` es donde se construye la imagen `initramfs`. Dentro de este directorio se encuentra el archivo `initramfs_list`, que contiene la lista de archivos que serán incluidos en el archivo `initramfs`. Su contenido por defecto en las versiones recientes del kernel es algo así:

```
dir /dev 0755 0 0
nod /dev/console 0600 0 0 c 5 1
dir /root 0700 0 0
```

Esto producirá una pequeña estructura de directorios que contendrá a los directorios `/root` y `/dev`, junto con un dispositivo que representa la consola.

La salida de este directorio será el archivo `initramfs_data_cpio.gz`, un archivo comprimido que contiene los archivos especificados en `initramfs_list`. Este archivo será enlazado a la imagen final del kernel, lo cual es otra ventaja de `initramfs` frente a `initrd`: no es necesario cargar la imagen del disco ram de forma separada al arrancar el kernel.

A

Scripts y código fuente

Código A.1 Creación de la hebra init

```
432 static void ninline __init_refok rest_init(void)
    __releases(kernel_lock)
{
435     int pid;

    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
440     kthreadd_task = find_task_by_pid(pid);
    unlock_kernel();

    /*
     * The boot idle thread must execute schedule()
445     * at least once to get things moving:
     */
    init_idle_bootup_task(current);
    preempt_enable_no_resched();
    schedule();
450     preempt_disable();

    /* Call into cpu_idle with preempt disabled */
    cpu_idle();
}
```

Código A.2 Últimos pasos del arranque en main.c

```
795 if (execute_command) {  
    run_init_process(execute_command);  
    printk(KERN_WARNING "Failed to execute %s. Attempting "  
        "defaults...\n", execute_command);  
}  
800 run_init_process("/sbin/init");  
    run_init_process("/etc/init");  
    run_init_process("/bin/init");  
    run_init_process("/bin/sh");  
805 panic("No init found. Try passing init= option to kernel.");
```

Código A.3 Ejemplo de /etc/inittab

```
1 # /etc/inittab: init(8) configuration.

# Runlevel por defecto.
4 id:2:initdefault:

# Script de inicializacion/configuracion durante el arranque.
# Es el primero que se ejecuta excepto cuando se arranca en modo de
# emergencia (-b)
si::sysinit:/etc/init.d/rcS

9
# Directorios con los scripts a ejecutar para cada runlevel
# wait significa que init debe esperar a que el script termine antes de
# seguir
10:0:wait:/etc/init.d/rc 0
11:1:wait:/etc/init.d/rc 1
14 12:2:wait:/etc/init.d/rc 2
13:3:wait:/etc/init.d/rc 3
14:4:wait:/etc/init.d/rc 4
15:5:wait:/etc/init.d/rc 5
16:6:wait:/etc/init.d/rc 6

19
# Esta entrada lanza una login shell en la consola
# Respawn significa que se volvera a lanzar cada vez que muera
con:2:respawn:/bin/sh
```

B

Archivos del código del kernel mencionados

Archivo	Propósito
.../System.map	Listado de los símbolos del kernel en texto plano
.../arch/arm/boot/compressed/vmlinux	Imagen del kernel
.../arch/arm/kernel/head.o	Punto de entrada al kernel
.../arch/arm/kernel/head.S	Archivo fuente de head.o
.../arch/arm/boot/compressed/head.o	Realiza tareas de inicialización específicas de la arquitectura y es parte del bootstrap loader
.../arch/arm/boot/compressed/head.S	Archivo fuente de head.o (del bootstrap loader)
.../arch/arm/boot/compressed/head-xxx.o	Realiza tareas de inicialización específicas de la arquitectura junto con .../arch/arm/boot/compressed/head.o
.../arch/arm/boot/compressed/head-common.S	Archivo fuente incluido desde head.S
.../arch/arm/boot/Image	Imagen del kernel sin símbolos ni comentarios

Tabla B.1: Archivos nombrados

Archivos del código del kernel mencionados

35

Archivo	Propósito
.../arch/arm/boot/compressed/piggy.gz	El archivo Image comprimido con gzip
.../arch/arm/boot/compressed/piggy.o	El archivo piggy.gz en formato ensamblador
.../arch/arm/boot/compressed/piggy.S	Archivo fuente de piggy.o
.../arch/arm/boot/compressed/misc.o	Rutinas para descomprimir la imagen del kernel (piggy.gz)
.../arch/arm/boot/compressed/big-endian.o	Rutina en ensamblador para configurar el procesador en modo big-endian
.../vmlinuz	Imagen compuesta del kernel
.../arch/arm/boot/zImage	Imagen compuesta final para ser cargada por un cargador de arranque
.../init/main.c	Punto de inicio de la ejecución del kernel
.../arch/arm/kernel/setup.c	Realiza tareas de inicialización del procesador
/sbin/init	Programa encargado de la inicialización del sistema tras el arranque del kernel
/etc/inittab	Directivas para el programa init
linuxrc	Script que se ejecuta cuando se monta un initrd

Tabla B.2: Archivos nombrados (cont.)

Bibliografía

- [1] Christopher Hallinan, *Embedded Linux primer : a practical, real-world approach* Prencice Hall, 2007
- [2] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel, 3ª edición* O'Reilly, 2005
- [3] Ricardo Cañuelo Navarro, *El enlazador y el formato ELF*
- [4] Linux Kernel Newbies, *FAQ / Kernel Compilation*
<http://kernelnewbies.org/FAQ/KernelCompilation>
- [5] *Linux Kernel 2.6 Howto*
http://armin.emx.at/kernel_2.6/kernel_2.6_howto.html
- [6] *Magic SysRq key*
http://en.wikipedia.org/wiki/Magic_SysRq_key
- [7] *The Linux Kernel Archives*
<http://www.kernel.org>
- [8] *Buildroot: making Embedded Linux easy*
<http://buildroot.uclibc.org>
- [9] *Linux System Administrators Guide*
<http://tldp.org/LDP/sag/html/init-intro.html>
- [10] *Busybox README*
<http://www.busybox.net/downloads/README>
- [11] *Filesystem Hierarchy Standard*
<http://www.pathname.com/fhs/pub/fhs-2.3.html>