Curso de Linux Embebido. Segunda parte.

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Derechos de copia

Introducción a Linux Embebido

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Nacimiento del software libre

- 1983, Richard Stallman, **proyecto GNU** y concepto de **software libre**. Comienzo del desarrollo de *gcc*, *gdb*, *glibc* y otras herramientas importantes
- 1991, Linus Torvalds, **proyecto Linux kernel**, un kernel de sistema operativo parecido a Unix. Junto con el software GNU y muchos otros componentes de código abierto: un sistema operativo completamente libre, GNU/Linux
- 1995, Linux es cada vez más popular en servidores
- 2000, Linux es cada vez más popular en **sistemas embebidos**
- 2008, Linux es cada vez más popular en dispositivos móviles
- 2010, Linux es cada vez más popular en teléfonos

¿Sofware libre?

- Un programa se considera **libre** cuando su licencia ofrece a todos sus usuarios las siguientes **cuatro** libertades
  - Libertad para correr el software con cualquier propósito
  - Libertad para estudiar el software y cambiarlo
  - Libertad para redistribuir copias
  - Libertad para distribuir copias de versiones modificadas
- Estas libertades están garantizadas tanto para usos comerciales como no comerciales
- Implican la disponibilidad del código fuente, el software puede ser modificado y distribuido a los clientes
- **Buena elección para sistemas embebidos**

¿Qué es Linux embebido?

Linux embebido es el uso del **kernel de Linux** y varios componentes **open-source** (de código abierto) en sistemas embebidos

Ventajas de Linux y el código abierto para los sistemas embebidos

Reutilizar componentes

- La ventaja clave de Linux y el código abierto en los sistemas embebidos es la capacidad de reutilizar componentes
- El ecosistema open-source ya proporciona muchos componentes para características estándar, desde soporte hardware hasta protocolos de red, pasando por multimedia, gráficos, librerías criptográficas, etc.
- En cuanto un dispositivo hardware, o protocolo, o característica, se extiende, hay una alta probabilidad de contar con componentes de código abierto que lo soporten.
- Permite diseñar y desarrollar rápidamente productos complicados, basados en componentes existentes.
- Nadie tiene que redesarrollar otro kernel de sistema operativo, pila TCP/IP, USB, u otra librería gráfica.
- **Permite concentrarse en el valor añadido del producto.**

Bajo coste

- El software libre puede estar duplicado en tantos componentes como se quiera, sin coste.
- Si un sistema embebido solo usa software libre, se puede reducir el coste de las licencias de software a cero. Incluso las herramientas de desarrollo son gratis, a menos que se decida utilizar una edición de Linux comercial.
- **Permite tener un mayor presupuesto para el hardware o incrementar las habilidades y conocimiento de la empresa**

Total control

- Con el open-source, se tiene el código fuente de todos los componentes del sistema
- Permite modificaciones, cambios, ajustes, depuraciones y optimizaciones ilimitadas, durante un periodo de tiempo ilimitado
- Sin bloqueos o dependencias de proveedores externos
  - Lo cierto es que los componentes que no sean de código abierto deberían evitarse cuando se diseña y desarrolla el sistema
- **Permite tener control total sobre la parte software del sistema**

Calidad

- Muchos componentes de código abierto son ampliamente utilizados, en millones de sistemas
- Normalmente mayor calidad que lo que puede producir un desarrollo casero, o incluso proveedores propietarios
- Por supuesto, no todos los componentes de código abierto son de buena calidad, pero la mayoría de los que se usan comúnmente lo son.
- **Permite diseñar el sistema con componentes de alta calidad en los cimientos**

Facilita el testeo de nuevas características

- Puesto que el open-source está libremente disponible, es fácil conseguir un trozo de código y evaluarlo
- Permite estudiar fácilmente varias opciones cuando se toma una decisión
- Mucho más fácil que los procedimientos de compra y demostración que se necesitan con la mayoría de productos propietarios
- **Permite explorar fácilmente nuevas posibilidades y soluciones**

Soporte de la comunidad

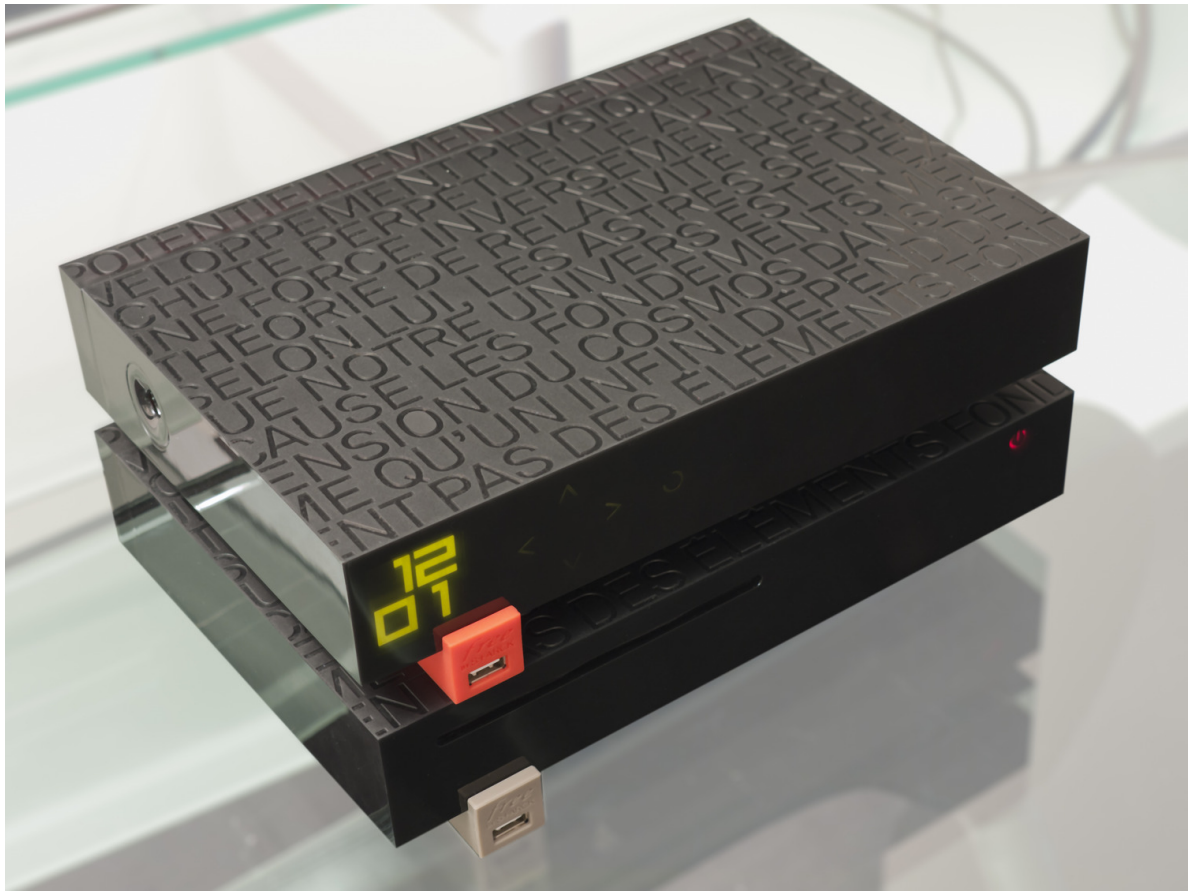
- Los componentes open-source se desarrollan en comunidades de desarrolladores y usuarios
- Esta comunidad puede proporcionar soporte de alta calidad: se puede contactar directamente con los desarrolladores principales del componente que se está usando. La probabilidad de conseguir una respuesta no depende de la compañía con la que se trabaja.
- A menudo, es mejor que el soporte tradicional, pero se necesita entender como funciona la comunidad para usar correctamente las posibilidades de soporte de la misma
- **Permite acelerar la resolución de problemas al desarrollar el sistema**

Formar parte de la comunidad

- Posibilidad de formar parte de la comunidad de desarrollo de algunos de los componentes que se usan en sistemas embebidos: reporte de bugs, prueba de nuevas versiones o características, parches que arreglan bugs o añaden nuevas características, etc.
- La mayoría del tiempo los componentes open-source no son el núcleo de valor del producto: es el interes de todo el mundo por devolver las colaboraciones
- Para los *ingenieros*: una forma muy **motivadora** de ser reconocidos fuera de la empresa, comunicación con otros del mismo campo, **apertura de nuevas posibilidades**, etc.
- Para los *directivos*: **factor de motivación** para los ingenieros, permite a la compañía ser **reconocida** en la comunidad open-source y, por tanto, conseguir soporte más fácilmente y ser **más atractiva** para los desarrolladores open-source

Unos cuantos ejemplos de sistemas que funcionan bajo Linux

Routers personales



Televisores

Terminal Point of Sale (POS)

Máquina de corte láser

Máquina de viticultura

Hardware embebido para sistemas Linux

Procesador y arquitectura (1)

- El kernel de Linux y la mayoría de los demás componentes dependientes de la arquitectura soportan un rango amplio de arquitecturas de 32 y 64 bits
  - x86 y x86-64, como se encuentra en las plataformas PC, pero también en sistemas embebidos (multimedia, industrial)
  - ARM, con cientos de SoC diferentes (multimedia, industrial)
  - PowerPC (principalmente aplicaciones industriales de tiempo real)
  - MIPS (principalmente aplicaciones de red)
  - SuperH (principalmente aplicaciones de decodificadores de TV y multimedia)
  - Blackfin (arquitectura DSP)
  - Microblaze (soft-core para Xilinx FPGA)
  - Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R

Procesador y arquitectura (2)

- Tanto las arquitecturas MMU (memory management unit) como las no MMU están soportadas, aunque las arquitecturas no MMU tienen algunas limitaciones.
- Linux no está diseñado para microcontroladores pequeños.
- Excepto el toolchain, el bootloader y el kernel, todos los demás componentes son, generalmente, **independientes de la arquitectura**.

RAM y almacenamiento

- **RAM**: un sistema Linux muy básico puede funcionar con 8 MB de RAM, pero un sistema más realista requerirá, al menos, 32 MB of RAM. Depende del tipo y el tamaño de las aplicaciones.
- **Almacenamiento**: un sistema Linux muy básico puede funcionar con 4 MB de almacenamiento, pero suele necesitarse más.
  - Se soporta el almacenamiento flash, tanto NAND como NOR, con sistemas de ficheros específicos
  - El almacenamiento de bloques, incluyendo tarjetas SD/MMC y eMMC, está soportado
- No es necesariamente interesante ser demasiado restrictivo en la cantidad de RAM/almacenamiento: tener flexibilidad en este nivel permite reutilizar tantos componentes como sea posible.

Comunicación

- El kernel de Linux tiene soporte para muchos buses comunes de comunicación
  - I2C
  - SPI
  - CAN
  - 1-wire
  - SDIO
  - USB
- Y también soporte de red extensivo
  - Ethernet, Wifi, Bluetooth, CAN, etc.
  - IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
  - Firewalling, advanced routing, multicast

Tipos de plataformas hardware

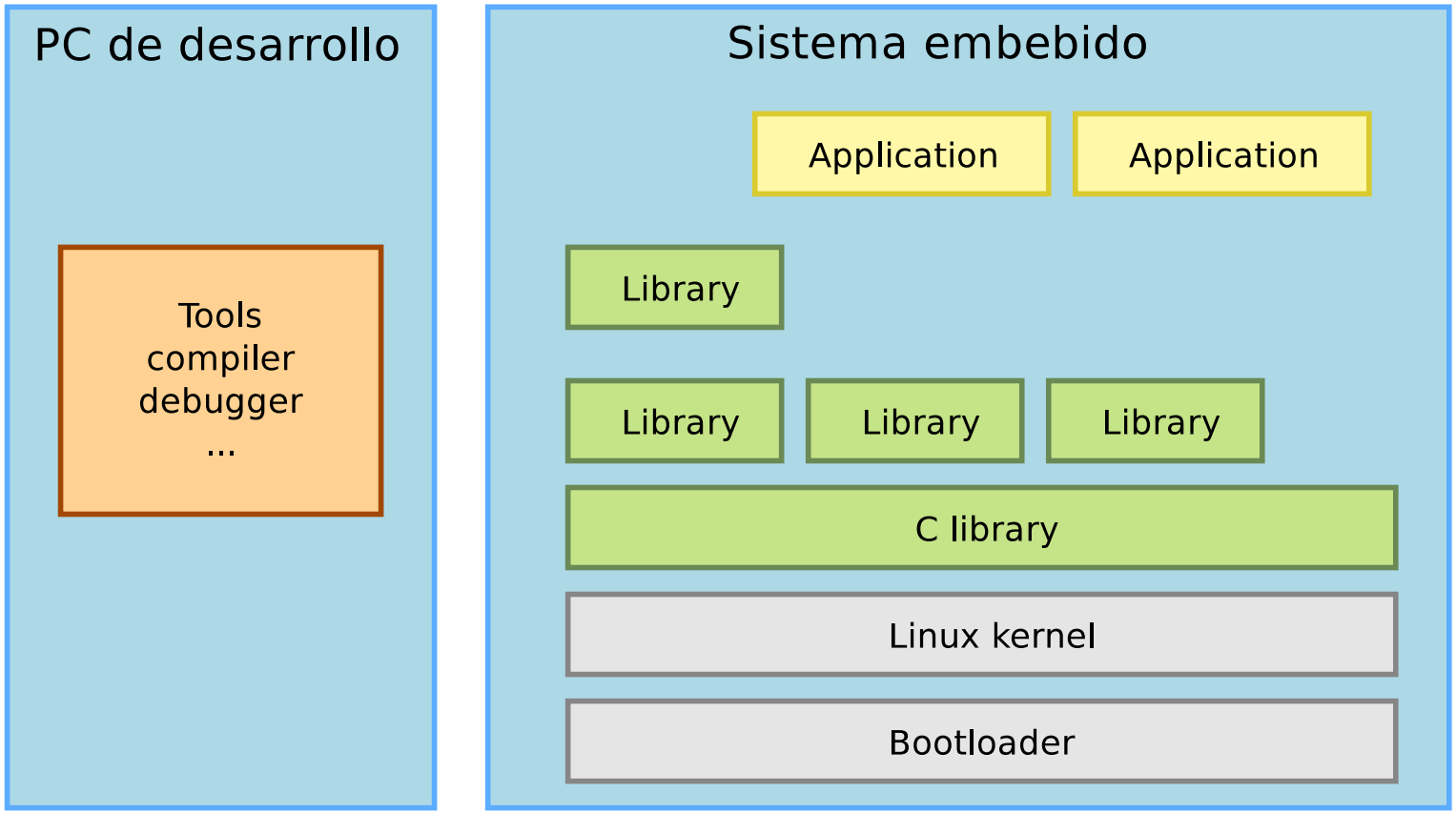
- **Plataformas de evaluación** del proveedor del SoC. Normalmente caro, pero con muchos periféricos integrados. Generalmente, no es adecuado para productos reales.
- **Componente en un módulo**, una placa pequeña con solo CPU/RAM/flash y otros componentes básicos, con conectores para acceder a todos los demás periféricos. Puede usarse para construir productos finales en series pequeñas/medias.
- **Plataformas de desarrollo de la comunidad**, una nueva tendencia para hacer popular y fácilmente disponible un SoC particular. Están preparadas para usarse y son de bajo coste, pero normalmente tienen menos periféricos que las de evaluación. Pueden usarse en productos reales.
- **Plataforma personalizada**. Los esquemáticos para las placas de evaluación o las plataformas de desarrollo están cada vez más frecuentemente disponibles, facilitando el desarrollo de plataformas personalizadas.

Criterios para escoger el hardware

- Asegurarse de que el hardware que se piensa utilizar ya está soportado por el kernel de Linux, y tiene un bootloader de código abierto, especialmente el SoC al que se orienta el desarrollo.
- Tener soporte en las versiones oficiales de los proyectos (kernel, bootloader) es mucho mejor: la calidad es mayor y están disponibles nuevas versiones.
- Algunos proveedores de SoC y/o proveedores de placas no contribuyen añadiendo sus cambios a la línea principal de desarrollo del kernel de Linux. Conviene pedirles que lo hagan, o usar otro producto. Una buena medida es ver la diferencia entre su kernel y el oficial.
- **Entre hardware correctamente soportado en el Linux oficial y hardware mal soportado, habrá grandes diferencias en cuanto a tiempo y coste de desarrollo.**

Arquitectura de un sistema Linux embebido

Arquitectura global



Componentes software

- Toolchain de compilación cruzada
  - Compilador que se ejecuta en la máquina de desarrollo, pero genera código para el objetivo
- Bootloader
  - Iniciado por el hardware, responsable de la inicialización básica y la carga y ejecución del kernel
- Kernel de Linux
  - Contiene la gestión de procesos y memoria, la pila de red, los drivers de dispositivo y proporciona servicios a las aplicaciones del espacio de usuario
- Librería de C
  - La interfaz entre el kernel y las aplicaciones del espacio de usuario
- Librerías y aplicaciones
  - De terceras partes o propias

Trabajo en Linux embebido

Varias tareas distintas son necesarias cuando se despliega Linux embebido en un producto:

- **Desarrollo del paquete de soporte de la placa - Board Support Package**
  - Un BSP contiene un bootloader y un kernel con los drivers de dispositivo adecuados para el hardware objetivo
- **Integración del sistema**
  - Integrar todos los componentes, bootloader, kernel, librerías y aplicaciones de terceras partes y aplicaciones propias en un sistema que funcione
- **Desarrollo de aplicaciones**
  - Aplicaciones normales de Linux, pero usando librerías específicamente seleccionadas

Bootloaders

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

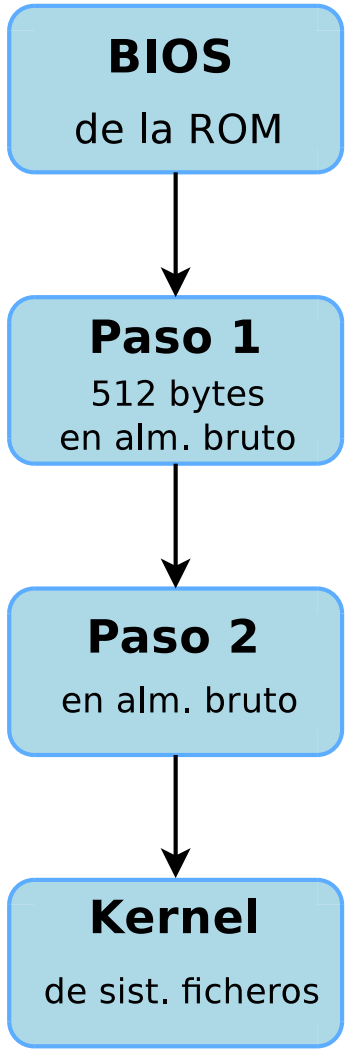
Secuencia de arranque

Bootloaders

- El bootloader es una pieza de código responsable de
  - La inicialización hardware básica
  - La carga del binario de una aplicación, normalmente un kernel de sistema operativo, desde un almacenamiento flash, desde red, o desde otro tipo de almacenamiento no volátil.
  - Posiblemente, la descompresión del binario de la aplicación
  - La ejecución de la aplicación
- Más allá de estas tres funciones básicas, la mayoría de los bootloaders proporcionan un shell con varios comandos que implementan diferentes operaciones.
  - Carga de datos desde almacenamiento o red, inspección de memoria, diagnóstico y prueba de hardware, etc.

Bootloaders en x86 (1)

- Los procesadores x86 habitualmente se incluyen en una placa con una memoria no volátil que contiene un programa, la BIOS.
- Este programa es ejecutado por la CPU después de un reset, y es responsable de la inicialización hardware básica y la carga de una pequeña pieza de código desde un almacenamiento no volátil.
  - Este trozo de código suelen ser los primeros 512 bytes de un medio de almacenamiento
- Esta pieza de código normalemente es un bootloader de primer nivel, que cargará el bootloader en sí.
- El bootloader ofrece entonces todas sus características. Suele entender los formatos de sistemas de ficheros para que el kernel pueda cargarse directamente desde uno de ellos.

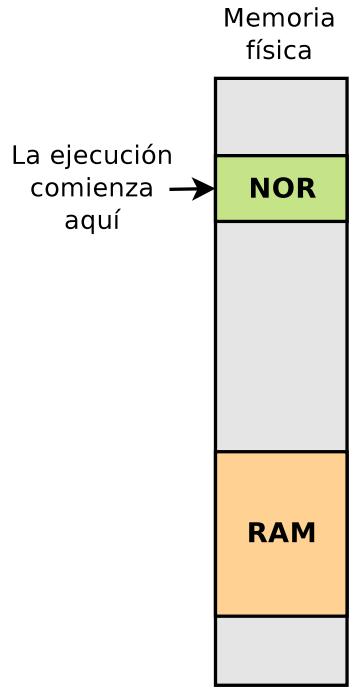


Bootloaders en x86 (2)

- GRUB, Grand Unified Bootloader, el más potente.
  `http://www.gnu.org/software/grub/`
  - Puede leer muchos formatos de sistemas de ficheros para cargar la imagen del kernel y la configuración, proporciona un shell potente con varios comandos, puede cargar imágenes del kernel por red, etc.
- Syslinux, para arranque desde red y medios extraíbles (pendrive, CD-ROM)
  `http://www.kernel.org/pub/linux/utils/boot/syslinux/`

Arranque en CPUs embebidas: caso 1

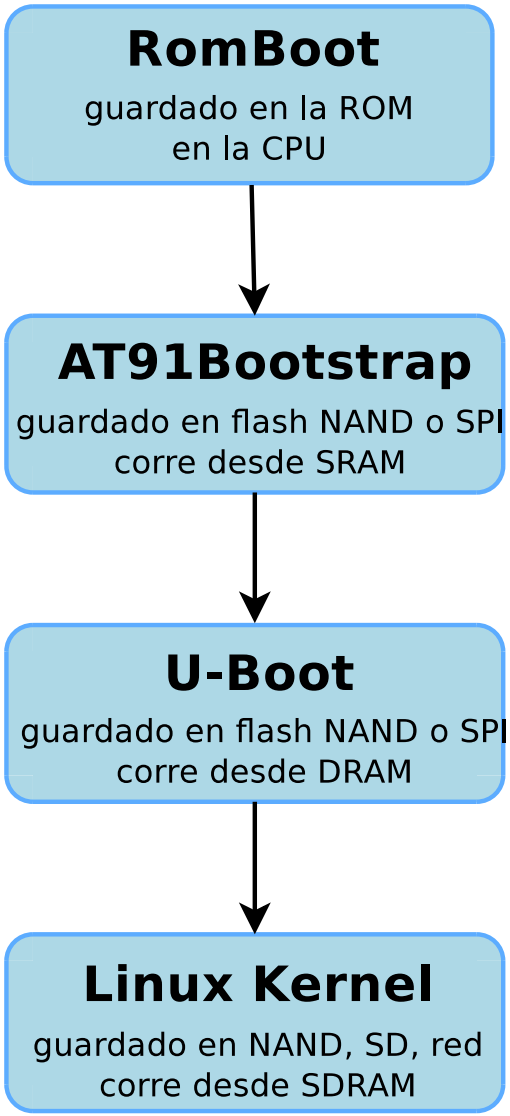
- Cuando se alimenta, la CPU empieza a ejecutar código en una dirección fija
- No hay otro mecanismo de arranque proporcionado por la CPU
- El diseño hardware debe asegurar que un chip flash NOR está cableado para que sea accesible en la dirección en la cual la CPU empieza a ejecutar instrucciones
- El bootloader de primer nivel debe programarse en esa dirección de la NOR
- La NOR es obligatoria, porque permite acceso aleatorio, al contrario que la NAND
- **Ya no es muy común** (poco práctico, y requiere flash NOR)



Arranque en CPUs embebidas: caso 2

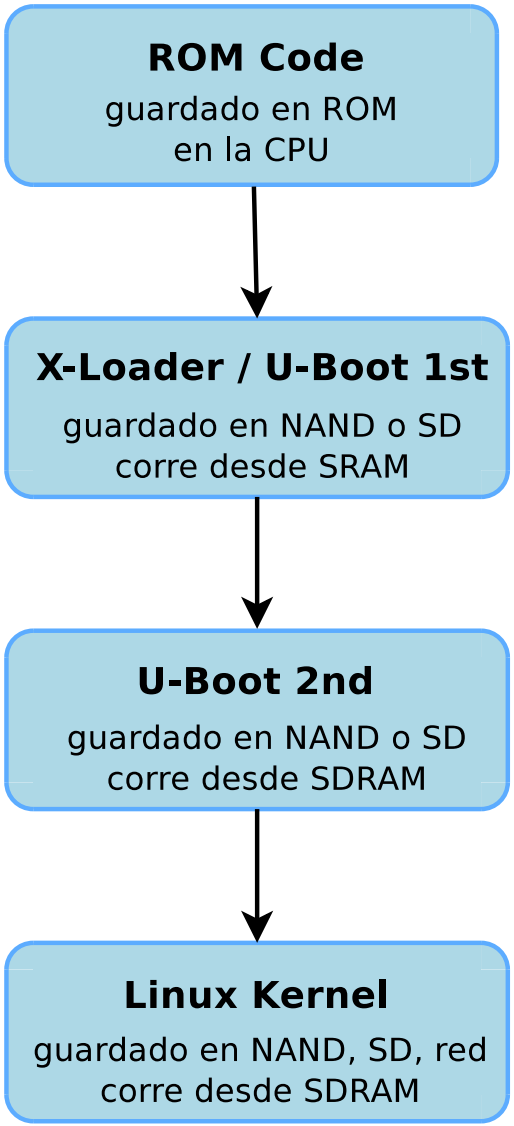
- La CPU tiene un código de arranque integrado en la ROM
  - BootROM en CPUs AT91, "ROM code" en OMAP, etc.
  - Los detalles exactos dependen de la CPU
- Este código de arranque es capaz de cargar un bootloader de primer nivel desde un dispositivo de almacenamiento dentro de una SRAM interna (la DRAM no está inicializada aún)
  - El dispositivo de almacenamiento puede ser: MMC, NAND, flash SPI, UART, etc.
- El bootloader de primer nivel es
  - Limitado en tamaño debido a restricciones hardware (tamaño SRAM)
  - Proporcionado o bien por el vendedor de la CPU o bien a través de proyectos de la comunidad
- El bootloader de primer nivel debe inicializar la DRAM y otros dispositivos hardware y cargar un bootloader de segundo nivel en la RAM

Arranque en ARM Atmel AT91



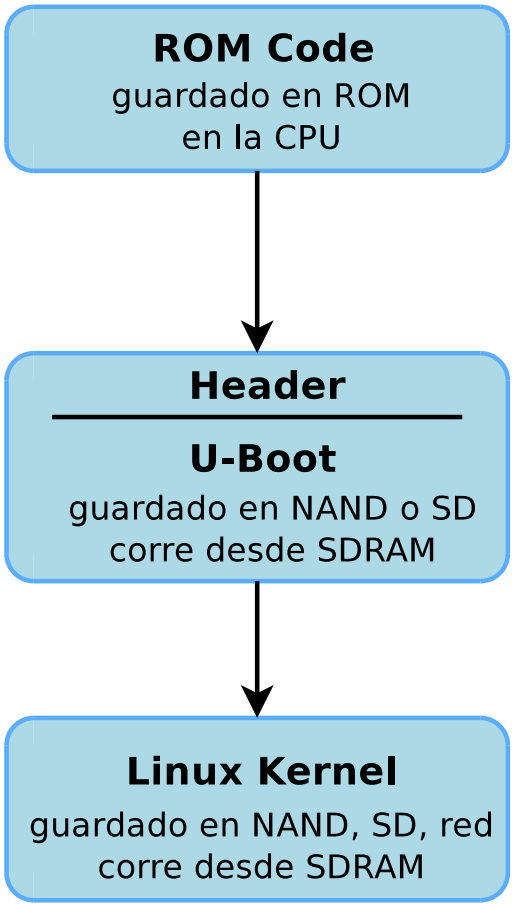
- **RomBoot**: intenta encontrar una imagen bootstrap válida desde varias fuentes de almacenamiento, y la carga en SRAM (la DRAM no está inicializada aún). Tamaño limitado a 4 KB. No es posible la interacción con el usuario en modo de arranque estándar.
- **AT91Bootstrap**: corre desde SRAM. Inicializa la DRAM, el controlador NAND o SPI, y carga el bootloader secundario en la RAM y lo inicia. No es posible la interacción con el usuario.
- **U-Boot**: corre desde RAM. Inicializa otros dispositivos hardware (red, USB, etc.). Carga la imagen del kernel desde una unidad de almacenamiento o red a la RAM y lo inicia. Proporciona un shell con comandos.
- **Linux Kernel**: corre desde RAM. Toma control sobre el sistema completamente (el bootloader deja de existir).

Arrancando en ARM TI OMAP3



- **ROM Code**: intenta encontrar una imagen bootstrap válida desde varias fuentes de almacenamiento, y la carga en SRAM o RAM. Tamaño limitado a <64 KB. No es posible la interacción con el usuario.
- **X-Loader** o **U-Boot**: corre desde SRAM. Inicializa la DRAM, los controladores NAND o MMC, y carga el bootloader secundario a la RAM y lo inicia. No es posible la interacción con el usuario. Archivo llamado `MLO`.
- **U-Boot**: corre desde RAM. Inicializa otros dispositivos hardware (red, USB, etc.). Carga la imagen del kernel desde un medio de almacenamiento o red a la RAM y lo inicia. Proporciona un shell con comandos. Archivo llamado `u-boot.bin` o `u-boot.img`.
- **Linux Kernel**: corre desde RAM. Toma el control sobre el sistema completamente (los bootloaders dejan de existir).

Arranque en Marvell SoC



- **ROM Code**: intenta encontrar una imagen bootstrap válida desde varias fuentes de almacenamiento, y la carga en la RAM. La configuración de la RAM se describe en un header específico de la CPU, antepuesto a la imagen del bootloader.
- **U-Boot**: corre desde RAM. Inicializa otros dispositivos hardware (red, USB, etc.). Carga la imagen del kernel desde un medio de almacenamiento o red a la RAM y lo inicia. Se proporciona un shell con comandos. Archivo llamado `u-boot.kwb`.
- **Linux Kernel**: corre desde RAM. Toma control sobre el sistema completamente (los bootloaders dejan de existir).

Bootloaders genéricos para CPUs embebidas

- Nos centraremos en la parte genérica, el bootloader principal, ofreciendo las características más importantes.
- Hay varios bootloaders genéricos de código abierto.
  Aquí están los más populares:
  - **U-Boot**, el bootloader universal de Denx
    El más utilizado en ARM, también se usa en PPC, MIPS, x86, m68k, NIOS, etc. Es el estándar de facto hoy en día. Lo estudiaremos en detalle.
    `http://www.denx.de/wiki/U-Boot`
  - **Barebox**, un nuevo bootloader multi-arquitectura, escrito como sucesor de U-Boot. Mejor diseñado, mejor código, desarrollo activo, pero aún no tiene tanto soporte hardware como U-Boot.
    `http://www.barebox.org`
- Además, hay muchos más bootloaders de código abierto y propietarios, a veces específicos de la arquitectura
  - RedBoot, Yaboot, PMON, etc.

El bootloader U-boot

U-Boot

U-Boot es un proyecto típico de software libre

- Licencia: GPLv2 (la misma que Linux)
- Disponible libremente en `http://www.denx.de/wiki/U-Boot`
- Documentación disponible en `http://www.denx.de/wiki/U-Boot/Documentation`
- EL último código fuente en desarrollo está disponible en un repositorio Git: `http://git.denx.de/?p=u-boot.git;a=summary`
- El desarrollo y las discusiones se producen en torno a una lista de correo abierta `http://lists.denx.de/pipermail/u-boot/`
- Desde finales de 2008, sigue un calendario de lanzamientos fijo. Cada dos meses, se lanza una nueva versión. Las versiones se llaman `AAAA.MM`.

Configuración U-Boot

- Conseguir el código fuente del sitio web y descomprimirlo
- El directorio `include/configs/` contiene un archivo de configuración para cada placa soportada
  - Define el tipo de CPU, los periféricos y su configuración, el mapeado de memoria, las características de U-Boot que deben compilarse, etc.
  - Es un archivo `.h` simple que fija constantes para el preprocesador de C. Véase el archivo `README` para documentación de estas constantes. Este archivo también puede ser ajustado para añadir o eliminar características de U-Boot (comandos, etc.).
- Asumiendo que una cierta placa ya está soportada por U-Boot, debería haber una entrada correspondiente a la misma en el archivo `boards.cfg`.

Extracto del archivo de configuration U-Boot

```
/* CPU configuration */
#define CONFIG_ARMV7 1
#define CONFIG_OMAP 1
#define CONFIG_OMAP34XX 1
#define CONFIG_OMAP3430 1
#define CONFIG_OMAP3_IGEP0020 1
[...]
/* Memory configuration */
#define CONFIG_NR_DRAM_BANKS 2
#define PHYS_SDRAM_1 OMAP34XX_SDRC_CS0
#define PHYS_SDRAM_1_SIZE (32 <<  20)
#define PHYS_SDRAM_2 OMAP34XX_SDRC_CS1
[...]
/* USB configuration */
#define CONFIG_MUSB_UDC 1
#define CONFIG_USB_OMAP3 1
#define CONFIG_TWL4030_USB 1
[...]
```

```
/* Available commands and features */
#define CONFIG_CMD_CACHE
#define CONFIG_CMD_EXT2
#define CONFIG_CMD_FAT
#define CONFIG_CMD_I2C
#define CONFIG_CMD_MMC
#define CONFIG_CMD_NAND
#define CONFIG_CMD_NET
#define CONFIG_CMD_DHCP
#define CONFIG_CMD_PING
#define CONFIG_CMD_NFS
#define CONFIG_CMD_MTDPARTS
[...]
```

Configurar y compilar U-Boot

- U-Boot debe configurarse antes de compilarse
  - `make BOARDNAME_config`
  - Donde `BOARDNAME` es el nombre de la placa, tal y como aparece en el archivo `boards.cfg` (primera columna).
- Asegurarse de que el compilador cruzado está disponible en `PATH`
- Compilar U-Boot, especificando el prefijo del compilador cruzado.
  Por ejemplo, si el ejecutable del compilador cruzado es `arm-linux-gcc`:
  `make CROSS_COMPILE=arm-linux-`
- El resultado principal es un archivo `u-boot.bin`, que es la imagen U-Boot. Dependiendo de la plataforma específica, podría haber otras imágenes especializadas: `u-boot.img`, `u-boot.kwb`, `MLO`, etc.

Instalar U-Boot

- U-Boot debe ser instalado, normalmente, en la memoria flash para ser ejecutado por el hardware. Dependiendo del hardware, la instalación de U-Boot se hace de forma diferente:
  - La CPU proporciona algún tipo de monitor de arranque con el cual es posible comunicarse a través del puerto serie o USB usando un protocolo específico.
  - La CPU arranca primero en un medio extraíble (MMC) antes de arrancar de un medio fijo (NAND). En este caso, arrancar desde MMC para reflashear una nueva versión
  - U-Boot ya está instalado, y puede usarse para flashear una nueva versión de U-Boot. Sin embargo, hay que tener cuidado: si la nueva versión de U-Boot no funciona, la tarjeta no puede usarse
  - La placa proporciona una interfaz JTAG, que permite escribir a la memoria flash remotamente, sin ningún sistema corriendo en la placa. Permite también rescatar una placa si no funciona el bootloader

Línea de comandos U-boot

- Conectar la placa al anfitrión a través de una consola serie
- Alimentar la placa. En la consola serie, se leerá algo como:

```
U-Boot 2013.04 (May 29 2013 - 10:30:21)

OMAP36XX/37XX-GP ES1.2, CPU-OPP2, L3-165MHz, Max CPU Clock 1 Ghz
IGEPv2 + LPDDR/NAND
I2C:   ready
DRAM:  512 MiB
NAND:  512 MiB
MMC:   OMAP SD/MMC: 0

Die ID #255000029ff800000168580212029011
Net:   smc911x-0
U-Boot #
```

- El shell de U-Boot ofrece un conjunto de comandos. Estudiaremos los más importantes, véase la documentación para la referencia completa o el comando `help` .

Comandos de información

Información Flash (Flash NOR y SPI)

```
U-Boot> flinfo
DataFlash:AT45DB021
Nb pages: 1024
Page Size: 264
Size= 270336 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C0001FFF (RO) Bootstrap
Area 1: C0002000 to C0003FFF Environment
Area 2: C0004000 to C0041FFF (RO) U-Boot
```

Información flash NAND

```
U-Boot> nand info
Device 0: nand0, sector size 128 KiB
  Page size      2048 b
  OOB size         64 b
  Erase size   131072 b
```

Detalles de la versión

```
U-Boot> version
U-Boot 2013.04 (May 29 2013 - 10:30:21)
```

Comandos importantes (1)

- El conjunto exacto de comandos depende de la configuración U-Boot
- `help` y `help comando`
- `boot`, ejecuta el comando de arranque por defecto, almacenado en `bootcmd`
- `bootm <direccion>`, arranca una imagen del kernel cargada en una dirección dada en RAM
- `ext2load`, carga un archivo de un sistema de ficheros ext2 a la RAM
  - También `ext2ls` para listar los ficheros, `ext2info` para información
- `fatload`, carga un archivo desde un sistema de ficheros FAT a la RAM
  - También `fatls` y `fatinfo`
- `tftp`, carga un archivo desde la red a la RAM
- `ping`, para probar la red

Comandos importantes (2)

- `loadb`, `loads`, `loady`, carga un archivo de la línea serie a la RAM
- `usb`, para inicializar y controlar el subsistema USB, principalmente se usa para los dispositivos de almacenamiento USB como pendrives
- `mmc`, para inicializar y controlar el subsistema MMC, usado en tarjetas SD y microSD
- `nand`, para borrar, leer y escribir contenidos a la flash NAND
- `erase`, `protect`, `cp`, para borrar, modificar y escribir a la flash NOR
- `md`, muestra los contenidos de la memoria. Puede ser útil para comprobar los contenidos cargados en memoria, o para observar los registros hardware.
- `mm`, modifica los contenidos de la memoria. Puede ser útil para modificar directamente registros hardware, con fines de testeo.

Commandos de variables de entorno (1)

- U-Boot puede configurarse a través de variables de entorno, que afectan al comportamiento de los diferentes comandos.
- Las variables de entorno se cargan desde la flash a la RAM en el arranque de U-Boot, pueden ser modificadas y guardadas de nuevo en la flash si se quiere que persistan
- Hay una localización dedicada en la flash (o en almacenamiento MMC) para guardar el entorno de U-Boot, definido en el archivo de configuración de la placa

Comandos de variables de entorno (2)

Comandos para manipular variables de entorno:

- `printenv`
  Muestra todas las variables
- `printenv <variable-name>`
  Muestra el valor de una variable
- `setenv <variable-name> <variable-value>`
  Cambia el valor de una variable, solo en RAM
- `editenv <variable-name>`
  Edita el valor de una variable, solo en RAM
- `saveenv`
  Guarda el estado actual del entorno en la flash

Comandos de variables de entorno - Ejemplo

```
u-boot # printenv
baudrate=19200
ethaddr=00:40:95:36:35:33
netmask=255.255.255.0
ipaddr=10.0.0.11
serverip=10.0.0.1
stdin=serial
stdout=serial
stderr=serial
u-boot # printenv serverip
serverip=10.0.0.2
u-boot # setenv serverip 10.0.0.100
u-boot # saveenv
```

Variables de entorno de U-Boot importantes

- `bootcmd`, contiene el comando que U-Boot ejecutará automáticamente en el arranque después de un retardo configurable, si el proceso no se interrumpe
- `bootargs`, contiene los argumentos que se pasan al kernel de Linux, se tratará más tarde
- `serverip`, la dirección IP del servidor que contactará U-Boot para los comandos relativos a la red
- `ipaddr`, la dirección IP que usará U-Boot
- `netmask`, la máscara de red para contactar con el servidor
- `ethaddr`, la dirección MAC, solo puede fijarse una vez
- `bootdelay`, el retardo en segundos antes deuqe U-Boot ejecute `bootcmd`
- `autostart`, si es *yes*, U-Boot arranca automáticamente una imagen que ha sido cargada en memoria

Scripts en variables de entorno

- Las variables de entorno puede contener pequeños scripts, para ejecutar algunos comandos y comprobar el resultado de los comandos.
  - Útil para arranque automático o procesos de actualización
  - Se pueden encadenar varios comandos con el operador `;`
  - Se pueden hacer pruebas usando
    `if command ; then ... ; else ... ; fi`
  - Los scripts se ejecutan usando `run <nombre-variable>`
  - Se pueden referenciar otras variables usando
    `${nombre-variable}`
- Ejemplo
  - `setenv mmc-boot 'if fatload mmc 0 80000000 boot.ini; then source; else if fatload mmc 0 80000000 uImage; then run mmc-bootargs; bootm; fi; fi'`

Transferir archivos a la placa

- U-Boot se usa principalmente para cargar y arrancar una imagen del kernel, pero también permite cambiar la imagen del kernel y el sistema de ficheros raíz guardado en la flash
- Los archivos han de ser intercambiados entre la placa y la estación de desarrollo. Esto es posible:
  - A través de la red si la placa tiene conexión Ethernet, y U-Boot contiene un driver para el chip Ethernet. Esta es la solución más rápida y eficiente.
  - A través de un pendrive, si U-Boot soporta el controlador USB de la plataforma
  - A través de una tarjeta SD o microSD, si U-Boot soporta el controlador MMC de la plataforma
  - A través del puerto serie

TFTP

- La transferencia de red desde la estación de desarrollo y U-Boot tiene lugar a través de TFTP
  - *Trivial File Transfer Protocol*
  - De alguna forma, parecido a FTP, pero sin autentificación y sobre UDP
- Se necesita un servidor TFTP en la estación de desarrollo
  - `sudo apt-get install tftpd-hpa`
  - Todos los archivos en `/var/lib/tftpboot` son, en ese caso, visibles a través de TFTP
  - Está disponible un cliente TFTP en el paquete `tftp-hpa`, como prueba
- Hay un cliente TFTP integrado dentro de U-Boot
  - Configurar las variables de entorno `ipaddr` y `serverip`
  - Usar `tftp <address> <filename>` para cargar un archivo

U-boot mkimage

- La imagen del kernel que carga y arranca U-Boot debe estar preparada, de forma que un header específico se añade delante de la imagen
  - Este header da detalles sobre el tamaño de la imagen, la dirección de carga esperada, el tipo de compresión, etc.
- Esto se hace con una herramienta que viene en U-Boot, `mkimage`
- Debian / Ubuntu: basta instalar el paquete `u-boot-tools`.
- O compilarlo uno mismo: simplemente, configurar U-Boot para cualquier placa de cualquier arquitectura y compilarlo. Entonces, instalar `mkimage`:
  `cp tools/mkimage /usr/local/bin/`
- El objetivo especial `uImage` del Makefile del kernel puede usarse entonces para generar una imagen del kernel adecuada para U-Boot.

Sistema de ficheros raíz de Linux

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Principio y soluciones

Sistemas de ficheros

- Los sistemas de ficheros se usan para organizar datos en directorios y archivos en los dispositivos de almacenamiento o en la red. Los directorios y archivos se organizan de acuerdo a una jerarquía.
- En los sistemas Unix, las aplicaciones y usuarios ven una **única jerarquía global** de archivos y directorios, que puede estar compuesta de múltiples sistemas de ficheros.
- Los sistemas de ficheros se **montan** en una localización específica en esa jerarquía de directorios
  - Cuando se monta un sistema de ficheros en un directorio (llamado *punto de montaje*), los contenidos de este directorio reflejan los contenidos del dispositivo de almacenamiento.
  - Cuando se desmonta el sistema de ficheros, el *punto de montaje* vuelve a estar vacío.
- Esto permite a las aplicaciones acceder a archivos y directorios fácilmente, independientemente de su localización exacta de almacenamiento

Sistemas de ficheros (2)

- Crear un punto de montaje, que es simplemente un directorio

```
$ mkdir /mnt/usbkey
```

- Está vacío

```
$ ls /mnt/usbkey
$
```

- Montar un dispositivo de almacenamiento en este punto de montaje

```
$ mount -t vfat /dev/sda1 /mnt/usbkey
$
```

- Se puede acceder a los contenidos del pendrive

```
$ ls /mnt/usbkey
docs prog.c picture.png movie.avi
$
```

mount / umount

- `mount` permite montar sistemas de ficheros
  - `mount -t type device mountpoint`
  - `type` es el tipo de sistema de ficheros
  - `device` es el dispositivo de almacenamiento, o localización de red, a montar
  - `mountpoint` es el directorio donde los archivos del dispositivo de almacenamiento o localización de red estarán accesibles
  - `mount` sin argumentos muestra los sistemas de ficheros actualmente montados
- `umount` permite desmontar sistemas de ficheros
  - Esto es necesario antes de reiniciar, o antes de desconectar un pendrive, porque el kernel de Linux cachea las escrituras en memoria para incrementar el rendimiento. `umount` asegura que estas escrituras se envían al almacenamiento.

Sistema de ficheros raíz

- Un sistema de ficheros particular se monta en la raíz de la jerarquía, identificada por `/`,
- Este sistema de ficheros se llama el **sistema de ficheros raíz**.
- Como `mount` y `umount` son programas, son archivos dentro de un sistema de ficheros.
  - No están accesibles antes de que se monte, al menos, un sistema de ficheros.
- Como el sistema de ficheros raíz es el primer sistema de ficheros que se monta, no puede montarse con el comando `mount` normal
- Lo monta directamente el kernel, según la opción del kernel `root=`
- Cuando no hay disponible un sistema de ficheros, el kernel falla

```
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)
```

Localización del sistema de ficheros raíz

- Puede montarse desde diferentes localizaciones
  - Desde la partición de un disco duro
  - Desde la partición de un pendrive
  - Desde la partición de una tarjeta SD
  - Desde la partición de un chip flash NAND o un dispositivo de almacenamiento de un tipo similar
  - Desde la red, usando el protocolo NFS
  - Desde memoria, usando un sistema de ficheros precargado (por el bootloader)
  - etc.
- Es tarea del diseñador del sistema elegir la configuración del sistema, y configurar el comportamiento del kernel con `root=`

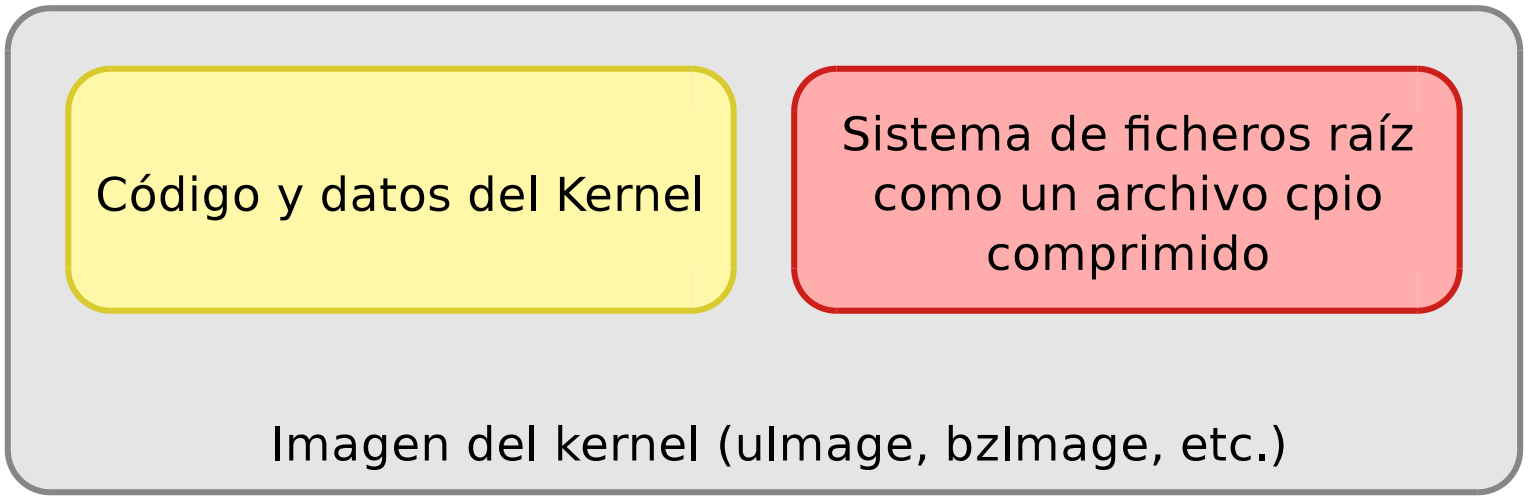
Montar rootfs desde dispositivos de almacenamiento

- Particiones de un disco duro o pendrive
  - `root=/dev/sdXY`, donde `X` es una letra que indica el dispositivo, y `Y` un número que indica la partición
  - `/dev/sdb2` es la segunda partición de la segunda unidad de disco (ya sea pendrive o disco duro ATA)
- Particiones de una tarjeta SD
  - `root=/dev/mmcblkXpY`, donde `X` es un número que indica el dispositivo y `Y` un número que indica la partición
  - `/dev/mmcblk0p2` es la segunda partición del primer dispositivo
- Particiones de un almacenamiento flash
  - `root=/dev/mtdblockX`, donde `X` es el número de partición
  - `/dev/mtdblock3` es la cuarta partición de un chip flash NAND (si solo hay presente un chip flash NAND)

rootfs en memoria: initramfs (1)

- También es posible tener el sistema de ficheros raíz integrado dentro de la imagen del kernel
- Se carga, por tanto, en memoria con el kernel
- Este mecanismo se llama **initramfs**
  - Integra un archivo comprimido del sistema de ficheros en la imagen del kernel
  - Variante: el archivo comprimido también puede ser cargado separadamente por el bootloader.
- Es útil en dos casos
  - Arranque rápido de sistemas de ficheros muy pequeños. Como el sistema de ficheros está completamente cargado en el arranque, el inicio de aplicaciones es muy rápido.
  - Como un paso intermedio antes de cambiar a un sistema de ficheros raíz real, en dispositivos para los cuales se necesitan drivers que no son parte de la imagen del kernel (drivers de almacenamiento, drivers de sistemas de ficheros, drivers de red). Es lo habitual en el kernel de distribuciones de escritorio/servidor.

rootfs en memoria: initramfs (2)



rootfs en memoria: initramfs (3)

- Los contenidos de un initramfs se definen a nivel de configuración del kernel, con la opción `CONFIG_INITRAMFS_SOURCE`
  - Puede ser la ruta a un directorio que contiene el sistema de ficheros raíz
  - Puede ser la ruta a un archivo cpio
  - Puede ser un fichero de texto que describa los contenidos del initramfs
    (véase documentación para más detalles)
- El proceso de compilación del kernel tendrá en cuenta automáticamente los contenidos de la opción `CONFIG_INITRAMFS_SOURCE` e integrará el sistema de ficheros raíz dentro de la imagen del kernel
- Detalles (en las fuentes del kernel):
  `Documentation/filesystems/ramfs-rootfs-initramfs.txt`
  `Documentation/early-userspace/README`

Contenidos

Organización del sistema de ficheros raíz

- La organización de un sistema de ficheros raíz en términos de directorios está bien definida por el **Estándar de Jerarquía de Sistemas de Ficheros** (**Filesystem Hierarchy Standard**)
- `http://www.linuxfoundation.org/collaborate/workgroups/lsb/fhs`
- La mayoría de sistemas Linux se ajustan a esta especificación
  - Las aplicaciones esperan esta organización
  - Facilita la tarea a los desarrolladores y usuarios puesto que la organización del sistema de ficheros es similar en todos los sistemas

Directorios importantes (1)

/bin	Programas básicos
/boot	Imagen del kernel (solo cuando se carga el kernel desde un sistema de ficheros, no es común en arquitecturas distintas de x86)
/dev	Archivos de dispositivos (se verá después)
/etc	Configuración para todo el sistema
/home	Directorio para los directorios home de los usuarios
/lib	Librerías básicas
/media	Puntos de montaje para los medios extraíbles
/mnt	Puntos de montaje para medios estáticos
/proc	Punto de montaje para el sistema de ficheros virtual proc

Directorios importantes (2)

- /root Directorio home del usuario `root`
- /sbin Programas básicos del sistema
- /sys Punto de montaje del sistema de ficheros virtual sysfs
- /tmp Archivos temporales
- /usr
  - /usr/bin Programas no básicos
  - /usr/lib Librerías no básicas
  - /usr/sbin Programas del sistema no básicos
- /var Archivos de datos variables. Incluye datos de logs y administrativos, archivos transitorios y temporales, etc.

Separación de programas y librerías

- Los programas básicos se instalan en `/bin` y `/sbin` y las librerías básicas en `/lib`
- Todos los demás programas se instalan en `/usr/bin` y `/usr/sbin` y todas las demás librerías en `/usr/lib`
- En el pasado, en los sistemas Unix, `/usr` se montaba a menudo sobre la red, a través de NFS
- Para permitir al sistema arrancar cuando la estaba está caída, algunos binarios y librerías se almacenaban en `/bin`, `/sbin` y `/lib`
- `/bin` y `/sbin` contienen programas como `ls`, `ifconfig`, `cp`, `bash`, etc.
- `/lib` contiene la librería de C y, a veces, unas cuantas librerías básicas más
- Todos los demás programas y librerías están en `/usr`

Archivos de dispositivo

Dispositivos

- Uno de las tareas importantes del kernel es **permitir a las aplicaciones acceder a los dispositivos hardware**
- En el kernel de Linux, la mayoría de dispositivos se presentan a las aplicaciones del espacio de usuario a través de dos abstracciones diferentes
  - Dispositivo de **caracteres**
  - Dispositivo de **bloques**
- Internamente, el kernel identifica cada dispositivo por una tripleta de información
  - **Tipo (Type)** (caracteres o bloques)
  - **N. Mayor (Major)** (típicamente la categoría del dispositivo)
  - **N. Menor (Minor)** (típicamente el identificador del dispositivo)

Tipos de dispositivos

- Dispositivos de bloques
  - Un dispositivo compuesto de bloques de tamaño fijo, que puede leerse y escribirse para almacenar datos
  - Usado para discos duros, pendrives, tarjetas SD, etc.
- Dispositivos de caracteres
  - Originalmente, una corriente infinita de bytes, sin principio, final ni tamaño. Ejemplo claro: un puerto serie.
  - Usado para puertos serie, terminales pero también tarjetas de sonido, dispositivos de adquisición de vídeo, frame buffers
  - La mayoría de los dispositivos que no son dispositivos de bloques se representan como dispositivos de caracteres por el kernel de Linux

Sistemas de ficheros virtuales

Sistema de ficheros virtual proc

- El sistema de ficheros virtual `proc` existe desde los comienzos de Linux
- Permite
  - Que el kernel exponga estadísticas sobre los procesos en ejecución en el sistema
  - Que el usuario ajuste varios parámetros del sistema en tiempo de ejecución, acerca de la gestión de procesos, la gestión de memoria, etc.
- El sistema de ficheros `proc` lo usan muchas aplicaciones estándar del espacio de usuario, y esperan que esté montado en `/proc`
- Aplicaciones como `ps` o `top` no funcionarían sin el sistema de ficheros `proc`
- Comando para montar `/proc`:
  `mount -t proc nodev /proc`
- `Documentation/filesystems/proc.txt` en las fuentes del kernel o `man proc`

Contenidos de proc

- Un directorio para cada proceso ejecutándose en el sistema
  - `/proc/<pid>`
  - `cat /proc/3840/cmdline`
  - Contiene detalles sobre los archivos abiertos por el proceso, la CPU y el uso de memoria, etc.
- `/proc/interrupts`, `/proc/devices`, `/proc/iomem`, `/proc/ioports` contiene información general acerca del dispositivo
- `/proc/cmdline` contiene la línea de comandos del kernel
- `/proc/sys` contiene muchos archivos que pueden escribirse para ajustar los parámetros del kernel
  - Se llaman *sysctl*. Véase `Documentation/sysctl/` en las fuentes del kernel.
  - Ejemplo `echo 3 > /proc/sys/vm/drop_caches`

Sistema de ficheros sysfs

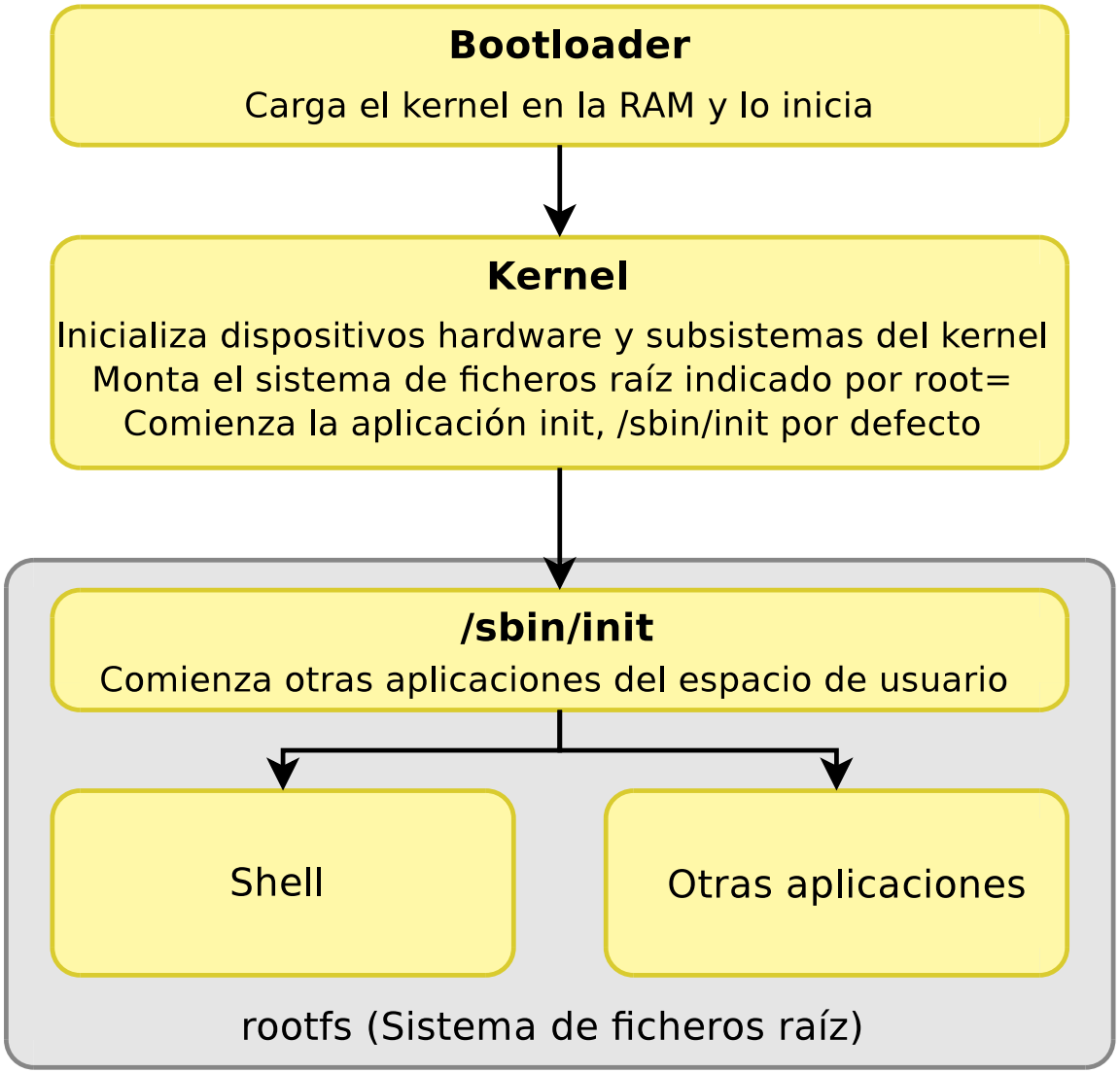
- El sistema de ficheros `sysfs` es una característica que se integró en el kernel de Linux 2.6
- Permite representar en el espacio de usuario la visión que tiene el kernel de los buses, dispositivos y drivers en el sistema
- Es útil para varias aplicaciones del espacio de usuario que necesitan listar y consultar el hardware disponible, por ejemplo `udev` o `mdev`.
- Todas las aplicaciones que usan sysfs esperan que esté montado en el directorio `/sys`
- Comando para montar `/sys`:
  `mount -t sysfs nodev /sys`
- ```
  $ ls /sys/
  block bus class dev devices firmware
  fs kernel module power
  ```
  ```

Sistema de ficheros mínimo

Aplicaciones básicas

- Para funcionar, un sistema Linux necesita al menos unas cuantas aplicaciones
- Una aplicación `init`, que es la primera aplicación del espacio de usuario arrancada por el kernel después de montar el sistema de ficheros raíz
  - El kernel intenta ejecutar `/sbin/init`, `/bin/init`, `/etc/init` y `/bin/sh`.
  - En el caso de un initramfs, solo buscará `/init`. Cualquier otra ruta tiene que darse en el argumento del kernel `rdinit`.
  - Si no se encuentra ninguno de ellos, el kernel falla y el proceso de arranque se para.
  - La aplicación `init` es responsable de arrancar todas las demás aplicaciones y servicios en el espacio de usuario
- Normalmente un shell, para permitir al usuario interactuar
- Aplicaciones Unix básicas, para copiar archivos, mover archivos, listar archivos (`mv`, `cp`, `mkdir`, `cat`, etc.)
- Estos componentes básicos han de estar integrados dentro del sistema de ficheros raíz para hacerlo utilizable

Proceso de arranque general



Introducción al Kernel de Linux

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

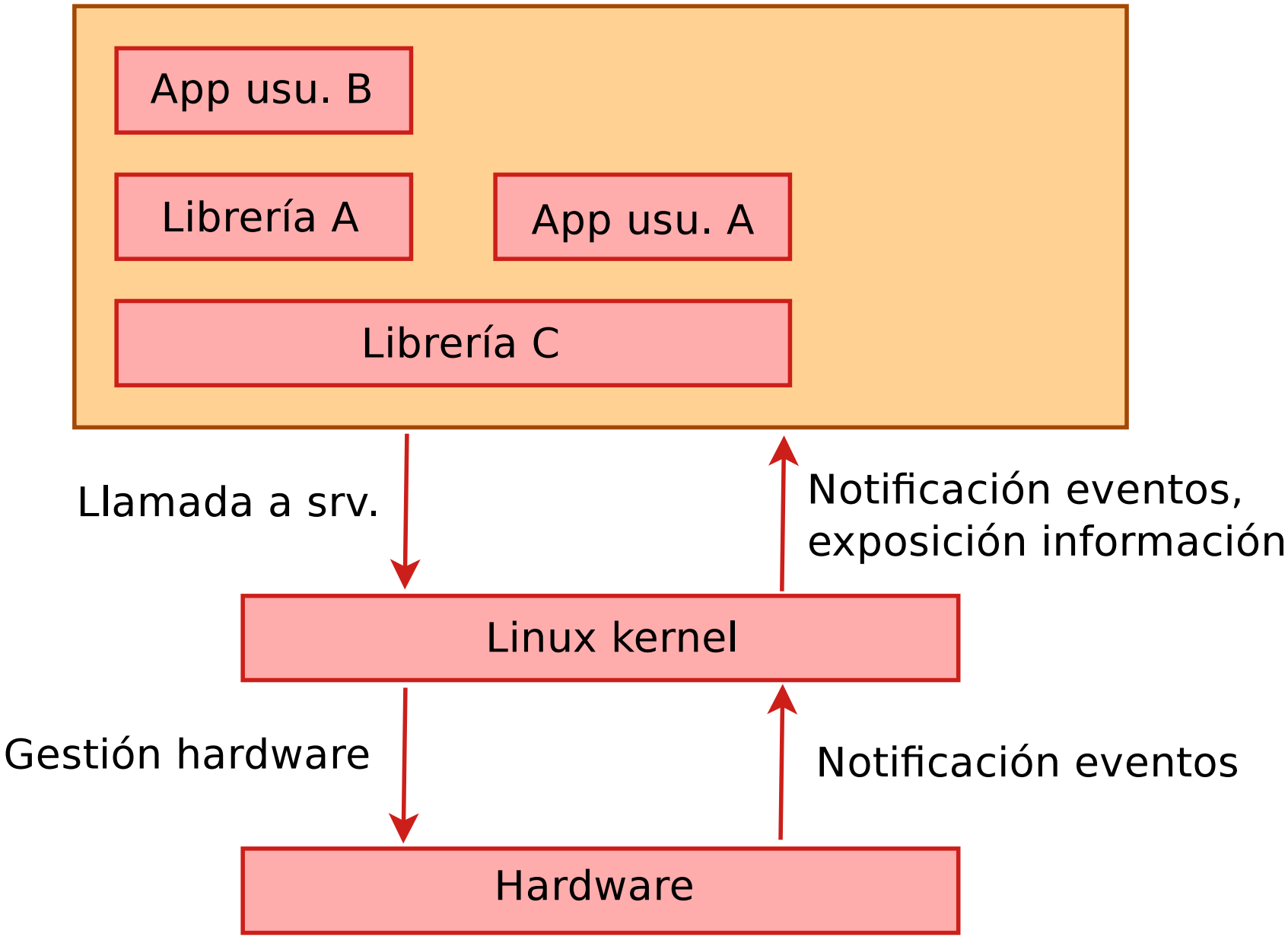
Última actualización: July 10, 2014.

Características de Linux

Historia

- El kernel de Linux es uno de los componentes de un sistema, el cual requiere también librerías y aplicaciones para proporcionar funcionalidades a los usuarios finales.
- El kernel de Linux fue creado como un hobby en 1991 por un estudiante finlandés, Linus Torvalds.
  - Linux empezó rápidamente a utilizarse como kernel de los sistemas operativos de software libre.
- Linus Torvalds ha sido capaz de crear una comunidad de usuarios y desarrolladores grande y dinámica alrededor de Linux.
- Hoy en día, más de mil personas contribuyen a cada liberación del kernel, tanto individuos como empresas grandes y pequeñas.

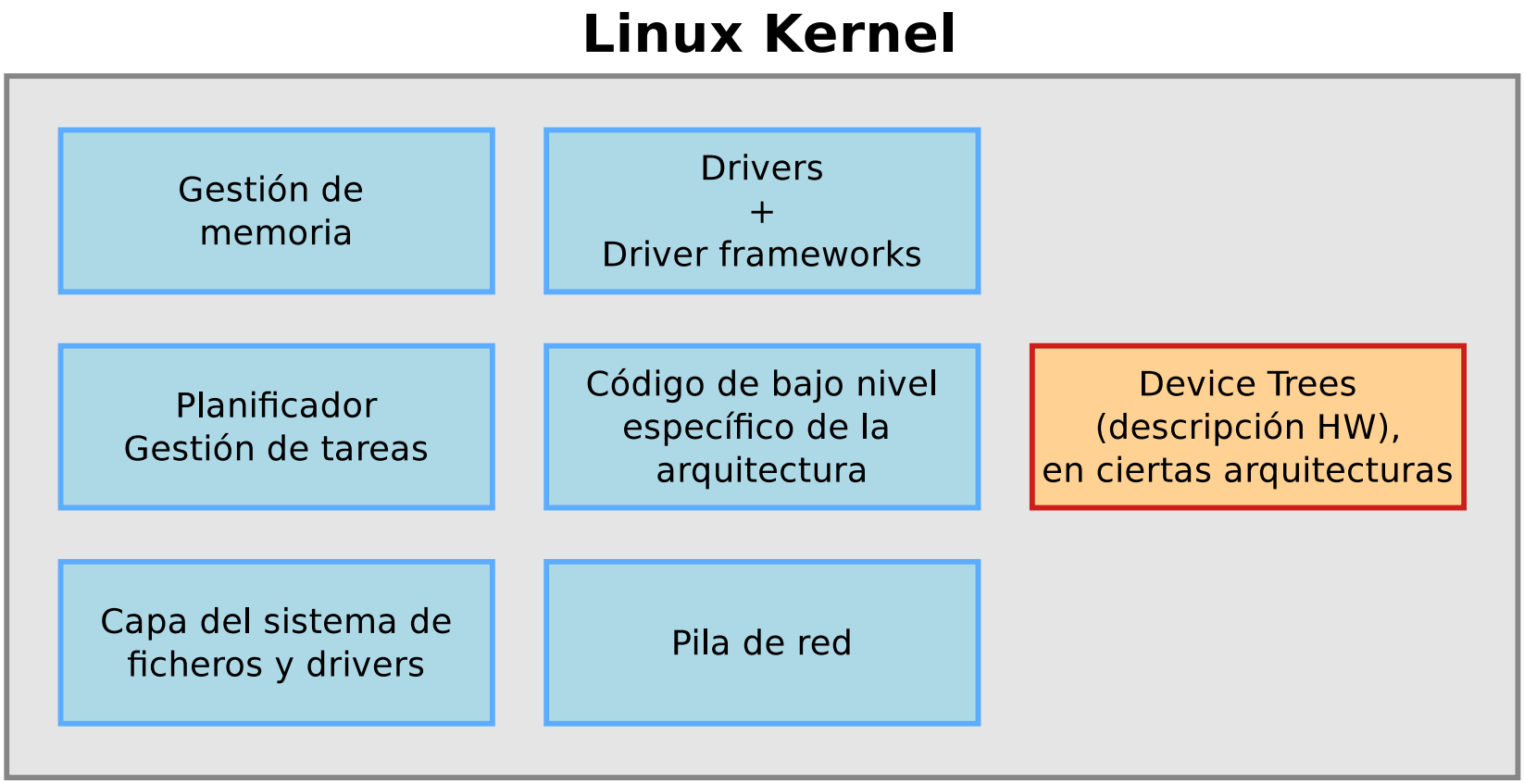
El kernel de Linux en el sistema



Tareas principales del kernel de Linux

- **Gestionar todos los recursos hardware**: CPU, memoria, E/S.
- Proporcionar un **conjunto de APIs portables independientes de la arquitectura y el hardware** para permitir a las aplicaciones y librerías del espacio de usuario usar los recursos hardware.
- **Manejar el acceso y uso concurrente** de los recursos hardware desde diferentes aplicaciones.
  - Por ejemplo: una interfaz de red única es usada por múltiples aplicaciones del espacio de usuario a través de varias conexiones de red. El kernel es responsable de "multiplexar" los recursos hardware.

Dentro del kernel de Linux



Licencia de Linux

- Las fuentes completas de Linux son Software Libre liberado bajo la GNU General Public License version 2 (GPL v2).
- Para el kernel de Linux, esto implica, básicamente, que:
  - Cuando se recibe o compra un dispositivo con Linux en él, se deberían recibir las fuentes de Linux, con derecho a estudiarlas, modificarlas y distribuirlas.
  - Cuando se producen dispositivos basados en Linux, se deben liberar las fuentes al receptor, con los mismos derechos, sin restricciones.

Características clave del kernel de Linux

- Portabilidad y soporte hardware. Corre en la mayoría de arquitecturas.
- Escalabilidad. Puede correr tanto en súper-ordenadores como en dispositivos minúsculos (Bastan 4 MB de RAM).
- Conformidad con los estándares e interoperabilidad.
- Soporte de red exhaustivo.
- Seguridad. No puede ocultar sus defectos. Su código es revisado por muchos expertos.
- Estabilidad y confiabilidad.
- Modularidad. Puede incluir únicamente lo que necesita un sistema incluso en tiempo de ejecución.
- Fácil de programar. Se puede aprender a partir del código existente. Hay infinidad de recursos en la red.

Arquitecturas hardware soportadas

- Véase el directorio `arch/` en las fuentes del kernel
- Mínimo: procesadores de 32 bits, con o sin MMU, y soporte para `gcc`
- Arquitecturas de 32 bits (subdirectorios `arch/`)
  Ejemplos: `arm`, `avr32`, `blackfin`, `c6x`, `m68k`, `microblaze`, `mips`, `score`, `sparc`, `um`
- Arquitecturas de 64 bits
  Ejemplos: `alpha`, `arm64`, `ia64`, `tile`
- Arquitecturas de 32/64 bits
  Ejemplos: `powerpc`, `x86`, `sh`, `sparc`

Llamadas del sistema

- La principal interfaz entre el kernel y el espacio de usuario es el conjunto de llamadas del sistema (*system calls*)
- Alrededor de 300 llamadas del sistema que proporcionan los principales servicios del kernel
  - Operaciones de ficheros y dispositivos, operaciones de red, comunicación entre procesos, gestión de procesos, mapeado de memoria, temporizadores, threads, primitivas de sincronización, etc.
- Esta interfaz es estable en el tiempo: los desarrolladores del kernel solo pueden añadir nuevas llamadas del sistema
- Esta interfaz de llamadas del sistema está envuelta por la librería de C, y las aplicaciones en el espacio de usuario normalmente nunca hacen una llamada del sistema directamente, sino que usan la función correspondiente de las librerías de C

Sistemas de ficheros virtuales

- Linux permite acceder a información sobre el sistema y el kernel a través de sistemas de ficheros virtuales.
- Usar sistemas de ficheros virtuales permite a las aplicaciones ver directorios y ficheros que no existen en un medio de almacenamiento real: el kernel los crea "en el aire".
- Los dos sistemas de ficheros virtuales más importantes son:
  - `proc`, normalmente montado en `/proc`:
    Información relativa al sistema operativo (procesos, parámetros de gestión de memoria...).
  - `sysfs`, normalmente montado en `/sys`:
    Representación del sistema como un conjunto de dispositivos y buses. Información sobre estos dispositivos.

Uso Embebido del Kernel de Linux

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Fuentes del kernel de Linux

Localización de las fuentes del kernel

- Las versiones oficiales del kernel de Linux, tal y como las libera Linus Torvalds, están disponibles en `http://www.kernel.org`
  - Estas versiones siguen el modelo de desarrollo del kernel
  - Sin embargo, pueden no contener aún los últimos desarrollos de un área específica. Algunas características en desarrollo podrían no estar listas para su inclusión en la *mainline*.
- Muchos vendedores de chips proporcionan sus propias fuentes del kernel
  - Centrándose principalemente en el soporte hardware
  - Pueden tener variaciones muy importantes con la *mainline*
  - Útil únicamente cuando la *mainline* no lo ha alcanzado
- Muchas sub-comunidades del kernel mantienen su propio kernel, habitualmente con características más nuevas, pero también menos estables
  - Comunidades de las arquitecturas (ARM, MIPS, PowerPC, etc.), de los drivers de dispositivos (I2C, SPI, USB, PCI, network, etc.), otras comunidades (tiempo real, etc.)
  - No hay lanzamientos oficiales, solo árboles de desarrollo

Consiguiendo las fuentes de Linux

- Las fuentes del kernel están disponibles en `http://kernel.org/pub/linux/kernel` como **full tarballs** (fuentes completas del kernel) y **patches** (diferencias entre dos versiones del kernel).
- Sin embargo, cada vez más gente usa el sistema de control de versiones `git`. Es absolutamente necesario para el desarrollo del kernel.
  - Extraer las fuentes completas del kernel y su histórico
    `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
  - Crear una rama que empieza en una versión estable específica
    `git checkout -b <name-of-branch> v3.11`
  - Interfaz web disponible en `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/`

Tamaño del kernel de Linux (1)

- Fuentes Linux 3.10:
  Tamaño bruto: 573 MB (43,000 ficheros, aprox. 15,800,000 líneas)
  `gzip` fichero tar comprimido: 105 MB
  `bzip2` fichero tar comprimido: 83 MB (mejor)
  `xz` fichero tar comprimido: 69 MB (el mejor)
- Tamaño mínimo del kernel Linux 2.6.29 compilado con `CONFIG_EMBEDDED`, para un kernel que carga un PC QEMU: 532 KB (comprimido), 1325 KB (bruto)
- ¿Por qué son las fuentes tan grandes?
  Porque incluyen miles de drivers de dispositivos, muchos protocolos de red, soporte para muchas arquitecturas y sistemas de ficheros...
- El núcleo fundamental de Linux (planificador, gestión de memoria...) es bastante pequeño

Tamaño del kernel de Linux (2)

En el kernel versión 3.10.

- `drivers/`: 49.4%
- `arch/`: 21.9%
- `fs/`: 6.0%
- `include/`: 4.7%
- `sound/`: 4.4%
- `Documentation/`: 4.0%
- `net/`: 3.9%
- `firmware/`: 1.0%
- `kernel/`: 1.0%
- `tools/`: 0.9%
- `scripts/`: 0.5%
- `mm/`: 0.5%
- `crypto/`: 0.4%
- `security/`: 0.4%
- `lib/`: 0.4%
- `block/`: 0.2%
- ...

Laboratorio - Obtención del Código Fuente del Kernel

- Clonar las fuentes de Hardkernel (ODROID) de Linux con Git

Código Fuente del Kernel

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Código y Drivers de Dispositivos de Linux

Lenguaje de programación

- Implementado en C como todos los sistemas Unix (C se creó para implementar los primeros sistemas Unix)
- Se usa también un poco de ensamblador:
  - Inicialización de la CPU y la máquina, excepciones
  - Rutinas críticas de las librerías
- No se usa C++
- Todo el código se compila con gcc
  - Muchas extensiones específicas de gcc usadas en el código del kernel, un compilador ANSI C cualquiera no compilaría el kernel.
  - Unos cuantos compiladores alternativos están soportados (Intel and Marvell)

Sin librería de C

- El kernel tiene que ser independiente y no puede usar código del espacio de usuario.
- El espacio de usuario está implementado por encima de los servicios del kernel, no al contrario.
- El código del kernel tiene que proporcionar sus propias implementaciones de librerías (utilidades de strings, criptografía, descompresión...)
- Por tanto, no se pueden usar las funciones de la librería estándar de C en el código del kernel (`printf()`, `memset()`, `malloc()`,...).
- Afortunadamente, el kernel proporciona funciones similares de C, como `printk()`, `memset()`, `kmalloc()`, ...

Portabilidad

- El código del kernel de Linux está diseñado para ser portable
- Todo el código fuera de `arch/` debería ser portable
- Con este fin, el kernel proporciona macros y funciones para abstraer los detalles específicos de la arquitectura
  - Endianness
    - `cpu_to_be32()`
    - `cpu_to_le32()`
    - `be32_to_cpu()`
    - `le32_to_cpu()`
  - Acceso memoria I/O
  - Barreras de memoria para proporcionar garantías de orden si se necesita
  - API DMA para limpiar e invalidar cachés si se necesita.

Sin computación en coma flotante

- No se deben usar nunca números en coma flotante en el código del kernel. El código podría correr en un procesador sin unidad de coma flotante (como ciertas CPUs ARM).
- No se debe confundir con las opciones de configuración relativas a la coma flotante
  - Esas están referidas a la emulación de operación de la coma flotante llevada a cabo por las aplicaciones del espacio de usuario, disparando una excepción en el kernel.
  - Usar soft-float, es decir, emulación en el espacio de usuario, está sin embargo recomendado por razones de rendimiento.

Sin API interna de Linux estable 1/3

- La API interna del kernel para implementar código del kernel puede experimentar cambios entre dos lanzamientos estables 2.6.x o 3.x. Un driver independiente compilado para una versión dada podría no volver a compilar o funcionar en una más reciente.
- Por supuesto, la API externa no debe cambiar (llamadas del sistema, `/proc`, `/sys`), dado que eso podría romper los programas existentes. Pueden añadirse nuevas funcionalidades, pero los desarrolladores del kernel intentan mantener compatibilidad hacia atrás con las versiones anteriores, al menos durante uno o varios años.

Sin API interna de Linux estable 2/3

- Siempre que un desarrollador cambia una API interna, tiene que actualizar todo el código que la usa, para que todo siga funcionando.
- Funciona bien para el código en el árbol *mainline* del kernel.
- Díficil de mantener en drivers fuera del árbol o de código cerrado.

Sin API interna de Linux estable 3/3

- Ejemplo USB
  - Linux ha actualizado su API interna del USB al menos tres veces (correcciones, problemas de seguridad, soporte para dispositivos de alta velocidad) y ahora tiene las mejores velocidades del bus USB (comparado con otros sistemas)
  - Windows XP también tuvo que reescribir su código USB tres veces. Sin embargo, debido a que no se pueden actualizar los binarios de los drivers de código cerrado, tuvieron que mantener compatibilidad hacia atrás con todas las implementaciones anteriores. Esto es muy costoso (desarrollo, seguridad, estabilidad, rendimiento).

Restricciones de memoria del kernel

- Sin protección de memoria
- Acceder a localizaciones ilegales de memoria resulta en *kernel oopses* (en ocasiones, fatales).
- Tamaño de pila fijo (8 o 4 KB). A diferencia del espacio de usuario, no hay forma de hacerla crecer.
- La memoria del kernel no puede "intercambiada" (*swapped out*) por las mismas razones.

Restricciones de la licencia del kernel de Linux

- El kernel de Linux está licenciado bajo la GNU General Public License version 2
  - Esta licencia da derecho a usar, estudiar, modificar y compartr el software libremente
- Sin embargo, cuando el software se redistribuye, ya sea modificado o sin modificar, la GPL obliga a que el software sea redistribuido bajo la misma licencia, con el código fuente
  - Si se hacen modificaciones en el kernel de Linux (por ejemplo, para adaptarlo al hardware propio), es una obra derivada del kernel, y por consiguiente debe ser liberada bajo GPLv2
  - La validez de la GPL en este punto ya ha sido verificada en los tribunales
- Sin embargo, únicamente se está obligado a hacerlo:
  - En el momento en que el dispositivo empieza a ser distribuido
  - A los clientes, no al mundo entero

Código propietario y el kernel

- Es ilegal distribuir un binario del kernel que incluya drivers estáticamente compilados
- Los módulos del kernel son un área nebulosa: ¿son obras derivadas del kernel o no?
  - La opinión general de la comunidad del kernel es que los drivers propietarios son malos
  - Desde un punto de vista legal, cada driver es probablemente un caso distinto
  - ¿Realmente es útil mantener los drivers en secreto?
- Hay algunos ejemplos de drivers propietarios, como los drivers gráficos de Nvidia
  - Usan una envoltura entre el driver y el kernel
  - No está claro si esto los convierte en legales

Ventajas de los drivers GPL 1/2

- No es necesario escribir los drivers desde cero. Se puede reutilizar código de drivers de software libre similares.
- Se consiguen contribuciones, soporte, revisión del código y pruebas de la comunidad gratutitamente. Los drivers propietarios no consiguen nada de eso.
- Los drivers pueden ser lanzados libremente por otros (principalmente, distribuciones).
- Los drivers de código cerrado a menudo soportan una versión dada del kernel. Un sistema con drivers de código cerrado de dos fuentes diferentes es inmanejable.

Ventajas de los drivers GPL 2/2

- Los usuarios y la comunidad obtienen una imagen positiva de la empresa. Hace más fácil contratar desarrolladores talentosos
- No es necesario proporcionar lanzamientos con los binarios del driver para cada versión del kernel y cada parche (drivers de código cerrado).
- Los drivers tienen todos los privilegios. Se necesitan las fuentes para asegurarse de que un driver no es un riesgo para la seguridad.
- Los drivers pueden compilarse estáticamente dentro del kernel (es útil tener una imagen del kernel con todos los drivers que se necesitan en el arranque).

Ventajas de los drivers dentro del árbol

- Una vez las fuentes son aceptadas en el árbol *mainline*, son mantenidos por la gente que hace cambios.
- Mantenimiento sin coste, reparaciones de seguridad y mejoras.
- Acceso fácil a las fuentes por parte de los usuarios.
- Mucha más gente revisando el código.

Drivers de dispositivos en el espacio de usuario 1/2

- Es posible implementar los drivers de dispositivos en el espacio de usuario.
- Esos drivers necesitan acceder solamente a los dispositivos a través de drivers genéricos mínimos del kernel.
- Ejemplos
  - Drivers de impresora y escáner (sobre los drivers del puerto paralelo o USB)
  - Drivers X: drivers del kernel a bajo nivel + drivers X en el espacio de usuario.

Drivers de dispositivos en el espacio de usuario 2/2

- Ventajas
  - No son necesarias habilidades de programación en el kernel. Es más fácil reusar código entre dispositivos.
  - Los drivers se pueden escribir en cualquier lenguaje.
  - Los drivers pueden mantenerse propietarios.
  - El código del driver puede matarse y depurarse. No se puede "romper" el kernel.
  - Pueden ser "intercambiados" (*swapped out*) (el código del kernel no puede).
  - Pueden usar computación en coma flotante.
  - Menos complejidad dentro del kernel.
- Inconvenientes
  - Manejo menos directo de interrupciones.
  - Latencia mayor que en el código del kernel.

Fuentes de Linux

Estructura de las fuentes de Linux 1/5

- `arch/<ARCH>`
  - Código específico de arquitectura
  - `arch/<ARCH>/mach-<machine>`, código específico de máquina/placa
  - `arch/<ARCH>/include/asm`, headers específicos de la arquitectura
  - `arch/<ARCH>/boot/dts`, ficheros fuente del árbol de dispositivos *Device Tree*, para algunas arquitecturas
- `block/`
  - Núcleo de la capa de bloques
- `COPYING`
  - Condiciones de copia de Linux (GNU GPL)
- `CREDITS`
  - Principales contribuidores de Linux
- `crypto/`
  - Librerías criptográficas

Estructura de las fuentes de Linux 2/5

- `Documentation/`
  - Documentación del kernel
- `drivers/`
  - Todos los drivers de dispositivos excepto los de sonido
- `firmware/`
  - Legado: imágenes de firmware extraídas de drivers antiguos
- `fs/`
  - Sistemas de ficheros (`fs/ext3/`, etc.)
- `include/`
  - Headers del kernel
- `include/linux/`
  - Headers del núcleo del kernel de Linux
- `include/uapi/`
  - Headers de la API del espacio de usuario
- `init/`
  - Inicialización de Linux (incluyendo `main.c`)
- `ipc/`
  - Código usado para la comunicación entre procesos

Estructura de las fuentes de Linux 3/5

- `Kbuild`
  - Parte del sistema de compilación del kernel
- `Kconfig`
  - Fichero de descripción de nivel superior para los parámetros de configuración
- `kernel/`
  - Núcleo del kernel de Linux (extremadamente pequeño)
- `lib/`
  - Rutinas de librerias (zlib, crc32...)
- `MAINTAINERS`
  - Mantenedores de cada parte del kernel
- `Makefile`
  - Makefile de nivel superior de Linux (fija la arquitectura y la versión)
- `mm/`
  - Código de gestión de memoria (también pequeño)

Estructura de las fuentes de Linux 4/5

- `net/`
  - Código de soporte de red (no drivers)
- `README`
  - Vista general e instrucciones de compilación
- `REPORTING-BUGS`
  - Instrucciones para reportar bugs
- `samples/`
  - Código de ejemplo (markers, kprobes, kobjects...)
- `scripts/`
  - Scripts para uso interno o externo
- `security/`
  - Implementaciones del modelo de seguridad (SELinux...)
- `sound/`
  - Código de soporte y drivers de sonido
- `tools/`
  - Código para varias herramientas del espacio de usuario (principalmente en C)

Estructura de las fuentes de Linux 5/5

- `usr/`
  - Código para generar un fichero initramfs cpio
- `virt/`
  - Soporte de virtualización (KVM)

Configuración del Kernel

Sistema de configuración y compilación del kernel

- La configuración y compilación del kernel están basadas en múltiples Makefiles
- Se puede interactuar directamente con el `Makefile` principal, presente en el **directorio raíz** del árbol fuente del kernel
- La interacción tiene lugar
  - usando la herramienta `make`, que interpreta el Makefile
  - a través de varios **objetivos**, que definen qué acción debe realizarse (configuración, compilación, instalación, etc.). Ejecutar `make help` para ver todos los objetivos disponibles.
- Ejemplo
  - `cd odroid-3.8.y/`
  - `make <objetivo>`

Configuración del kernel (1)

- El kernel contiene miles de drivers de dispositivos, drivers de sistemas de ficheros, protocolos de red y otros elementos configurables
- Están disponibles miles de opciones, que se usan selectivamente para compilar partes del código fuente del kernel
- La configuración del kernel es el proceso de definir el conjunto de opciones con las cuales se quiere compilar el kernel
- El conjunto de opciones depende
  - Del hardware (para los drivers de dispositivos, etc.)
  - De las capacidades que se quieren dar al kernel (capacidades de red, sistemas de ficheros, tiempo real, etc.)

Configuración del kernel (2)

- La configuración se almacena en el fichero `.config` en la raíz de las fuentes del kernel
  - Fichero de texto simple, estilo `clave=valor`
- Como las opciones tienen dependencias, no suele editarse nunca a mano, sino a través de interfaces gráficas o de texto:
  - `make xconfig`, `make gconfig` (gráficas)
  - `make menuconfig`, `make nconfig` (texto)
  - Se puede cambiar de una a otra, todas cargan/guardan el mismo fichero `.config`, y muestran el mismo conjunto de opciones
- Para modificar el kernel en una distribución GNU/Linux: los ficheros de configuración se suelen lanzar en `/boot/`, junto con las imágenes del kernel: `/boot/config-3.8.13.23`

¿Kernel o módulo?

- La **imagen del kernel** es un **único archivo**, resultante de linkar todos los archivos objeto que corresponden a características seleccionadas en la configuración.
  - Este es el fichero que carga en memoria el bootloader
  - Todas las características incluídas están por tanto disponibles tan pronto como arranca el kernel, cuando no existe sistema de ficheros
- Algunas características (drivers de dispositivos, sistemas de ficheros, etc.) se pueden compilar, no obstante, como **módulos**
  - Estos son *plugins* que pueden cargarse/descargarse dinámicamente para añadir/quitar características al kernel
  - Cada **módulo se almacena como un archivo separado en el sistema de ficheros**, y por tanto es obligatorio el acceso al sistema de ficheros para usar los módulos
  - Esto no es posible en el procedimiento temprano de arranque del kernel, porque no hay sistema de ficheros disponible

Tipos de opciones del kernel

- Hay diferentes tipos de opciones
  - Opciones `bool`, pueden ser:
    - *true* (para incluir la característica en el kernel) o
    - *false* (para excluir la característica del kernel)
  - Opciones `tristate`, pueden ser:
    - *true* (para incluir la característica en la imagen del kernel) o
    - *module* (para incluir la característica como un módulo del kernel) o
    - *false* (para excluir la característica)
  - Opciones `int`, para especificar valores enteros
  - Opciones `hex`, para especificar valores hexadecimales
  - Opciones `string`, para especificar valores string

Dependencias de las opciones del kernel

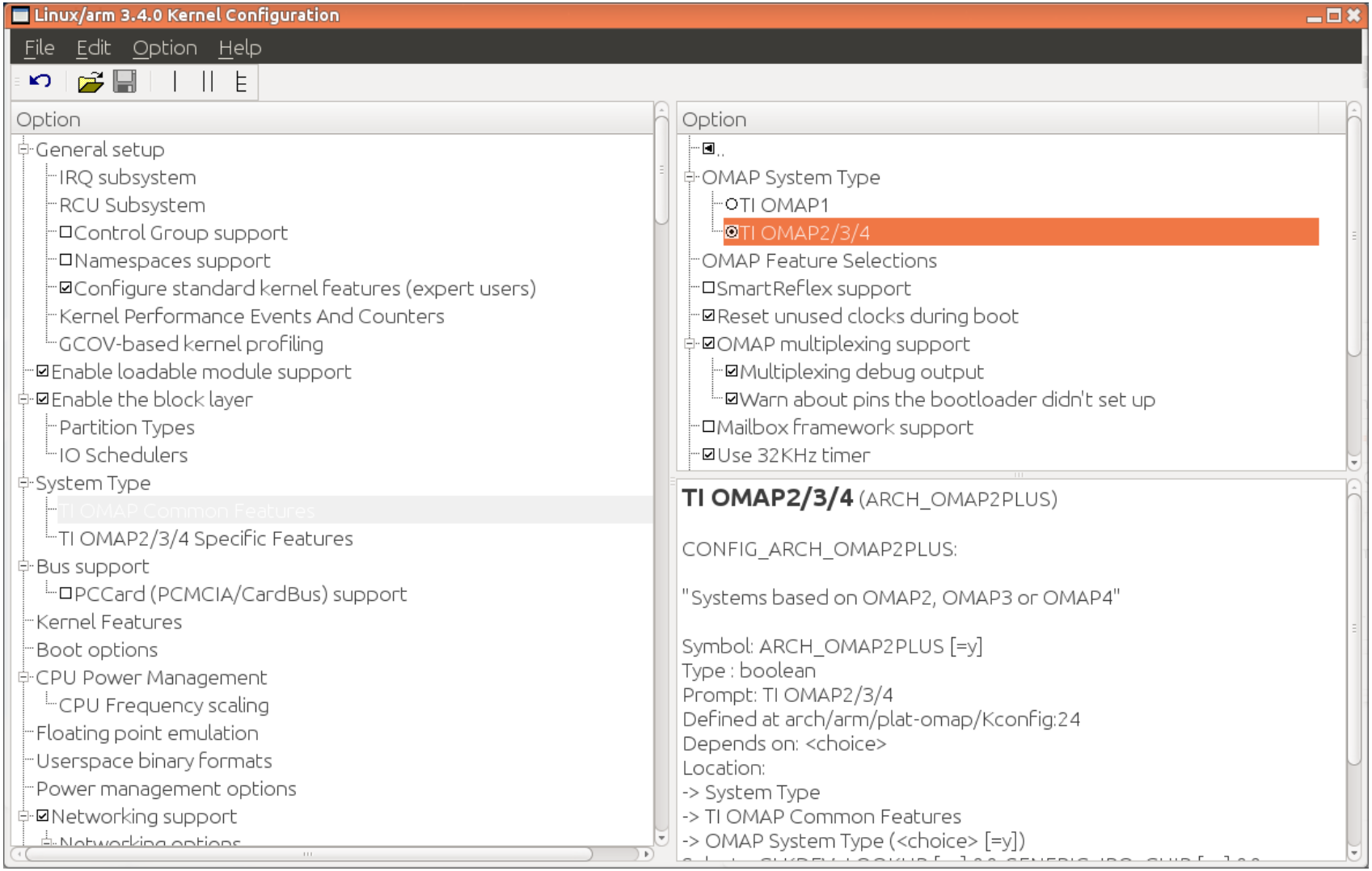
- Hay dependencias entre las opciones del kernel
- Por ejemplo, habilitar un driver de red requiere que esté habilitada la parte de redes
- Dos tipos de dependencias
  - Dependencias `depends on`. En este caso, la opcion A que depende de la opción B no es visible hasta que se habilita la opción B.
  - Dependencias `select`. En este caso, si la opción A depende de la B, cuando se habilita la opción A, la opción B es automáticamente habilitada.
  - `make xconfig` permite ver todas las opciones, incluso aquellas que no pueden seleccionarse debido a dependencias no cumplidas. En ese caso, se muestran en gris

make xconfig

`make xconfig`

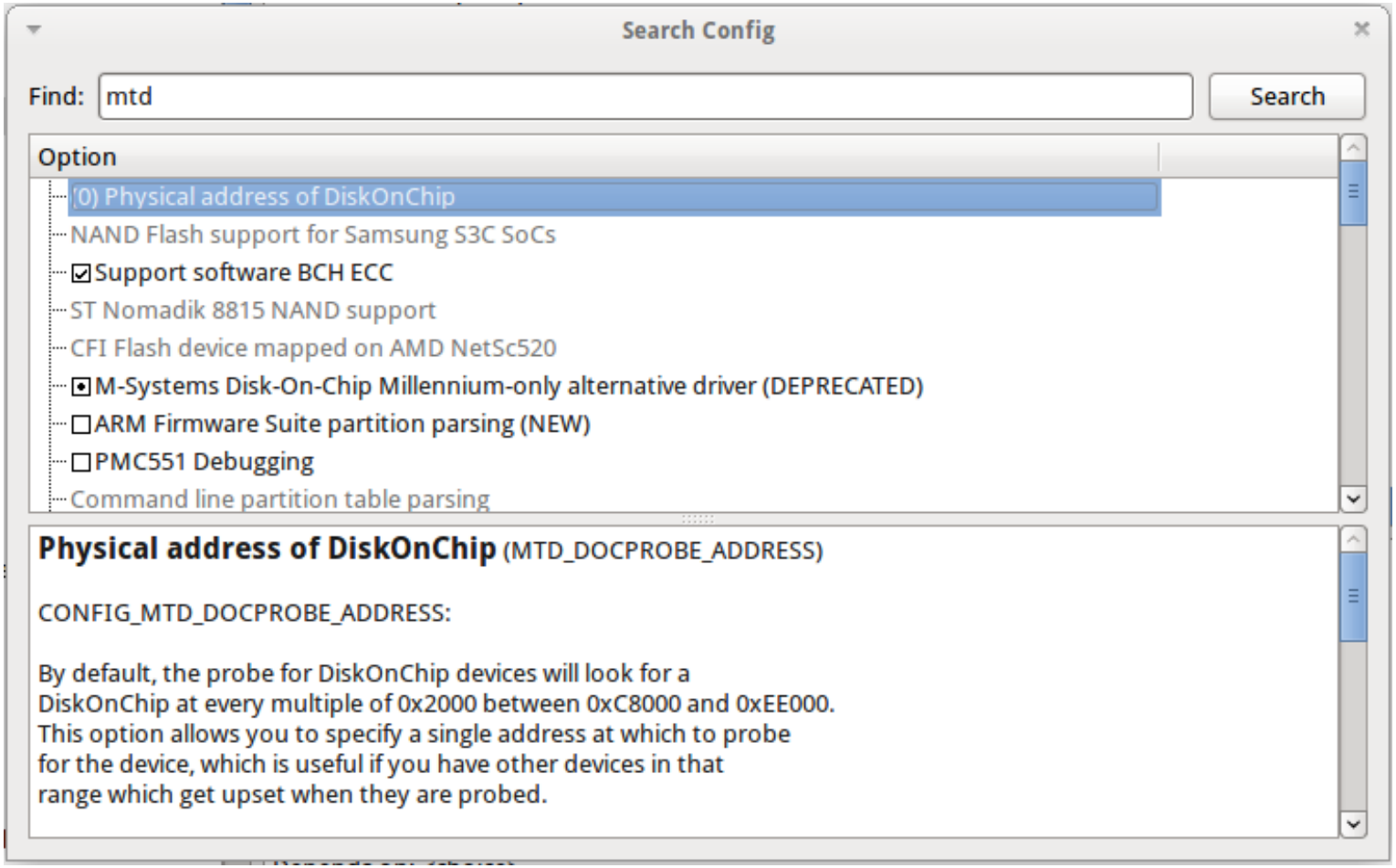
- La interfaz gráfica más común para configurar el kernel.
- Explorador de ficheros: más fácil cargar archivos de configuración
- Interfaz de búsqueda para buscar parámetros
- Paquetes de Ubuntu/Debian requeridos: `libqt4-dev g++` (`libqt3-mt-dev` para versiones del kernel anteriores)

Captura de pantalla make xconfig



Interfaz de búsqueda make xconfig

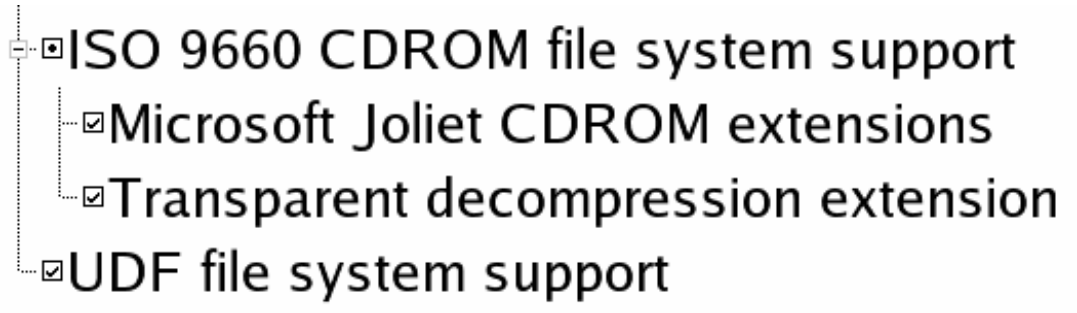
Busca una palabra clave en el nombre del parámetro. Permite seleccionar o deseleccionar parámetros encontrados.



Opciones de configuración del kernel

Compilado como módulo (archivo separado)
`CONFIG_ISO9660_FS=m`

Opciones de driver
`CONFIG_JOLIET=y`
`CONFIG_ZISOFS=y`



Compilado estáticamente dentro del kernel
`CONFIG_UDF_FS=y`

Extracto del archivo .config correspondiente

Las opciones se agrupan por secciones y comienzan con CONFIG_.

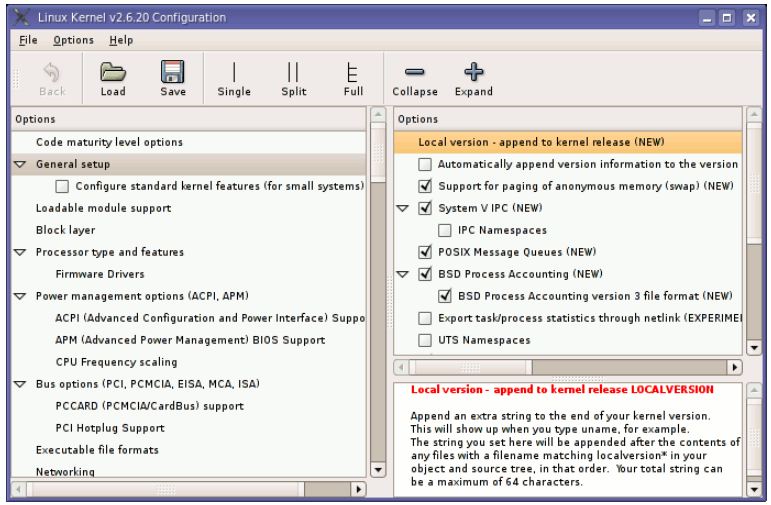
```
#
# CD-ROM/DVD Filesystems
#
CONFIG_ISO9660_FS=m
CONFIG_JOLIET=y
CONFIG_ZISOFS=y
CONFIG_UDF_FS=y
CONFIG_UDF_NLS=y

#
# DOS/FAT/NT Filesystems
#
# CONFIG_MSDOS_FS is not set
# CONFIG_VFAT_FS is not set
CONFIG_NTFS_FS=m
# CONFIG_NTFS_DEBUG is not set
CONFIG_NTFS_RW=y
```

make gconfig

`make gconfig`

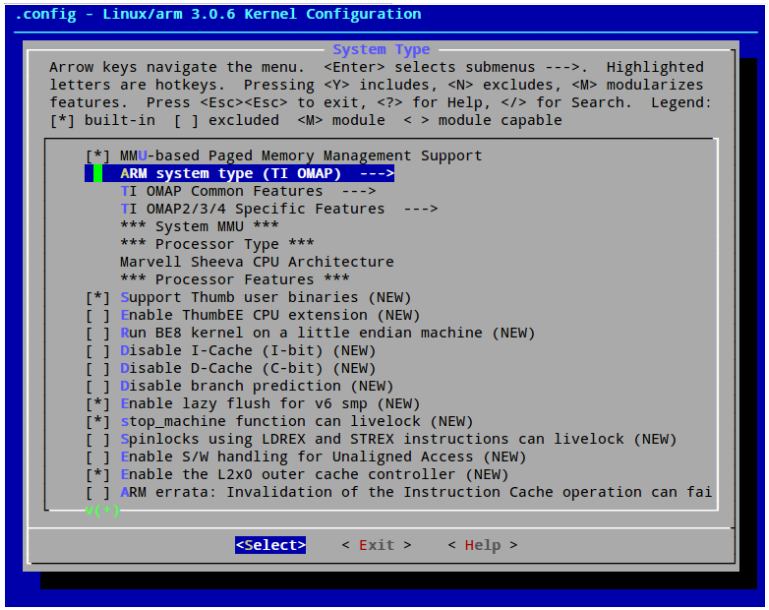
- Interfaz gráfica de configuración basada en *GTK*. Funcionalidad similar a la de make `xconfig`.
- Carece de la funcionalidad de búsqueda.
- Paquetes Debian requeridos: `libglade2-dev`



make menuconfig

`make menuconfig`

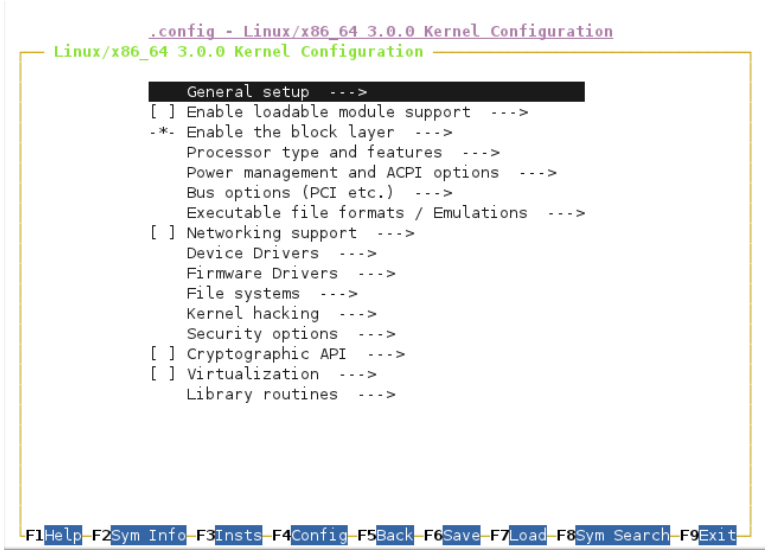
- Útil cuando no hay gráficos disponibles.
- Misma interfaz de otras herramientas: BusyBox, Buildroot...
- Paquetes Debian requeridos: `libncurses-dev`



make nconfig

`make nconfig`

- Una interfaz de texto similar pero más nueva.
- Más amigable (p.ej., más fácil acceder a la información de ayuda).
- Paquetes Debian requeridos: `libncurses-dev`



make oldconfig

`make oldconfig`

- Frecuentemente necesaria
- Útil para actualizar un archivo `.config` de una versión anterior del kernel
- Dispara alertas para parámetros de configuración que ya no existen en el nuevo kernel
- Pide valores para los nuevos parámetros

Si se edita un archivo `.config` a mano, se recomienda encarecidamente ejecutar `make oldconfig` después.

Deshaciendo cambios de configuración

Un problema frecuente:

- Después de cambiar algunos ajustes de configuración del kernel, este deja de funcionar.
- Si no se recuerdan todos los cambios que se hicieron, es posible recuperar la configuración previa:
  `$cp .config.old .config`
- Todas las interfaces de configuración del kernel (`xconfig`, `menuconfig`, `oldconfig`...) mantienen esta copia de seguridad `.config.old`.

Configuración por arquitectura

- El conjunto de opciones de configuración es dependiente de la arquitectura
  - Algunas opciones de configuración son muy específicas de la arquitectura
  - La mayoría de opciones de configuración (opciones globales del kernel, subsistema de redes, sistemas de ficheros, la mayoría de drivers de dispositivos) son visibles en todas las arquitecturas.
- Por defecto, el sistema de compilación del kernel asume que el kernel está siendo compilado para la arquitectura huésped; es decir, compilación nativa
- La arquitectura no se define dentro de la configuración, sino en un nivel superior
- Veremos posteriormente como modificar este comportamiento, para permitir configurar kernels para una arquitectura diferente

Repaso de opciones del kernel (en inglés) (1)

- General setup
  - *Local version - append to kernel release* allows to concatenate an arbitrary string to the kernel version that a user can get using `uname -r`. Very useful for support!
  - *Support for swap*, can usually be disabled on most embedded devices
  - *Configure standard kernel features (expert users)* allows to remove features from the kernel to reduce its size. Powerful, but use with care!

Repaso de opciones del kernel (en inglés) (2)

- Loadable module support
  - Allows to enable or completely disable module support. If your system doesn't need kernel modules, best to disable since it saves a significant amount of space and memory
- Enable the block layer
  - If `CONFIG_EXPERT` is enabled, the block layer can be completely removed. Embedded systems using only raw flash storage (MTD) can safely disable the block layer
- Processor type and features (x86) or System type (ARM) or CPU selection (MIPS)
  - Allows to select the CPU or machine for which the kernel must be compiled
  - On x86, only optimization-related, on other architectures very important since there's no compatibility

Repaso de opciones del kernel (en inglés) (3)

- Kernel features
  - Tickless system, which allows to disable the regular timer tick and use on-demand ticks instead. Improves power savings
  - High resolution timer support. By default, the resolution of timer is the tick resolution. With high resolution timers, the resolution is as precise as the hardware can give
  - Preemptible kernel enables the preemption inside the kernel code (the user space code is always preemptible). See our real-time presentation for details
- Power management
  - Global power management option needed for all power management related features
  - Suspend to RAM, CPU frequency scaling, CPU idle control, suspend to disk

Repaso de opciones del kernel (en inglés) (4)

- Networking support
  - The network stack
  - Networking options
    - Unix sockets, needed for a form of inter-process communication
    - TCP/IP protocol with options for multicast, routing, tunneling, Ipsec, Ipv6, congestion algorithms, etc.
    - Other protocols such as DCCP, SCTP, TIPC, ATM
    - Ethernet bridging, QoS, etc.
  - Support for other types of network
    - CAN bus, Infrared, Bluetooth, Wireless stack, WiMax stack, etc.

Repaso de opciones del kernel (en inglés) (5)

- Device drivers
  - MTD is the subsystem for flash (NOR, NAND, OneNand, battery-backed memory, etc.)
  - Parallel port support
  - Block devices, a few misc block drivers such as loopback, NBD, etc.
  - ATA/ATAPI, support for IDE disk, CD-ROM and tapes. A new stack exists
  - SCSI
    - The SCSI core, needed not only for SCSI devices but also for USB mass storage devices, SATA and PATA hard drives, etc.
    - SCSI controller drivers

Repaso de opciones del kernel (en inglés) (6)

- Device drivers (cont)
  - SATA and PATA, the new stack for hard disks, relies on SCSI
  - RAID and LVM, to aggregate hard drives and do replication
  - Network device support, with the network controller drivers. Ethernet, Wireless but also PPP
  - Input device support, for all types of input devices: keyboards, mice, joysticks, touchscreens, tablets, etc.
  - Character devices, contains various device drivers, amongst them
    - serial port controller drivers
    - PTY driver, needed for things like SSH or telnet
  - I2C, SPI, 1-wire, support for the popular embedded buses
  - Hardware monitoring support, infrastructure and drivers for thermal sensors

Repaso de opciones del kernel (en inglés) (7)

- Device drivers (cont)
  - Watchdog support
  - Multifunction drivers are drivers that do not fit in any other category because the device offers multiple functionality at the same time
  - Multimedia support, contains the V4L and DVB subsystems, for video capture, webcams, AM/FM cards, DVB adapters
  - Graphics support, infrastructure and drivers for framebuffers
  - Sound card support, the OSS and ALSA sound infrastructures and the corresponding drivers
  - HID devices, support for the devices that conform to the HID specification (Human Input Devices)

Repaso de opciones del kernel (en inglés) (8)

- Device drivers (cont)
  - USB support
    - Infrastructure
    - Host controller drivers
    - Device drivers, for devices connected to the embedded system
    - Gadget controller drivers
    - Gadget drivers, to let the embedded system act as a mass-storage device, a serial port or an Ethernet adapter
  - MMC/SD/SDIO support
  - LED support
  - Real Time Clock drivers
  - Voltage and current regulators
  - Staging drivers, crappy drivers being cleaned up

Repaso de opciones del kernel (en inglés) (9)

- For some categories of devices the driver is not implemented inside the kernel
  - Printers
  - Scanners
  - Graphics drivers used by X.org
  - Some USB devices
- For these devices, the kernel only provides a mechanism to access the hardware, the driver is implemented in user space

Repaso de opciones del kernel (en inglés) (10)

- File systems
  - The common Linux filesystems for block devices: ext2, ext3, ext4
  - Less common filesystems: XFS, JFS, ReiserFS, GFS2, OCFS2, Btrfs
  - CD-ROM filesystems: ISO9660, UDF
  - DOS/Windows filesystems: FAT and NTFS
  - Pseudo filesystems: proc and sysfs
  - Miscellaneous filesystems, with amongst other flash filesystems such as JFFS2, UBIFS, SquashFS, cramfs
  - Network filesystems, with mainly NFS and SMB/CIFS
- Kernel hacking
  - Debugging features useful for kernel developers

Compilar e instalar el kernel para el sistema anfitrión

Compilación del kernel

- `make`
  - en el directorio raíz de las fuentes del kernel
  - Recordar ejecutar múltiples trabajos en paralelo si se tienen varios núcleos en la CPU. Ejemplo: `make -j 8`
  - No es necesario ejecutarlo como root
- Genera
  - `vmlinux`, la imagen del kernel en bruto, sin comprimir, en formato ELF, útil para depuración, pero no puede cargarse
  - `arch/<arch>/boot/*Image`, la imagen del kernel final, normalmente comprimida, que puede arrancarse
    - `bzImage` para x86, `zImage` para ARM, `vmImage.gz` para Blackfin, etc.
  - `arch/<arch>/boot/dts/*.dtb`, archivos compilados del árbol de dispositivos (en algunas arquitecturas)
  - Todos los módulos del kernel, repartidos por el árbol fuente del kernel, como archivos `.ko`.

Instalación del kernel

- `make install`
  - Hace la instalación para el sistema anfitrión por defecto, por lo que necesita ejecutarse como root. No suele utilizarse cuando se compila para un sistema embebido.
- Instala
  - `/boot/vmlinuz-<version>`
    La imagen comprimida del kernel. La misma que la de `arch/<arch>/boot`
  - `/boot/System.map-<version>`
    Almacena las direcciones de los símbolos del kernel
  - `/boot/config-<version>`
    Configuración del kernel para esta versión
- Típicamente, vuelve a ejecutar la utilidad de configuración del bootloader para tener en cuenta el nuevo kernel.

Instalación de módulos

- `make modules_install`
  - Hace la instalación en el sistema anfitrión por defecto, por lo que necesita ejecutarse como root
- Instala todos los módulos en `/lib/modules/<version>/`
  - `kernel/`
    Archivos módulo `.ko` (Kernel Object), en la misma estructura de directorios que en las fuentes.
  - `modules.alias`
    Alias de los módulos para las utilidades de carga de módulos.
    Línea de ejemplo:
    `alias sound-service-?-0 snd_mixer_oss`
  - `modules.dep`
    Dependencias de los módulos
  - `modules.symbols`
    Dice a qué módulo pertenece un símbolo dado.

Objetivos de limpieza del kernel

- Limpiar archivos generados (para forzar la recompilación):
  `make clean`
- Borrar todos los archivos generados. Necesario cuando se cambia de una arquitectura a otra. Cuidado: también borra el fichero `.config`:
  `make mrproper`
- Borra también la copia de seguridad del editor y los archivos de rechazo de parches (principalmente para generar parches):
  `make distclean`

Archivos de configuración predefinidos

- Archivos de configuración por defecto disponibles, por placa o por familia de CPU
  - Se almacenan en `arch/<arch>/configs/`, y son simplemente archivos `.config` mínimos
  - Esta es la forma más común de configurar un kernel para plataformas embebidas
- Ejecutar `make help` para buscar si hay alguno disponible para la plataforma
- Para cargar un archivo de configuración por defecto, basta ejecutar
  `make acme_defconfig`
  - Esto sobrescribirá el archivo `.config` existente
- Para crear un archivo de configuración por defecto propio
  - `make savedefconfig`, para crear un archivo de configuración mínimo
  - `mv defconfig arch/<arch>/configs/myown_defconfig`

Configurando el kernel

- Después de cargar un archivo de configuración por defecto, se puede ajustar la configuración de acuerdo a las necesidades con las interfaces normales `xconfig`, `gconfig` o `menuconfig`
- También se puede empezar la configuración desde cero sin cargar un fichero de configuración por defecto
- Como la arquitectura es diferente de la arquitectura del anfitrión
  - Algunas opciones serán diferentes de la configuración nativa (opciones específicas del procesador y la arquitectura, drivers específicos, etc.)
  - Muchas opciones serán idénticas (sistemas de ficheros, protocolos de red, drivers independientes de arquitectura, etc.)
- Hay que asegurarse de tener soporte para la CPU correcta, la placa correcta y los drivers de dispositivo correctos.

Árbol de dispositivos - Device Tree

- Muchas arquitecturas embebidas tienen una gran cantidad de hardware no detectable.
- Dependiendo de la arquitectura, este hardware se describe usando código C directamente dentro del kernel, o usando un lenguaje especial de descripción del hardware en un *Árbol de dispositivos* o *Device Tree*.
- ARM, PowerPC, OpenRISC, ARC, Microblaze son ejemplos de arquitecturas que usan el *Device Tree*.
- Una fuente *Device Tree*, escrita por los desarrolladores del kernel, se compila en un binario *Device Tree Blob*, que se pasa al kernel en el arranque.
  - Hay un Device Tree diferente para cada placa/plataforma soportada por el kernel, disponible en `arch/arm/boot/dts/<board>.dtb`.
- El bootloader debe cargar tanto la imagen del kernel como el *Device Tree Blob* en memoria antes de iniciar el kernel.

Compilar e instalar el kernel

- Ejecutar `make`
- Copiar la imagen final del kernel al almacenamiento del objetivo
  - puede ser `uImage`, `zImage`, `vmlinux`, `bzImage` en `arch/<arch>/boot`
  - copiar el Device Tree Blob también podría ser necesario, están disponibles en `arch/<arch>/boot/dts`
- `make install` raramente se usa en el desarrollo embebido, puesto que la imagen del kernel es un único archivo, fácil de manejar
  - Sin embargo, es posible personalizar el comportamiento de make install en `arch/<arch>/boot/install.sh`
- `make modules_install` se usa aun así en desarrollo embebido, puesto que instala muchos módulos y archivos de descripción
  - `make INSTALL_MOD_PATH=<dir>/ modules_install`
  - La variable `INSTALL_MOD_PATH` es necesaria para instalar los módulos en el sistema de ficheros raíz del objetivo en lugar de en el sistema de ficheros raíz del anfitrión.

Arrancar con U-Boot

- Las versiones recientes de U-Boot pueden cargar el binario `zImage`.
- Las versiones antiguas requieren un formato especial de imagen del kernel: `uImage`
  - `uImage` se genera a partir de `zImage` usando la herramienta `mkimage`. Lo hace automáticamente el objetivo `make uImage` del kernel.
  - En algunas plataformas ARM, `make uImage` requiere pasar una variable de entorno `LOADADDR`, que indica en qué dirección de memoria física se ejecutará el kernel.
- Además de la imagen del kernel, U-Boot puede también pasar un *Device Tree Blob* al kernel.
- El proceso típico de arranque es, por tanto:
  1. Cargar `zImage` o `uImage` en la dirección X de memoria
  2. Cargar `<board>.dtb` en la dirección Y en memoria
  3. Empezar el kernel con `bootz X - Y` o `bootm X - Y`
     La `-` del medio indica "sin *initramfs*"

Línea de comandos del kernel

- Además de la configuración en tiempo de compilación, el comportamiento del kernel puede ajustarse sin recompilación usando la **línea de comandos del kernel**
- La línea de comandos del kernel es un string que define varios argumentos al kernel
  - Es muy importante para la configuración del sistema
  - `root=` para el sistema de ficheros raíz
  - `console=` para el destino de los mensajes del kernel
  - Existen muchos más. Los más importantes están documentados enen las fuentes del kernel.
- La línea de comandos del kernel, puede
  - Ser pasada por el bootloader. En U-Boot, los contenidos de la variable de entorno `bootargs` se pasan automaticamente al kernel
  - Compilarse dentro del kernel, usando la opción `CONFIG_CMDLINE`.

Compilación cruzada del kernel

Compilación cruzada del kernel

Cuando se compila un kernel de Linux para otra arquitectura CPU

- Mucho más rápido que compilar nativamente, cuando el sistema embebido es mucho más lento que la estación de trabajo GNU/Linux.
- Mucho más fácil puesto que las herramientas de desarrollo para la estación de trabajo GNU/Linux son mucho más fáciles que encontrar.
- Para diferenciarse del compilador nativo, los ejecutables para la compilación cruzada están prefijados por el nombre del sistema objetivo, la arquitectura y, a veces, la librería.
  Ejemplos:
  `mips-linux-gcc`, el prefijo es `mips-linux-`
  `arm-linux-gnueabi-gcc`, el prefijo es `arm-linux-gnueabi-`

Especificando la compilación cruzada (1)

La arquitectura CPU y el prefijo del compilador cruzado se definen a través de las variables `ARCH` y `CROSS_COMPILE` en el Makefile de nivel superior.

- `ARCH` es el nombre de la arquitectura. Viene definido por el nombre del subdirectorio en `arch/` en las fuentes del kernel
  - Ejemplo: `arm` si se quiere compilar un kernel para la arquitectura `arm`.
- `CROSS_COMPILE` es el prefijo de las herramientas de compilación cruzada.
  - Ejemplo: `arm-linux-` si el compilador es `arm-linux-gcc`

Especificando la compilación cruzada (2)

Dos soluciones para definir `ARCH` y `CROSS_COMPILE`:

- Pasar `ARCH` y `CROSS_COMPILE` en la línea del `make`:
  `make ARCH=arm CROSS_COMPILE=arm-linux- ...`
  Inconveniente: Es fácil olvidar pasar estas variables cuando se ejecuta cualquier comando `make`, estropeando la configuración y compilación.
- Definir `ARCH` y `CROSS_COMPILE` como variables de entorno:
  ```
  export ARCH=arm
  export CROSS_COMPILE=arm-linux-
  ```
  Inconveniente: solo funciona dentro del terminal actual. Es posible poner estos ajustes en un fichero al que se llame cada vez que se empieza a trabajar en el projecto. Si solo se trabaja en una única arquitectura, siempre con el mismo *toolchain*, incluso se pueden colocar estos ajustes en el archivo `~/.bashrc` para hacerlos permanentes y visibles desde cualquier terminal.

Laboratorio - Compilación y arranque del kernel

- Compilar el kernel
- Volver a arrancar el ODROID

Usar módulos del kernel

Ventajas de los módulos

- Los módulos hacen más fácil el desarrollo de drivers sin reiniciar: cargar, probar, descargar, recompilar, cargar...
- Útil para mantener el tamaño de la imagen del kernel en el mínimo (esencial en distribuciones GNU/Linux para PC).
- También es útil para reducir el tiempo de arranque: no se necesita invertir tiempo inicializando dispositivos y características del kernel que solo harán falta más tarde.
- Cuidado: una vez cargados, tienen control y privilegios absolutos en el sistema. No hay ninguna protección particular. Por ello, solo el usuario `root` puede cargar y descargar módulos.

Dependencias de los módulos

- Algunos módulos del kernel pueden depender de otros módulos, que necesitan ser cargados primero.
- Ejemplo: el módulo `usb-storage` depende de los módulos `scsi_mod`, `libusual` y `usbcore`.
- Las dependencias se describen en `/lib/modules/<kernel-version>/modules.dep`
  Este archivo se genera cuando se ejecuta `make modules_install`.

Log del Kernel

Cuando se carga un nuevo módulo, la información relativa a él está dispoible en el log del kernel.

- El kernel mantiene sus mensajes en un buffer circular (de forma que no consume más memoria con muchos mensajes)
- Los mensajes del log del kernel están accesibles mediante el comando `dmesg` (**d**iagnostic **mes**sa**g**e)
- Los mensajes del log del kernel también se muestran en la consola del sistema (los mensajes de la consola pueden filtrarse por nivel usando el parámetro del kernel `loglevel`, o desactivados completamente con el parámetro `quiet`).
- Nótese que se puede escribir en el log del kernel desde el espacio de usuario también:
  `echo "<n>Debug info" > /dev/kmsg`

Utilidades para los módulos (1)

- `modinfo <module_name>`
  `modinfo <module_path>.ko`
  Obtiene información sobre un módulo: parámetros, licencia, descripción y dependencias.
  Muy útil antes de decidir si cargar un módulo o no.
- `sudo insmod <module_path>.ko`
  Intenta cargar el módulo dado. debe proporcionarse la ruta completa al archivo objeto del módulo.

Entender los problemas en la carga de módulos

- Cuando falla la carga de un módulo, `insmod` no suele dar suficientes detalles
- A veces, los detalles están disponibles en el log del kernel.
- Ejemplo:

```
$ sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1 Device or resource busy
$ dmesg
[17549774.552000] Failed to register handler for irq channel 2
```

Utilidades para los módulos (2)

- `sudo modprobe <module_name>`
  Uso más común de `modprobe`: intenta cargar todos los módulos de los que depende el módulo dado, y entonces, el propio módulo. Hay muchas opciones disponibles. `modprobe` automáticamente busca en `/lib/modules/<version>/` el archivo objeto que corresponde al nombre de módulo dado.
- `lsmod`
  Muestra la lista de módulos cargados
  Comparar su salida con los contenidos de `/proc/modules`

Utilidades para los módulos (3)

- `sudo rmmod <module_name>`
  Intenta eliminar el módulo dado.
  Solo se permitirá si el módulo ya no está en uso (por ejemplo, no hay procesos abriendo un fichero de dispositivo)
- `sudo modprobe -r <module_name>`
  Intenta eliminar el módulo dado y todos sus módulos dependientes (que ya no se necesitan al eliminar el módulo)

Pasar parámetros a los módulos

- Encontrar los parámetros disponibles:
  `modinfo snd-intel8x0m`
- A través de `insmod`:
  `sudo insmod ./snd-intel8x0m.ko index=-2`
- A través de `modprobe`:
  Ajustar parámetros en `/etc/modprobe.conf` o en cualquier fichero en `/etc/modprobe.d/`:
  `options snd-intel8x0m index=-2`
- A través de la línea de comandos del kernel, cuando el driver se compila estáticamente dentro del kernel:
  `snd-intel8x0m.index=-2`
  - `snd-intel8x0m` es el *nombre del driver*
  - `index` es el *nombre del parámetro del driver*
  - `-2` es el *valor del parámetro del driver*

Comprobar los valores de los parámetros del módulo

¿Cómo encontrar los valores actuales para los parámetros de un módulo cargado?

- Comprobar `/sys/module/<name>/parameters`.
- Hay un archivo por parámetro, conteniendo el valor del parámetro.

Desarrollo de módulos del kernel

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Módulo Hola mundo 1/2

```c
/* holamundo.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init holamundo_init(void)
{
  pr_alert("Hola mundo.\n");
  return 0;
}

static void __exit holamundo_exit(void)
{
  pr_alert("Adios mundo.\n");
}

module_init(holamundo_init);
module_exit(holamundo_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Mi primer modulo");
MODULE_AUTHOR("Programador");
```

Módulo Hola mundo 2/2

- `__init`
  - se elimina después de la inicialización (kernel estático o módulo.)
- `__exit`
  - se descarta cuando el módulo se compila estáticamente en el kernel o cuando el soporte para la descarga de módulos no está habilitado.

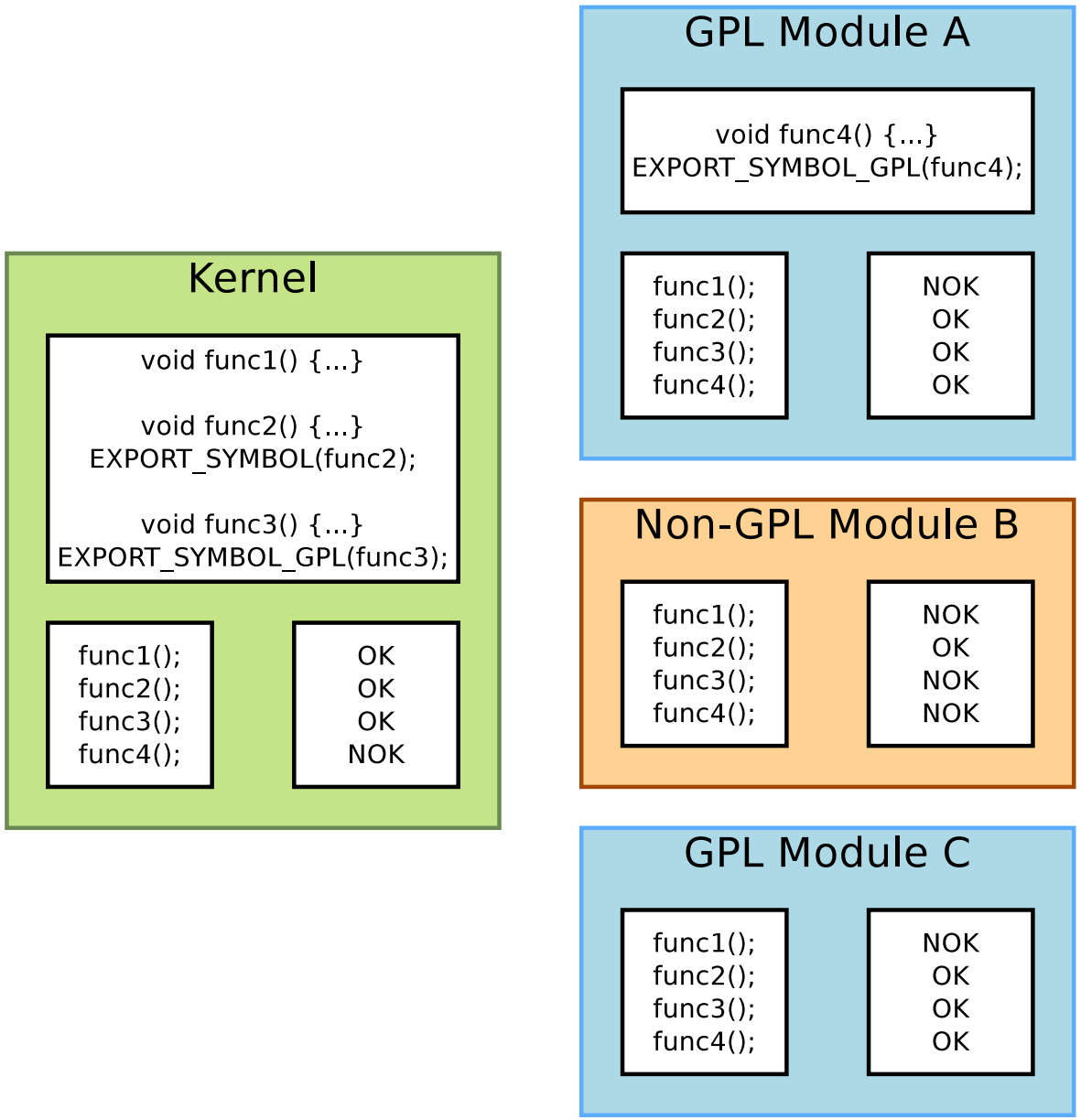
Explicaciones sobre el módulo Hola mundo

- Headers específicos al kernel de Linux: `linux/xxx.h`
  - No hay acceso a la librería de C habitual, estamos programando en el kernel
- Una función de inicialización
  - Se llama cuando el módulo es cargado, devuelve un código de error (`0` en caso de éxito, valor negativo en caso de fallo).
  - Declarada por la macro `module_init()`: el nombre de la función no importa, aun así `<modulename>_init()` es una convención.
- Una función de limpieza
  - Se llama al descargar el módulo.
  - Declarada por la macro `module_exit()`.
- Información de metadatos declarada usando `MODULE_LICENSE()`, `MODULE_DESCRIPTION()` and `MODULE_AUTHOR()`

Símbolos exportados a los módulos 1/2

- Desde un módulo del kernel, solo puede llamarse un número limitado de funciones
- Las funciones y variables tienen que ser explícitamente exportadas por el kernel para ser visibles en el módulo del kernel
- Se usan dos macros en el kernel para exportar funciones y variables:
  - `EXPORT_SYMBOL(symbolname)`, que exporta una función o variable a todos los módulos
  - `EXPORT_SYMBOL_GPL(symbolname)`, que exporta una función o variable solo a los módulos GPL
- Un driver normal no debería necesitar ninguna función no exportada.

Símbolos exportados a los módulos 2/2



Licencia de los módulos

- Algunos usos
  - Usado para restringir las funciones del kernel que el módulo puede usar si no es un módulo licenciado GPL
    - Diferencia entre `EXPORT_SYMBOL()` y `EXPORT_SYMBOL_GPL()`
  - Usado por los desarrolladores del kernel para identificar problemas procedentes de drivers propietarios, sobre los que no pueden hacer nada
  - Útil para que los usuarios comprueben que su sistema es 100% libre (comprobar `/proc/sys/kernel/tainted`)
- Valores
  - Compatible con GPL (véase `include/linux/license.h`: `GPL`, `GPL v2`, `GPL and additional rights`, `Dual MIT/GPL`, `Dual BSD/GPL`, `Dual MPL/GPL`)
  - `Proprietary`

Compilar un módulo

- Dos soluciones
  - *Fuera del árbol*
    - Cuando el código está fuera del árbol fuente del kernel, en un directorio diferente
    - Ventaja: Podría ser más fácil de manejar que las modificaciones sobre el kernel en sí mismo.
    - Inconvenientes: No integrado en el proceso de configuración/compilación, necesita compilarse separadamente, el driver no se puede compilar estáticamente
  - Dentro del árbol del kernel
    - Bien integrado dentro del proceso de configuración/compilación del kernel
    - El driver puede compilarse estáticamente si es necesario

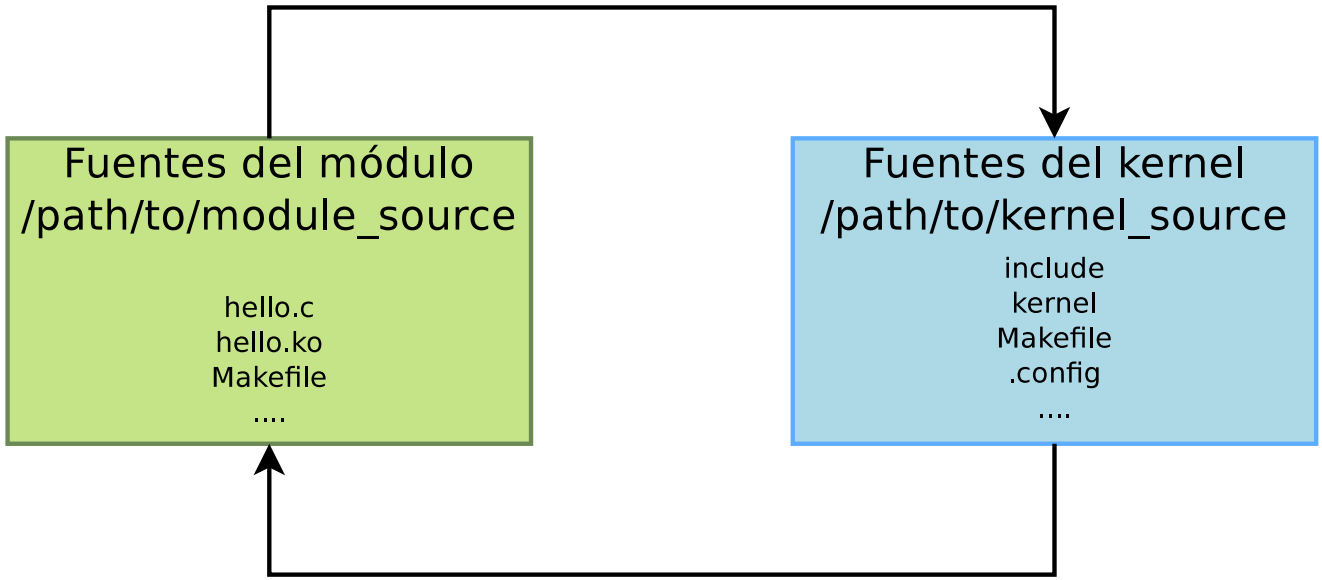
Compilar un módulo fuera del árbol 1/2

- El siguiente `Makefile` debería ser reutilizable para cualquier módulo de Linux fuera del árbol de un solo fichero
- El código fuente es `holamundo.c`
- Simplemente, correr `make` para compilar el archivo `holamundo.ko`

```
ifneq ($(KERNELRELEASE),)
obj-m := holamundo.o
else
KDIR := /path/to/kernel/sources

all:
<tab>$(MAKE) -C $(KDIR) M=$$PWD
endif
```

- Para `KDIR`, se puede elegir:
  - el directorio de las fuentes del kernel completas (configurado + `make modules_prepare`)
  - o el directorio de headers del kernel (`make headers_install`)

Compilar un módulo fuera del árbol 2/2



- El Makefile del módulo se interpreta con `KERNELRELEASE` indefinida, por lo que llama al Makefile del kernel, pasando el directorio del módulo en la variable `M`
- El Makefile del kernel sabe cómo compilar un módulo, y gracias a la variable `M`, sabe donde está el Makefile para nuestro módulo. El Makefile del módulo se interpreta con `KERNELRELEASE` definida, por lo que el kernel ve la definición de `obj-m`.

Módulos y versiones del kernel

- Para ser compilado, un módulo del kernel necesita acceder a los headers del kernel, que contienen las definiciones de funciones, tipos y constantes
- Dos soluciones
  - Fuentes completas del kernel
  - Solo headers del kernel (paquetes `linux-headers-*` en distribuciones Debian/Ubuntu)
- Las fuentes o headers tienen que configurarse
  - Muchas macros o funciones dependen de la configuración
- Un módulo del kernel compilado contra la versión X de los headers del kernel no se cargará en la versión Y del kernel
  - `modprobe` / `insmod` darán error `Invalid module format`

Módulo Hola mundo con parámetros 1/2

```
/* holamundo_param.c */
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");

/* Dos parametros: cuantas veces decir hola y a quien */

static char *whom = "mundo";
module_param(whom, charp, 0);

static int howmany = 1;
module_param(howmany, int, 0);
```

Módulo Hola mundo con parámetros 2/2

```
static int __init holamundo_init(void)
{
  int i;
  for (i = 0; i < howmany; i++)
    pr_alert("(%d) Hola, %s\n", i, whom);
  return 0;
}

static void __exit holamundo_exit(void)
{
  pr_alert("Adios, %s\n", whom);
}

module_init(holamundo_init);
module_exit(holamundo_exit);
```

Declaramdo un parámetro de un módulo

```
module_param(
    name, /* nombre de una variable ya definida */
    type, /* byte, short, ushort, int, uint, long, ulong,
             charp, bool o invbool. */
    perm  /* para /sys/module/<module_name>/parameters/<param>,
             0: sin fichero de valor de parametro */
);

/* Ejemplo */
static int irq=5;
module_param(irq, int, S_IRUGO);
```

Los arrays de parámetros de módulos también son posibles con `module_param_array()`.

Nuevo driver en las fuentes del Kernel 1/2

- Para añadir un nuevo driver a las fuentes del kernel:
  - Añadir el nuevo archivo fuente al directorio apropiado de las fuentes. Ejemplo: `drivers/usb/serial/navman.c`
  - El caso común son los drivers de un único fichero, incluso aunque este fichero tenga miles de líneas de código. Solo los drivers extremadamente grandes se dividen en varios ficheros o tienen su propio directorio.
  - Describir la interfaz de configuración para el nuevo driver añadiendo las siguientes líneas al archivo `Kconfig` en ese directorio:

```
config USB_SERIAL_NAVMAN
        tristate "USB Navman GPS device"
        depends on USB_SERIAL
        help
          To compile this driver as a module, choose M
          here: the module will be called navman.
```

Nuevo driver en las fuentes del Kernel 2/2

- Añadir una linea en el archivo `Makefile` basada en la configuración `Kconfig`:
  `obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o`
- Le dice al sistema de compilación del kernel que compile `navman.c` cuando la opción `USB_SERIAL_NAVMAN` está habilitada. Funciona tanto si se compila estáticamente como si se hace como módulo.
  - Ejecutar `make xconfig` y ver las nuevas opciones
  - Ejecutar `make` y los nuevos archivos se compilan

Laboratorio - Escibiendo módulos

- Crear, compilar y cargar nuestro primer módulo
- Añadir parámetros del módulo
- Acceder al interior del kernel desde el módulo

APIs de propósito general del kernel

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Utilidades de memoria/strings

- En `linux/string.h`
  - Relacionadas con la memoria: `memset()`, `memcpy()`, `memmove()`, `memscan()`, `memcmp()`, `memchr()`
  - Relacionadas con los string: `strcpy()`, `strcat()`, `strcmp()`,`strchr()`, `strrchr()`, `strlen()` y variantes
  - Reservar y copiar un string: `kstrdup()`, `kstrndup()`
  - Reservar y copiar un área de memoria: `kmemdup()`
- En `linux/kernel.h`
  - Conversión string a int: `simple_strtoul()`, `simple_strtol()`, `simple_strtoull()`, `simple_strtoll()`
  - Otras funciones para strings: `sprintf()`, `sscanf()`

Listas enlazadas

- Servicio de listas enlazadas en `linux/list.h`
  - Usado en miles de sitios en el kernel
- Añadir un miembro `struct list_head` a la estructura cuyas instancias serán parte de la lista enlazada. Normalmente, se llama `node` cuando cada instancia necesita ser solo parte de una única lista.
- Definir la lista con la macro `LIST_HEAD()` para una lista global, o definir un elemento `struct list_head` e inicializarlo `INIT_LIST_HEAD()` para listas embebidas en una estructura
- Después, usar la API `list_*()` para manipular la lista
  - Añadir elementos: `list_add()`, `list_add_tail()`
  - Eliminar, mover o reemplazar elementos: `list_del()`, `list_move()`, `list_move_tail()`, `list_replace()`
  - Probar la lista: `list_empty()`
  - Iterar sobre la lista: familia `list_for_each_*()` de macros

Ejemplos de listas enlazadas (1)

- De `include/linux/atmel_tc.h`

```
/*
 * Definition of a list element, with a
 * struct list_head member
 */
struct atmel_tc
{
    /* some members */
    struct list_head node;
};
```

Ejemplos de listas enlazadas (2)

- From drivers/misc/atmel_tclib.c

```
/* Define the global list */
static LIST_HEAD(tc_list);

static int __init tc_probe(struct platform_device *pdev) {
    struct atmel_tc *tc;
    tc = kzalloc(sizeof(struct atmel_tc), GFP_KERNEL);
    /* Add an element to the list */
    list_add_tail(&tc->node, &tc_list);
}

struct atmel_tc *atmel_tc_alloc(unsigned block, const char *name)
{
    struct atmel_tc *tc;
    /* Iterate over the list elements */
    list_for_each_entry(tc, &tc_list, node) {
        /* Do something with tc */
    }
    [...]
}
```

Modelo de dispositivos y drivers de Linux

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Introducción

¿La necesidad de un modelo de dispositivo?

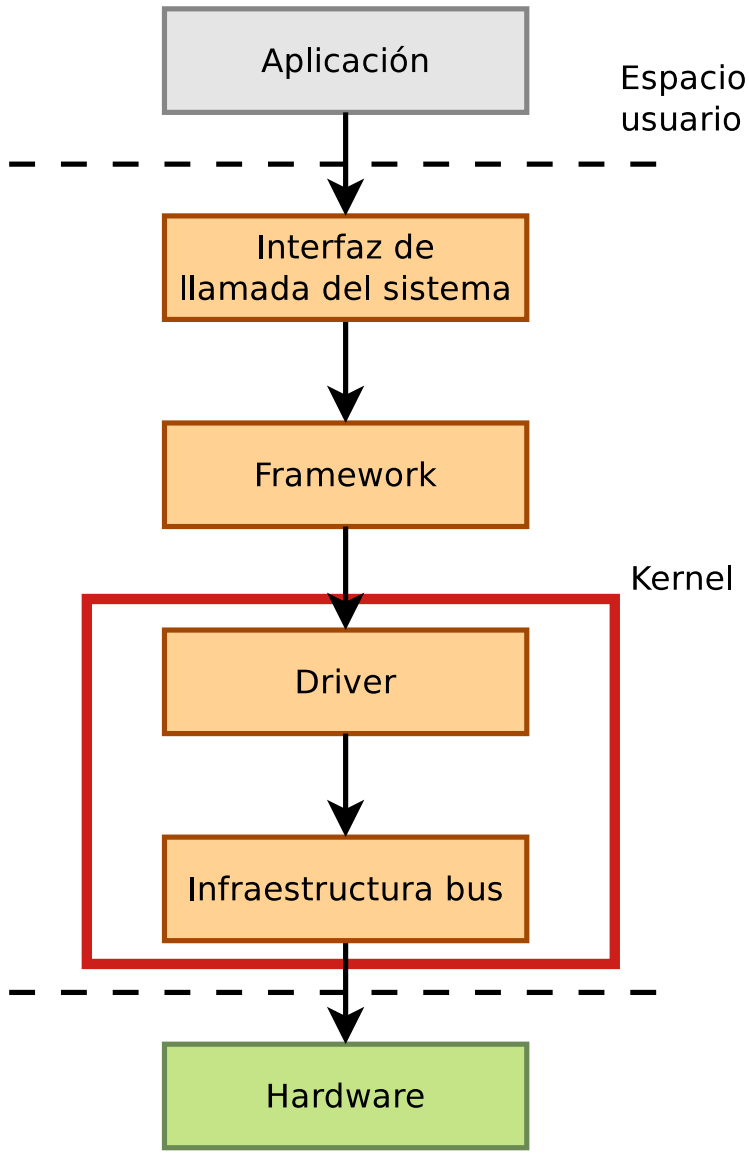
- El kernel de Linux corre sobre un amplio rango de arquitecturas y plataformas hardware, y por tanto necesita **maximizar la reutilización** de código entre plataformas.
- Por ejemplo, queremos que el mismo *driver de dispositivo USB* sea utilizable en un PC x86, o en una plataforma ARM, aun cuando los controladores USB usados en esas plataformas son diferentes.
- Esto requiere una organización limpia del código, con los *drivers de dispositivo* separados de los *drivers de las controladoras*, la descripción del hardware separada de los drivers en sí, etc.
- Esto es lo que permite el **Modelo de Dispositivos** del kernel de Linux, además de otras ventajas cubiertas en esta sección.

Kernel y Drivers de Dispositivos

En Linux, un driver siempre tiene interfaz con:

- un **framework** que permite al driver exponer las características hardware de una forma genérica.
- una **infraestructura de bus**, parte del modelo de dispositivo, para detectar/comunicarse con el hardware.

Esta sección se centra en el *modelo de dispositivo*, mientras que los *frameworks del kernel* se tratan más adelante en este curso.



Estructuras de datos del modelo de dispositivo

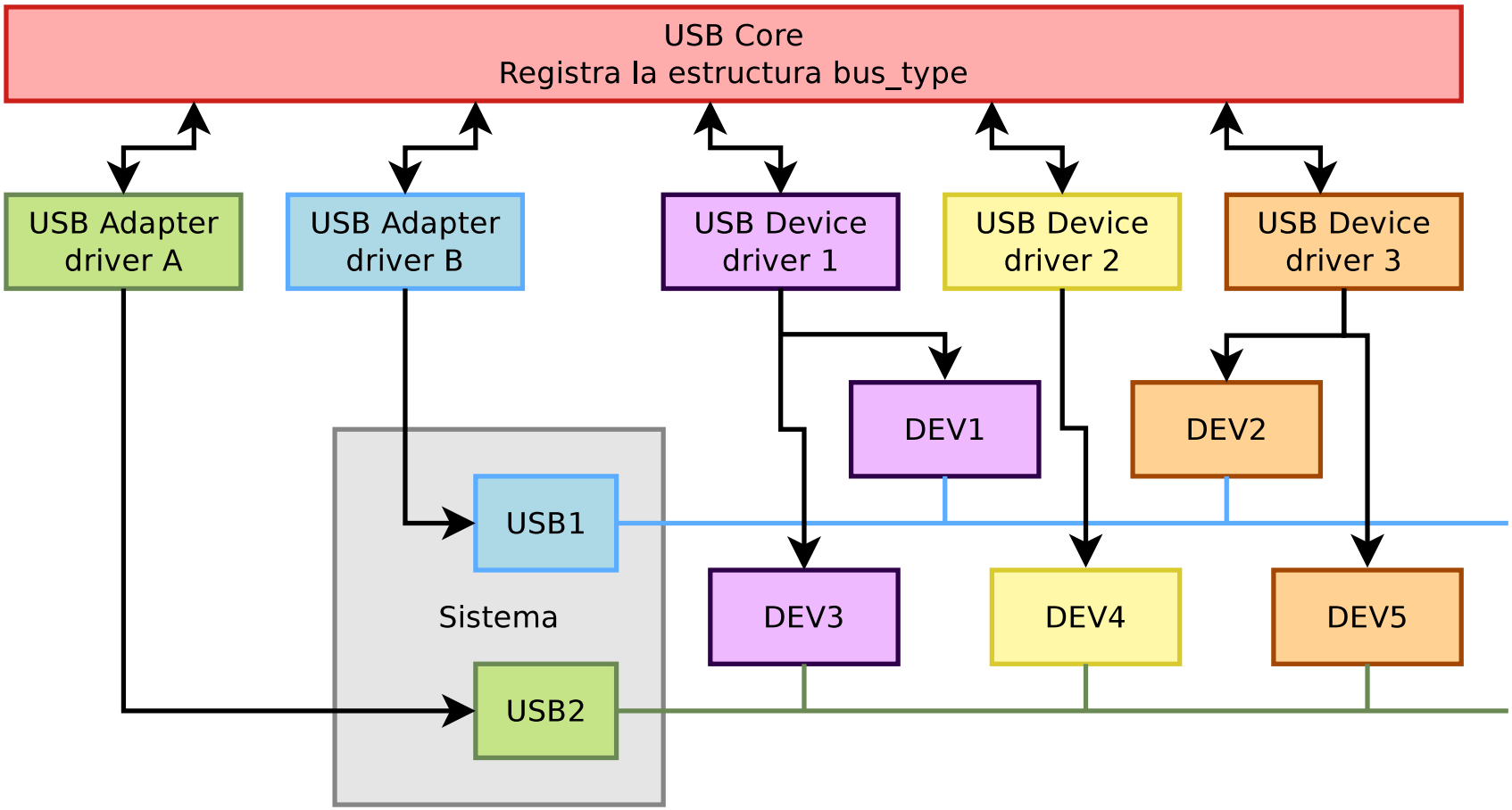
- El *modelo de dispositivo* se organiza alrededor de tres estructuras de datos principales:
  - La estructura `struct bus_type`, que representa un tipo de bus (USB, PCI, I2C, etc.)
  - La estructura `struct device_driver`, que representa un driver capaz de manejar ciertos dispositivos en un cierto bus.
  - La estructura `struct device`, que representa un dispositivo conectado a un bus.
- El kernel usa herencia para crear versiones más especializadas de `struct device_driver` y `struct device` para cada subsistema de bus.
- Para explorar el modelo de dispositivo, vamos a:
  - En primer lugar, echar un vistazo a un bus popular que ofrece enumeración dinámica, el *bus USB*
  - Continuar estudiando como se manejan los buses que no ofrecen enumeración dinámica.

Drivers de los Buses

- El primer componente del modelo es el driver del bus
  - Un driver de bus para cada tipo de bus: USB, PCI, SPI, MMC, I2C, etc.
- Es responsable de
  - Registrar el tipo de bus (`struct bus_type`)
  - Permitir el registro de los drivers de los adaptadores (controladores USB, adaptadores I2C, etc.), capaces de detectar los dispositivos conectados, y proporcionar un mecanismo de comunicación con los dipositivos
  - Permitir el registro de los drivers de dispositivo (dispositivos USB, dispositivos I2C, dispositivos PCI, etc.), gestionando los dispositivos
  - Enlazar los drivers de los dispositivos con los dispositivos detectados por los drivers de las adaptadores.
  - Proporciona una API tanto para los drivers de las adaptadores como para los drivers de los dispositivos
  - Definir las estructuras específicas de los driver y los dispositivos, principalmente `struct usb_driver` y `struct usb_interface`

Ejemplo del bus USB

Ejemplo: Bus USB 1/2



Ejemplo: Bus USB 2/2

- Infraestructura del núcleo (driver del bus)
  - `drivers/usb/core`
  - `struct bus_type` se define en `drivers/usb/core/driver.c` y se registra en `drivers/usb/core/usb.c`
- Drivers del adaptador
  - `drivers/usb/host`
  - Para EHCI, UHCI, OHCI, XHCI, y sus implementaciones en varios sistemas (Atmel, IXP, Xilinx, OMAP, Samsung, PXA, etc.)
- Drivers de dispositivo
  - En todas partes en el árbol del kernel, clasificados por su tipo

Ejemplo de Driver de Dispositivo

- Para ilustrar como se implementan los drivers para trabajar con el modelo de dispositivo, estudiaremos el código fuente de un driver para una tarjeta de red USB
  - Es un dispositivo USB, por lo que tiene que ser un driver de dispositivo USB
  - Es un dispositivo de red, por lo que tiene que ser, efectivamente, un dispositivo de red
  - La mayoría de los drivers cuentan con una infraestructura de bus (en este caso, USB) y se registran a sí mismos en un framework (en este caso, red)
- Solo nos fijaremos en el lado del driver de dispositivo, no en el lado del driver del adaptador
- El driver en el que vamos a fijarnos es `drivers/net/usb/rtl8150.c`

Identificadores de dispositivo

- Define el conjunto de dispositivos que ese driver puede gestionar, de forma que el núcleo USB sepa para qué dispositivos debería usarse este driver
- La macro `MODULE_DEVICE_TABLE()` permite a `depmod` extraer en tiempo de compilación la relación entre los identificadores de dispositivo y los drivers, para que los drivers puedan ser cargados automáticamente por `udev`. Véase `/lib/modules/$(uname -r)/modules.{alias,usbmap}`

```
static struct usb_device_id rtl8150_table[] = {
    { USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150) },
    { USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX) },
    { USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR) },
    { USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX) },
    [...]
    {}
};
MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Instanciación de usb_driver

- `struct usb_driver` es una estructura definida por el núcleo USB. Cada driver de dispositivo USB debe instanciarla, y registrarse a sí mismo en el núcleo USB (*USB core*) usando esta estructura
- Esta estructura hereda de `struct device_driver`, que se define en el modelo de dispositivo.

```
static struct usb_driver rtl8150_driver = {
    .name = "rtl8150",
    .probe = rtl8150_probe,
    .disconnect = rtl8150_disconnect,
    .id_table = rtl8150_table,
    .suspend = rtl8150_suspend,
    .resume = rtl8150_resume
};
```

(Des)Registro del driver

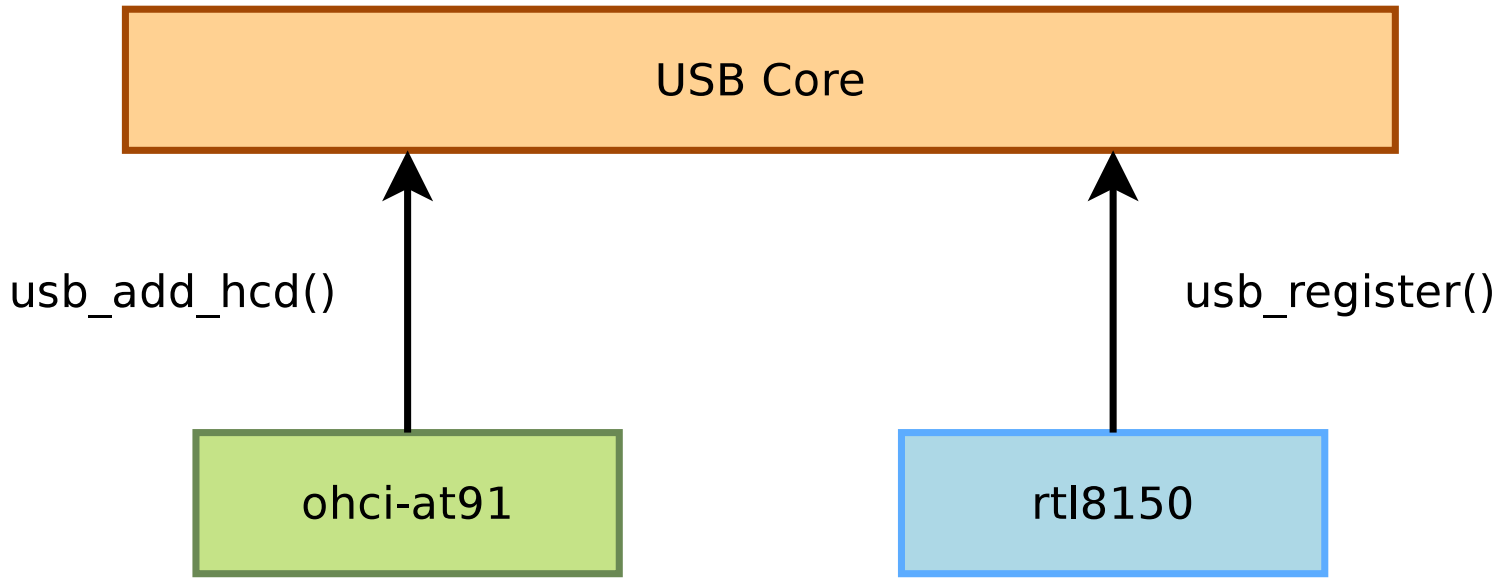
- Cuando se carga o descarga el driver, debe registrarse o desregistrarse a sí mismo en el núcleo USB
- Se hace usando `usb_register()` y `usb_deregister()`, proporcionadas por el núcleo USB.

```
static int __init usb_rtl8150_init(void)
{
    return usb_register(&rtl8150_driver);
}

static void __exit usb_rtl8150_exit(void)
{
    usb_deregister(&rtl8150_driver);
}

module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

- Nota: este código ha sido reemplazado por una llamada más corta a la macro `module_usb_driver()`.

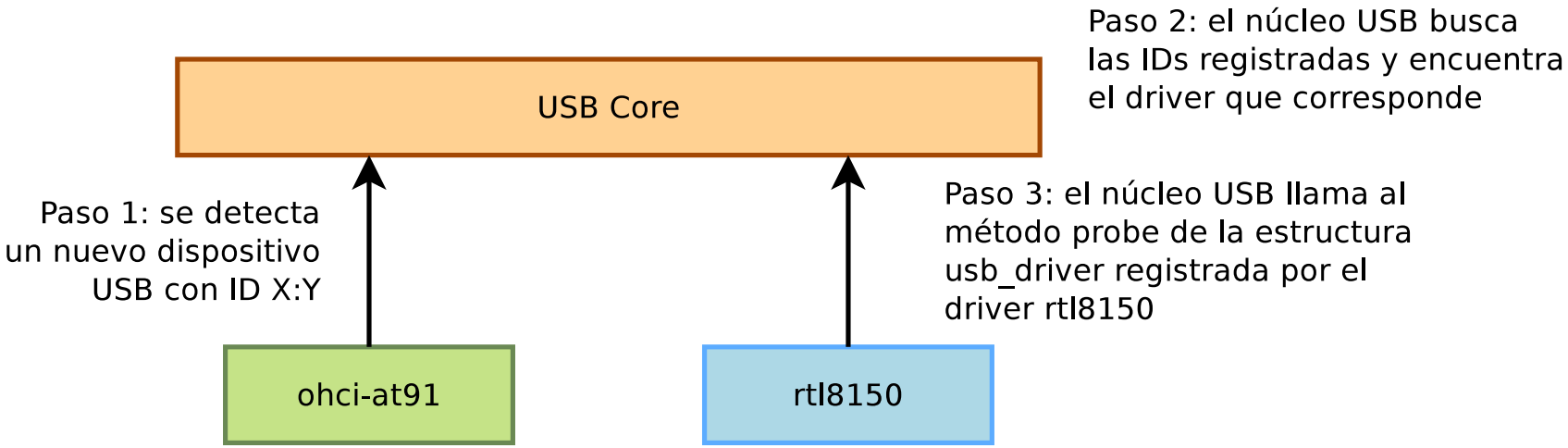
En la inicialización

- El driver del adaptador USB que corresponde al controlador USB del sistema se registra a sí mismo en el núcleo USB
- El driver de dispositivo `rtl8150` se registra a sí mismo en el núcleo USB



- El núcleo USB conoce ahora la asociación entre las ID vendedor/producto de `rtl8150` y la estructura `struct usb_driver` de este driver

Cuando se detecta un dispositivo



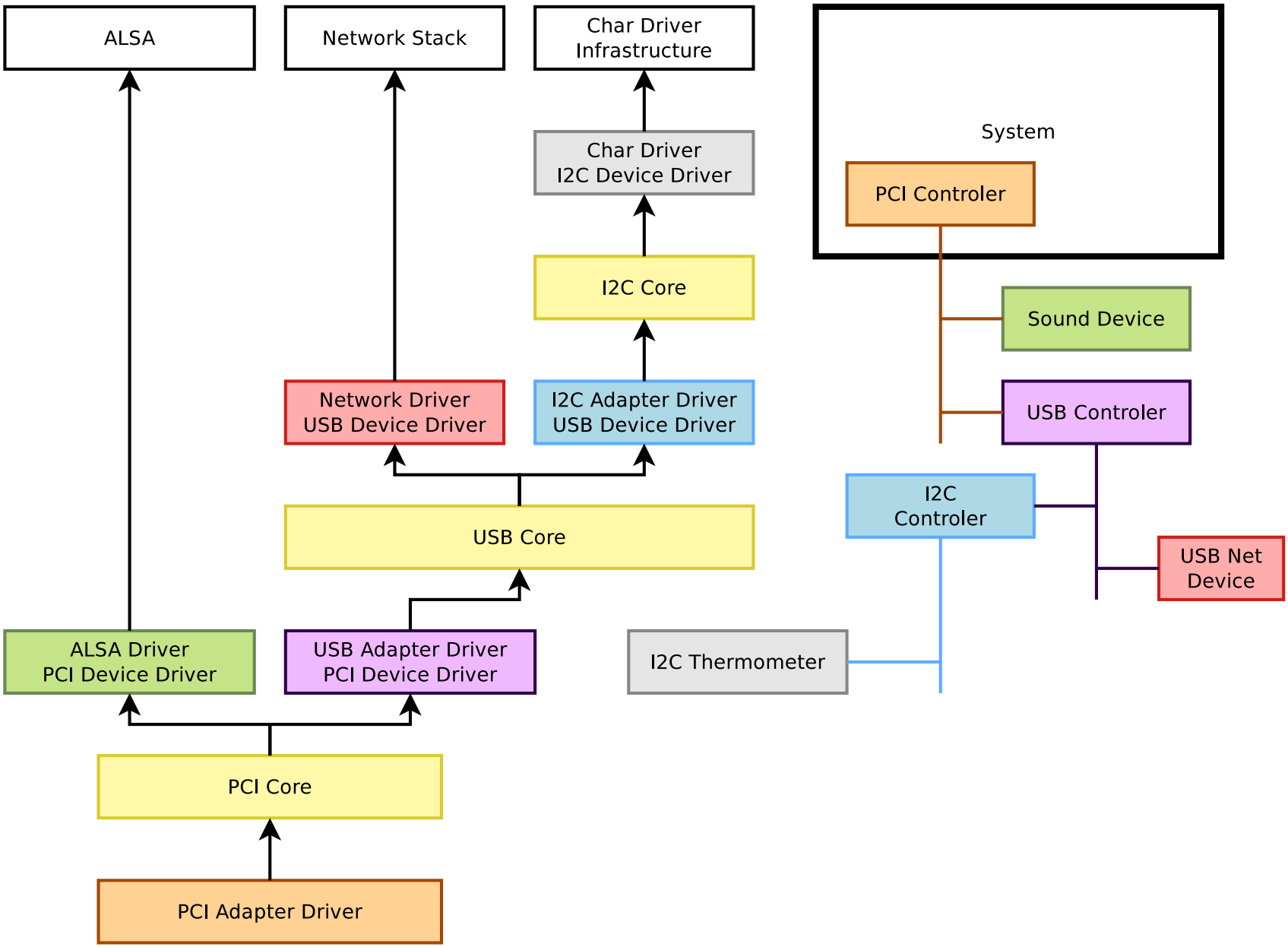
Método Probe

- El método `probe()` recibe como argumento una estructura que describe el dispositivo, normalmente especializada por la infraestructura del bus (`struct pci_dev`, `struct usb_interface`, etc.)
- Esta función es responsable de
  - Inicializar el dispositivo, mapear la memoria de E/S, registrar los manejadores de las interrupciones. La infraestructura de bus proporciona métodos para conseguir las direcciones, los números de interrupción y otra información específica del driver.
  - Registrar el dispositivo en el framework adecuado del kernel, por ejemplo en la infraestructura de red.

Ejemplo de Método Probe

```
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    [...]
    dev = netdev_priv(netdev);
    tasklet_init(&dev->tl, rx_fixup, (unsigned long)dev);
    spin_lock_init(&dev->rx_pool_lock);
    [...]
    netdev->netdev_ops = &rtl8150_netdev_ops;
    alloc_all_urbs(dev);
    [...]
    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);
    register_netdev(netdev);

    return 0;
}
```

El Modelo es Recursivo



Drivers de plataforma - Platform drivers

Buses no detectables

- En los sistemas embebidos, los dispositivos no se conectan frecuentemente a través de un bus que permita enumeración, hotplugging y proporcione identificadores únicos para los dispositivos.
- Por ejemplo, los dispositivos en los buses I2C o SPI, o los dispositivos que son, directamente, parte del system-on-chip.
- Sin embargo, seguimos queriendo que todos estos dispositivos sean parte del modelo de dispositivo.
- Estos dispositivos, en lugar de ser detectados dinámicamente, deben ser descritos estáticamente, ya sea:
  - En el código fuente del kernel.
  - En el *Device Tree*, un archivo de descripción de hardware usado en algunas arquitecturas.

Dispositivos de plataforma

- Entre los dispositivos no detectables, una enorme familia son los dispositivos que son directamente parte de un system-on-chip: controladores UART, controladores Ethernet, controladores SPI or I2C controllers, dispositivos gráficos o de audio, etc.
- En el kernel de Linux, se ha creado un bus especial, llamado bus de plataforma (**platform bus**), para manejar tales dispositivos.
- Soporta drivers de plataforma (**platform drivers**) que manejan dispositivos de plataforma (**platform devices**).
- Funciona como cualquier otro bus (USB, PCI), excepto porque los dispositivos se enumeran estáticamente en lugar de descubrirse dinámicamente.

Implementación de un Platform Driver

- El driver implementa una estructura `struct platform_driver` (ejemplo tomado de `drivers/serial/imx.c`)

```
static struct platform_driver serial_imx_driver = {
    .probe = serial_imx_probe,
    .remove = serial_imx_remove,
    .driver = {
        .name = "imx-uart",
        .owner = THIS_MODULE,
    },
};
```

- Y registra su driver a la infraestructura de driver de plataforma

```
static int __init imx_serial_init(void) {
    ret = platform_driver_register(&serial_imx_driver);
}

static void __exit imx_serial_cleanup(void) {
    platform_driver_unregister(&serial_imx_driver);
}
```

Instanciación: estilo tradicional (1/2)

- Como los dispositivos de plataforma no pueden detectarse dinámicamente, se definen estáticamente
  - Por instanciación directa de estructuras `struct platform_device`, como se hace en algunas plataformas ARM. La definición se hace en código específico de la placa o el SoC.
  - Usando un *device tree*, como se hace en Power PC (y en algunas plataformas ARM) del cual se crean estructuras `struct platform_device`
- Ejemplo en ARM, donde la instanciación se hace en `arch/arm/mach-imx/mx1ads.c`

```
static struct platform_device imx_uart1_device = {
    .name = "imx-uart",
    .id = 0,
    .num_resources = ARRAY_SIZE(imx_uart1_resources),
    .resource = imx_uart1_resources,
    .dev = {
        .platform_data = &uart_pdata,
    }
};
```

Instanciación: estilo tradicional (2/2)

- El dispositivo es parte de una lista

```
static struct platform_device *devices[] __initdata = {
    &cs89x0_device,
    &imx_uart1_device,
    &imx_uart2_device,
};
```

- Y la lista de dispositivos se añade al sistema durante la inicialización de la placa

```
static void __init mx1ads_init(void)
{
    [...]
    platform_add_devices(devices, ARRAY_SIZE(devices));
}

MACHINE_START(MX1ADS, "Freescale MX1ADS")
    [...]
    .init_machine = mx1ads_init,
MACHINE_END
```

El mecanismo de Recursos

- Cada dispositivo gestionado por un driver particular usa, habitualmente, diferentes recursos hardware: direcciones para los registros E/S, canales DMA, líneas IRQ, etc.
- Esta información se puede representar usando `struct resource`, y un array de `struct resource` se asocia a un `struct platform_device`
- Permite que un driver sea instanciado para múltiples dispositivos que funcionan de manera similar, pero con diferentes direcciones, IRQs, etc.

Declarar recursos

```
static struct resource imx_uart1_resources[] = {
    [0] = {
        .start = 0x00206000,
        .end = 0x002060FF,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = (UART1_MINT_RX),
        .end = (UART1_MINT_RX),
        .flags = IORESOURCE_IRQ,
    },
};
```

Usar recursos

- Cuando se añade un `struct platform_device` al sistema usando `platform_add_device()`, se llama al método `probe()` del driver de plataforma
- Este método es responsable de inicializar el hardware, registrar el dispositivo en el framework adecuado (en nuestro caso, el framework de driver serie)
- El driver de plataforma tiene acceso a los recursos de E/S:

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
sport->rxirq = platform_get_irq(pdev, 0);
```

Mecanismo platform_data

- Además de los recursos bien definidos, muchos drivers requieren información específica del driver para cada dispositivo de plataforma
- Esta información se puede pasar usando el campo `struct platform_data` de `struct device` (del cual hereda `struct platform_device`)
- Como es un puntero `void *`, se puede usar para pasar cualquier tipo de información.
  - Típicamente, cada driver define una estructura a través de la cual pasar información `struct platform_data`

Ejemplo platform_data 1/2

- El driver de puerto serie i.MX define la siguiente estructura para que se pase a través de `struct platform_data`

```
struct imxuart_platform_data {
    int (*init)(struct platform_device *pdev);
    void (*exit)(struct platform_device *pdev);
    unsigned int flags;
    void (*irda_enable)(int enable);
    unsigned int irda_inv_rx:1;
    unsigned int irda_inv_tx:1;
    unsigned short transceiver_delay;
};
```

- El código de la placa MX1ADS instancia tal estructura

```
static struct imxuart_platform_data uart1_pdata = {
    .flags = IMXUART_HAVE_RTSCTS,
};
```

Ejemplo platform_data 2/2

- La estructura `uart_pdata` se asocia a la estructura `struct platform_device` en el archivo de la placa MX1ADS (el código real es ligeramente más complicado)

```
struct platform_device mx1ads_uart1 = {
    .name = "imx-uart",
    .dev {
         .platform_data = &uart1_pdata,
    },
    .resource = imx_uart1_resources,
    [...]
};
```

- El driver puede acceder a los platform data:

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imxuart_platform_data *pdata;
    pdata = pdev->dev.platform_data;
    if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
        sport->have_rtscts = 1;
    [...]
```

Device Tree

- En muchas arquitecturas embebidas, la instanciación manual de dispositivos de plataforma se consideraba demasiado tediosa y difícilmente mantenible.
- Tales arquitecturas se están trasladando, o se han trasladado, al uso del *Device Tree*.
- Es un **árbol de nodos** que modela la jerarquía de dispositivos en el sistema, desde los dispositivos internos del procesador a los dispositivos en la placa.
- Cada nodo puede tener un cierto número de **propiedades** que describen varias propiedades de los dispositivos: direcciones, interrupciones, relojes, etc.
- En el arranque, el kernel recibe una versión compilada, el **Device Tree Blob**, que se interpreta para instanciar todos los dispositivos descritos en el DT.
- En ARM, se localizan en `arch/arm/boot/dts`.

Ejemplo de Device Tree

```
uart0: serial@44e09000 {
        compatible = "ti,omap3-uart";
        ti,hwmods = "uart1";
        clock-frequency = <48000000>;
        reg = <0x44e09000 0x2000>;
        interrupts = <72>;
        status = "disabled";
};
```

- `serial@44e09000` es el **nombre del nodo**
- `uart0` es un **alias**, al que se puede hacer referencia en otras partes del DT como `&uart0`
- las otras líneas son **propiedades**. Sus valores son, habitualmente, strings, listas de enteros o referencias a otros nodos.

Herencia en el Device Tree (1/2)

- Cada plataforma hardware particular tiene su propio *device tree*.
- Sin embargo, varias plataformas hardware usan el mismo procesador y, a veces, varios procesadores de la misma familia comparten ciertas similitudes.
- Para permitir esto, un archivo *device tree* puede incluir a otro. Los árboles descritos por el archivo incluyente se superponen al árbol descrito por el archivo incluido. Esto puede hacerse:
  - O bien usando la declaración `/include/` proporcionada por el lenguaje del Device Tree.
  - O bien usando la declaración `\#include`, que requiere llamar al preprocesador C antes de parsear el Device Tree.

  Linux usa actualmente ambas técnicas (cambia entre una subarquitectura ARM y otra, por ejemplo).

Herencia en el Device Tree (2/2)

Definición del SoC AM33xx

```
/ {
     compatible = "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
              compatible = "ti,omap3-uart";
              reg = <0x44e09000 0x2000>;
              interrupts = <72>;
              status = "disabled";
          };
     };
};
```

am33xx.dtsi

+

Definición de la placa BeagleBone

```
#include "am33xx.dtsi"

/ {
     compatible = "ti,am335x-bone", "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
              pinctrl-names = "default";
              pinctrl-0 = <&uart0_pins>;
              status = "okay";
          };
     };
};
```

am335x-bone.dts

=

DTB compilado

```
/ {
     compatible = "ti,am335x-bone", "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
              compatible = "ti,omap3-uart";
              reg = <0x44e09000 0x2000>;
              interrupts = <72>;
              pinctrl-names = "default";
              pinctrl-0 = <&uart0_pins>;
              status = "okay";
          };
     };
};
```

am335x-bone.dtb

Nota: el DTB real está en formato binario. Aquí se muestra el texto equivalente a su contenido.

Device Tree: string compatible

- Con el *device tree*, un *dispositivo* está ligado con el *driver* correspondiente a través del string **compatible**.
- El campo `of_match_table` de `struct device_driver` lista los string compatibles soportados por el driver.

```
#if defined(CONFIG_OF)
static const struct of_device_id omap_serial_of_match[] = {
        { .compatible = "ti,omap2-uart" },
        { .compatible = "ti,omap3-uart" },
        { .compatible = "ti,omap4-uart" },
        {},
};
MODULE_DEVICE_TABLE(of, omap_serial_of_match);
#endif
static struct platform_driver serial_omap_driver = {
        .probe          = serial_omap_probe,
        .remove         = serial_omap_remove,
        .driver         = {
                .name   = DRIVER_NAME,
                .pm     = &serial_omap_dev_pm_ops,
                .of_match_table = of_match_ptr(omap_serial_of_match),
        },
};
```

Recursos en el Device Tree

- Los drivers usan el mismo mecanismo que se vio anteriormente para recuperar la información básica: números de interrupción, direcciones físicas, etc.
- La lista de recursos disponibles será compilada por el kernel en el arranque a partir del device tree, de forma que no sea necesario hacer ninguna búsqueda irrelevante en el DT cuando se carga el driver.
- Cualquier información adicional será específica de un driver o de la clase a la que pertenezca, definiendo los *bindings*

Device Tree Bindings

- El string compatible y las propiedades asociadas definen lo que se llama un *device tree binding*.
- Los *Device tree bindings* están todos documentados en `Documentation/devicetree/bindings` .
- Dado que el Device Tree normalmente es parte de la API del kernel, los *bindings* deben permanecer compatibles a lo largo del tiempo.
  - Un nuevo kernel debe ser capaz de usar un Device Tree viejo.
  - Esto requiere un diseño muy cuidadoso de los bindings. Se revisan todos en la lista de correo `devicetree@vger.kernel.org`.
- Un Device Tree binding solo debería contener una *descripción del hardware* y no una *configuración*.
  - Un número de interrupción puede ser parte del Device Tree puesto que describe el hardware.
  - Pero no si debe usarse DMA o no para un dispositivo, puesto que es una elección de configuración.

sysfs

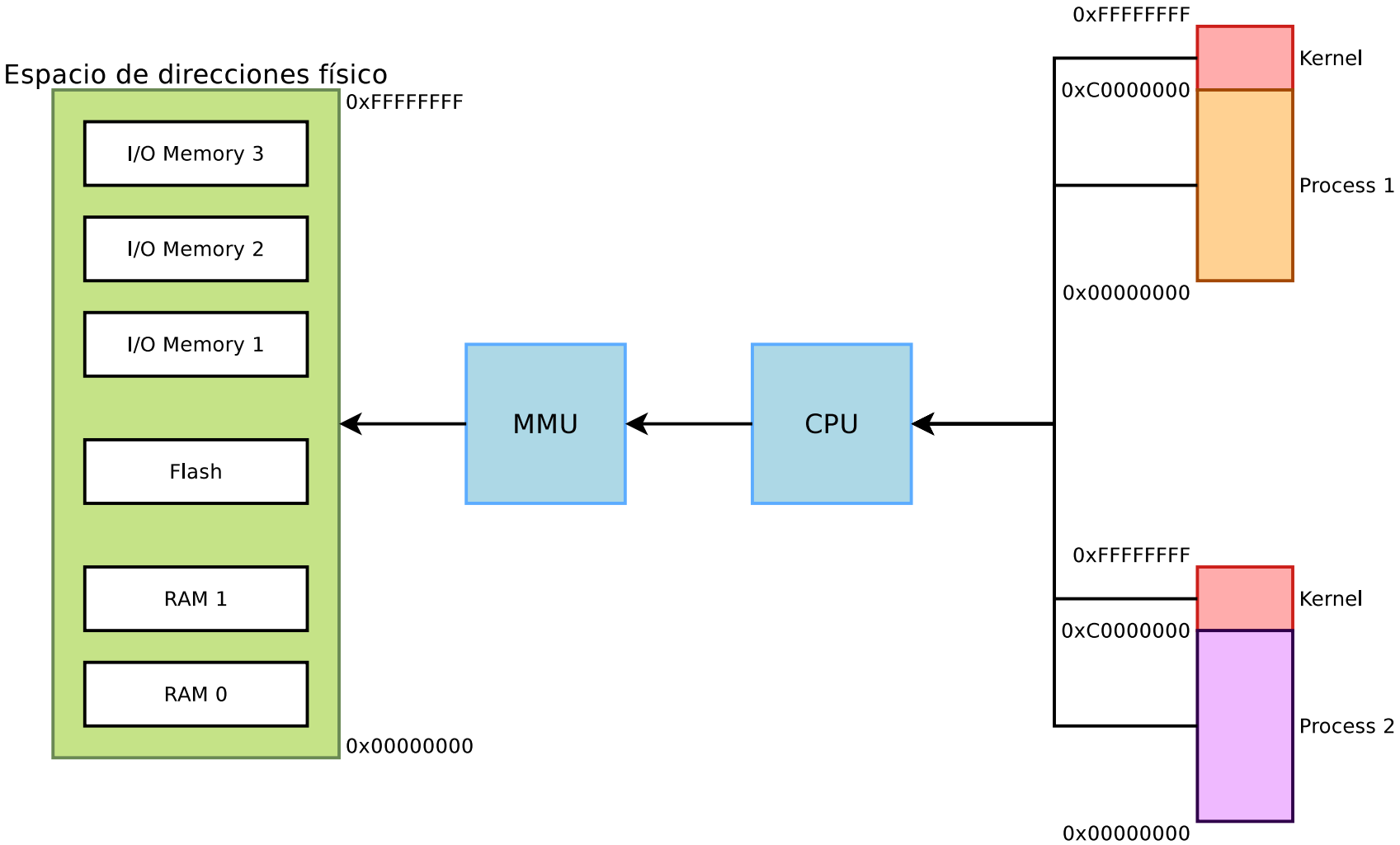
- Las estructuras del bus, dispositivo, drivers, etc. son internas al kernel
- El sistema de ficheros virtual `sysfs` ofrece un mecanismo para exportar esta información al espacio de usuario
- Usado por ejemplo por `udev` para propocionar la carga automática del módulo, la carga del firmware, la creación del archivo de dispositivo, etc.
- `sysfs` está normalmente montado en `/sys`
  - `/sys/bus/` contiene la lista de buses
  - `/sys/devices/` contiene la lista de dispositivos
  - `/sys/class` enumera los dispositivos por clase (`net`, `input`, `block`...), sea cual sea el bus al que están conectados.

Gestión de memoria

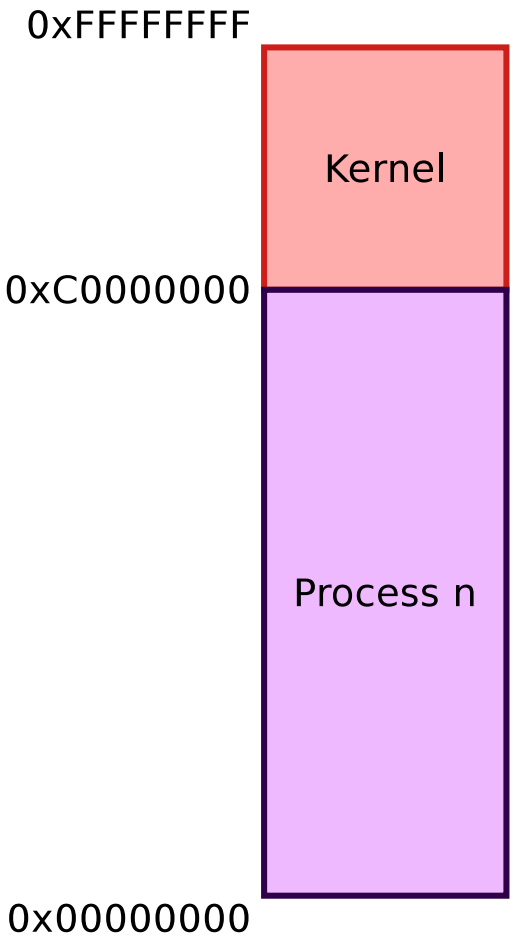
Laboratorio de Sistemas Inteligentes

Última actualización: July 10, 2014.

Memoria física y virtual

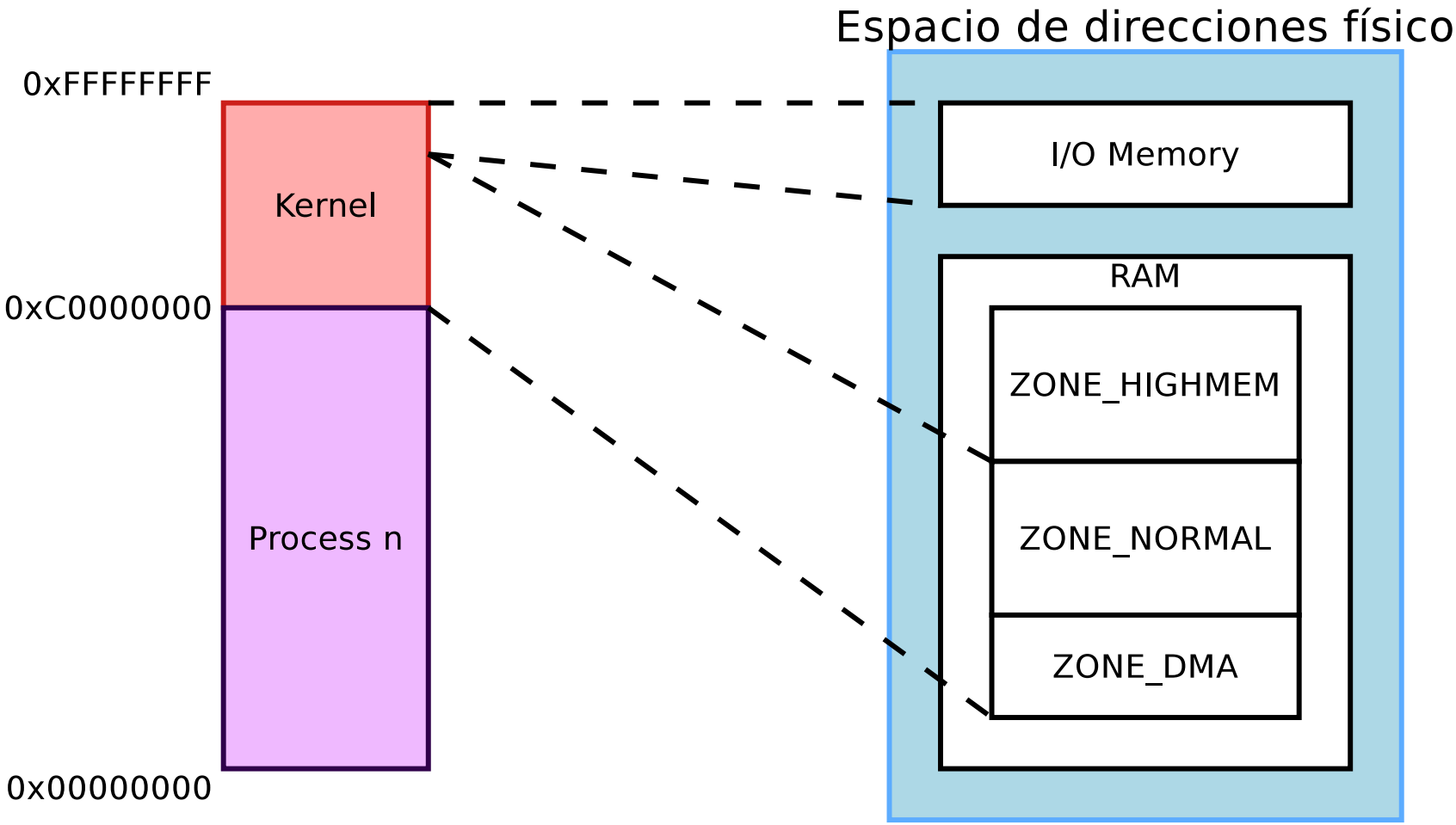


Organización de la memoria virtual



- 1GB reservado para el espacio del kernel
  - Contiene el código del kernel y estructuras de datos básicas, idénticas en todos los espacios de direcciones
  - La mayor parte de la memoria puede ser un mapeo directo de la memoria física con un offset fijo
- Mapa de memoria completo de 3GB exclusivo para cada proceso del usuario
  - Datos y código de los procesos (programa, pila, ...)
  - Archivos mapeados en memoria
  - No necesariamente mapeado a una memoria física (paginación bajo demanda usada para mapear a páginas de memoria física)
  - Difiere de un espacio de direcciones a otro

Mapeado de memoria físico / virtual



Acceder a más memoria física

- Solamente menos de 1GB de memoria es direccionable directamente a través del espacio de direcciones virtual del kernel
- Si hay presente más memoria física en la plataforma, parte de la memoria no será accesible por el espacio del kernel, pero puede ser usada por el espacio de usuario
- Para permitir al kernel acceder a más memoria física:
  - Cambiar división de memoria 1GB/3GB (2GB/2GB) (`CONFIG_VMSPLIT_3G`) $\Rightarrow$ reduce la memoria total disponible para cada proceso
  - Cambio para la arquitectura de 64 bits. Véase `Documentation/x86/x86_64/mm.txt` para un ejemplo.
  - Activar soporte *highmem* si está disponible para la arquitectura:
    - Permite al kernel mapear partes de su memoria no accesible directamente
    - El mapeado debe requerirse explícitamente
    - Rangos de direcciones limitados reservados para este uso

Accediendo a incluso más memoria física

- Si la plataforma de 32 bits posee más de 4 GB, simplemente no puede ser mapeada
- PAE (Physical Address Expansion) podría estar soportado por la arquitectura
- Añade algunos bits de extensión de direcciones usados para indexar áreas de memoria
- Permite acceder hasta a 64 GB de memoria física en x86
- Nótese que cada proceso del espacio de usuario sigue limitado a un espacio de memoria de 3 GB

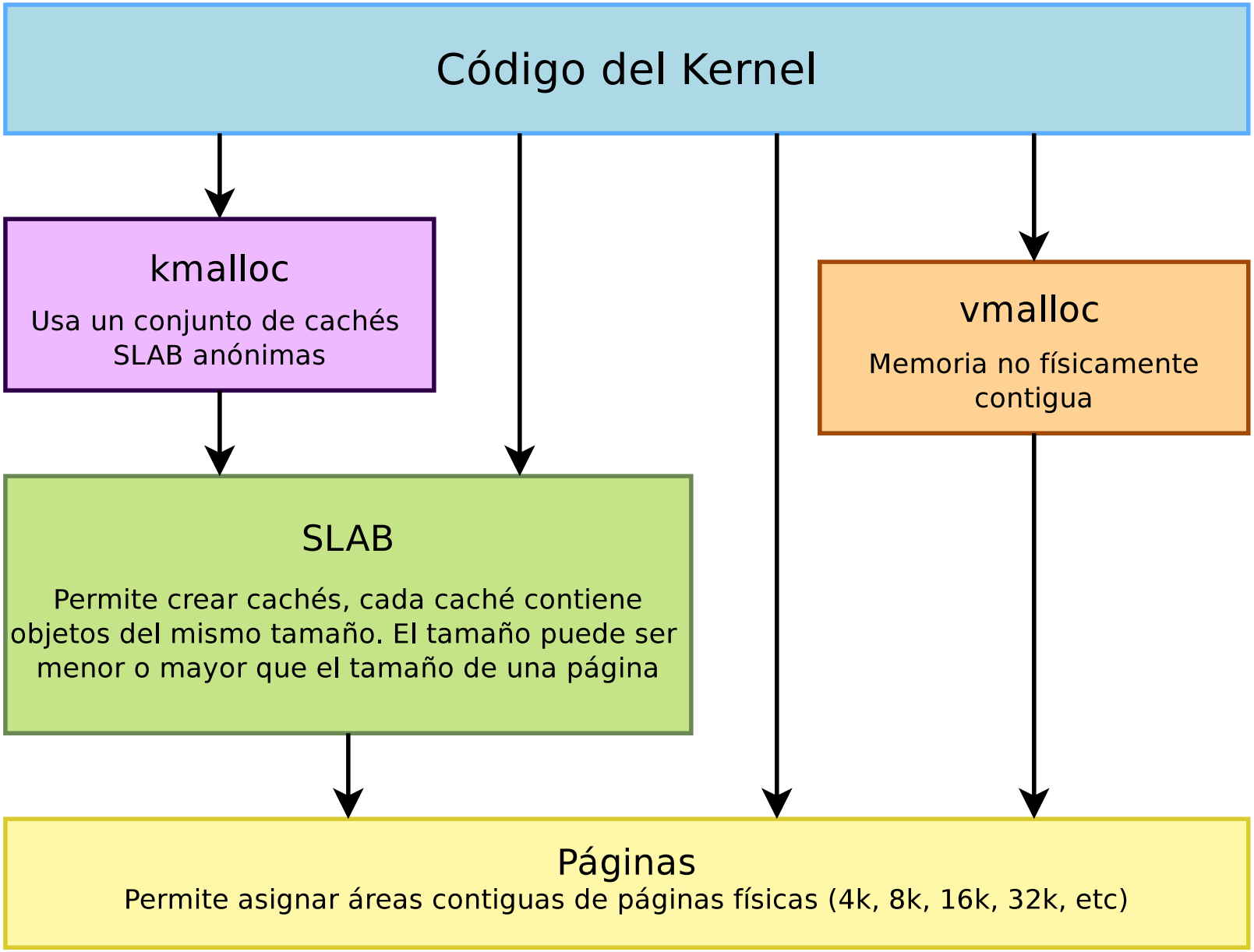
Notas sobre la memoria del espacio de usuario

- La memoria nueva en el espacio de usuario se asigna o bien a partir de la memoria del proceso ya asignada, o bien usando la llamada del sistema `mmap`
- Nótese que la memoria asignada podría no estar físicamente asignada:
  - El kernel usa paginación bajo demanda (*demand fault paging*) para asignar la página física (la página física se asigna cuando el acceso a la dirección virtual genera un error de paginación)
  - ... o podría haber sido "intercambiada", lo que produce también un error de paginación
- La asignación de memoria en el espacio de usuario tiene permiso para sobre-reservar memoria (más que la memoria física disponible) $\Rightarrow$ puede llevar a la falta de memoria (*out of memory*)
- El OOM killer entra en acción y selecciona un proceso para matarlo y recuperar memoria. Esto es mejor que dejar que el sistema se bloquee por completo.

De vuelta a la memoria del kernel

- Los asignadores de memoria del kernel asignan páginas físicas, y la memoria reservada del kernel no puede ser intercambiada, por lo que no se requiere manejo de fallos en la memoria del kernel.
- La mayor parte de funciones de asignación de memoria del kernel también devuelven una dirección virtual que puede usarse en el espacio del kernel.
- El asignador de bajo nivel de memoria del kernel gestiona las páginas. Esta es la granularidad más fina (normalmente 4 KB, dependiente de la arquitectura).
- Sin embargo, la gestión de memoria del kernel maneja asignaciones de memoria menores a través de su asignador, usado por `kmalloc()`).

Asignadores en el kernel



Asignador de páginas - Page Allocator

- Apropiado para asignaciones de tamaño medio
- Una página es, habitualmente, de 4K, pero puede hacerse mayor en ciertas arquitecturas (sh, mips: 4, 8, 16 o 64 KB, aunque no es configurable en x86 ni arm).
- Estrategia de asignación tal que solo son posibles asignaciones de un número de páginas igual a una potencia de 2: 1 página, 2 páginas, 4 páginas, 8 páginas, 16 páginas, etc.
- El tamaño típico máximo es 8192 KB, pero podría depender de la configuración del kernel.
- El área asignada es virtualmente contigua (por supuesto), pero también físicamente continua. Se asigna en la parte idénticamente mapeada del espacio de memoria del kernel.
  - Esto significa que las áreas grandes podrían no estar disponibles o ser difíciles de conseguir por la fragmentación de la memoria física.

API del asignador de páginas: conseguir páginas libres

- `unsigned long get_zeroed_page(int flags)`
  - Devuelve la dirección virtual de una página libre, inicializada a cero
  - `flags`: véanse las siguientes páginas para más detalles.
- `unsigned long __get_free_page(int flags)`
  - Lo mismo, pero no inicializa los contenidos
- `unsigned long __get_free_pages(int flags, unsigned int order)`
  - Devuelve la dirección virtual de inicio de un área de varias páginas contiguas en la RAM física, siendo el órden `log2(number_of_pages)`. Puede calcularse a partir del tamaño con la función `get_order()`.

API del asignador de páginas: liberar páginas

- `void free_page(unsigned long addr)`
  - Libera una página.
- `void free_pages(unsigned long addr, unsigned int order)`
  - Libera múltiples páginas. Necesario usar el mismo orden que en la asignación.

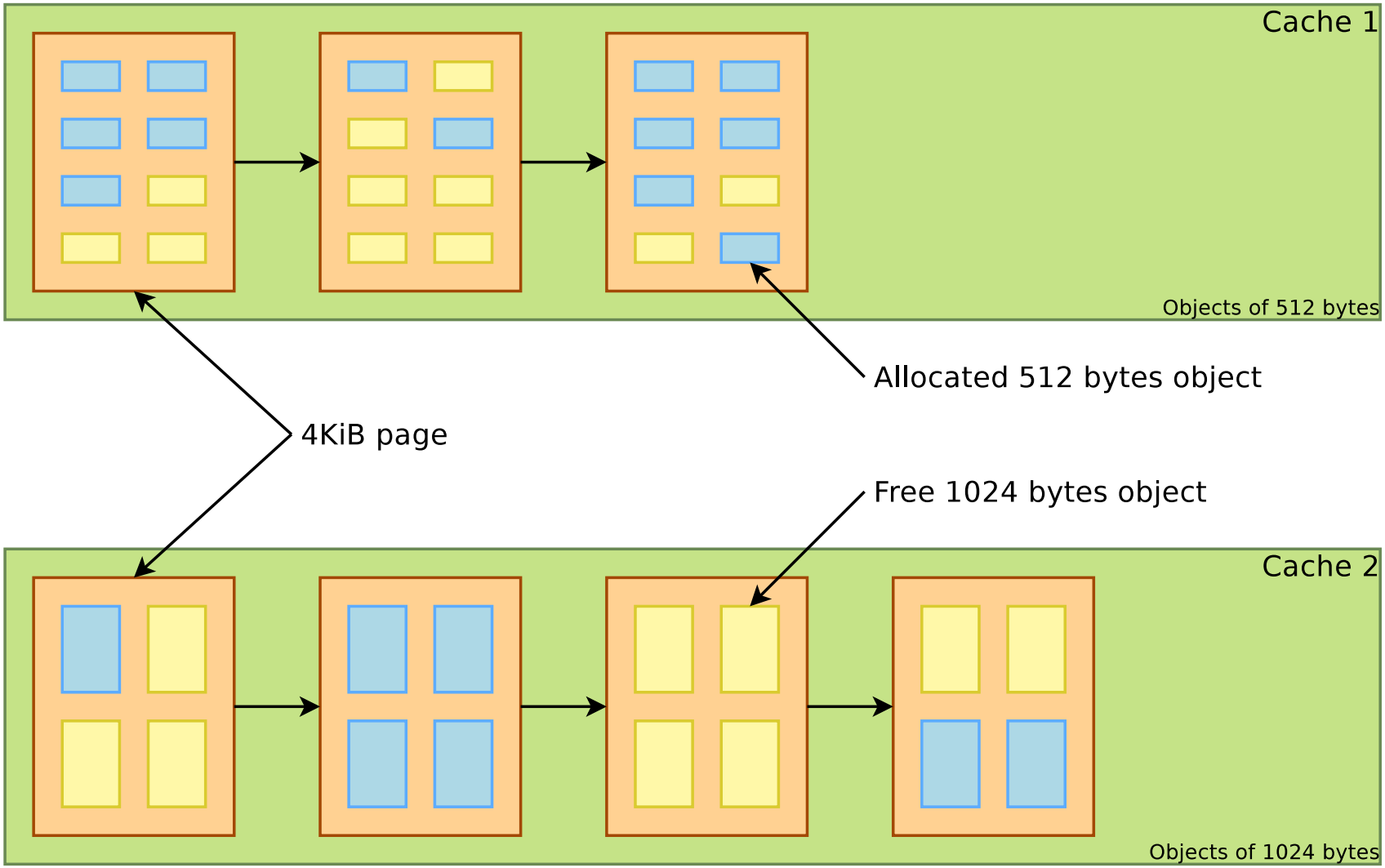
Flags del asignador de página

- Los más comunes son:
  - `GFP_KERNEL`
    - Asignación estándar de la memoria del kernel. La asignación puede bloquear hasta encontrar suficiente memoria disponible. Adecuado para la mayoría de las necesidades, excepto en el contexto de los manejadores de interrupciones.
  - `GFP_ATOMIC`
    - RAM asignada desde código que no puede bloquear (manejadores de interrupciones o secciones críticas). Nunca bloquea, permite el acceso a pools de emergencia, pero puede fallar si no hay memoria libre inmediatamente disponible.
  - `GFP_DMA`
    - Asigna memoria en un área de memoria física utilizable para transferencias DMA.
  - Otras se definen en `include/linux/gfp.h`

Asignador SLAB 1/2

- El asignador SLAB permite crear cachés, que contienen un conjunto de objetos del mismo tamaño
- El tamaño de objeto puede ser menor o mayor que el tamaño de página
- El asignador SLAB se encarga de aumentar o reducir el tamaño de la caché según se necesite, dependiendo del número de objetos asignados. Usa el asignador de página para asignar y liberar páginas.
- Las cachés SLAB se usar para estructuras de datos que están presentes en muchas instancias del kernel: entradas de directorio, objetos de archivos, descriptores de paquetes de red, descriptores de proceso, etc.
  - Véase `/proc/slabinfo`
- Raramente se usan para drivers individuales
- Véase `include/linux/slab.h` para la API

SLAB Allocator 2/2



Asignadores SLAB diferentes

- Hay tres implementaciones diferentes, aunque compatibles con la API, del asignador SLAB en el kernel de Linux. La implementación particular se escoge en tiempo de configuración.
  - SLAB: legacy, asignador muy probado.
    Todavía el asignador por defecto en la mayoría de archivos `defconfig` ARM.
  - SLOB: mucho más simple. Más eficiente en el uso del espacio pero no escala bien. Ahorra unos cuantos cientos de KB en pequeños sistemas (depende de `CONFIG_EXPERT`)
    Linux 3.13 en ARM: usando en 5 archivos `defconfig`
  - SLUB: más reciente y más simple que SLAB, escala mucho mejor (especialmente en sistemas enormes) y crea menos fragmentación.
    Linux 3.13 en ARM: usado en 0 archivos `defconfig`

Asignador kmalloc

- El asignador kmalloc es el asignador de propósito general del kernel de Linux
- Para tamaños pequeños, se basa en cachés genéricas SLAB, llamadas `kmalloc-XXX` en `/proc/slabinfo`
- Para tamaños más grandes, se basa en el asignador de páginas
- Se garantiza que el área asignada es físicamente contigua
- El área asignada se redondea al tamaño de la caché SLAB más pequeña en la que cabe (mientras que usar el asignador SLAB directamente permite tener más flexibilidad)
- Usa las mismas flags que el asignador de páginas (`GFP_KERNEL`, `GFP_ATOMIC`, `GFP_DMA`, etc.)
- Tamaños máximos, en `x86` y `arm`:
  - Por asignación: 4 MB
  - Total de asignaciones: 128 MB
- Debería usarse como asignador primario a menos que haya una razón de peso para usar otro.

API kmalloc 1/2

- `#include <linux/slab.h>`
- `void *kmalloc(size_t size, int flags);`
  - Asigna `size` bytes, y devuelve un puntero al área (dirección virtual)
  - `size`: número de bytes a asignar
  - `flags`: las mismas flags que el asignador de páginas
- `void kfree(const void *objp);`
  - Libera un área asignada
- Ejemplo: (`drivers/infiniband/core/cache.c`)

```
struct ib_update_work *work;
work = kmalloc(sizeof *work, GFP_ATOMIC);
...
kfree(work);
```

API kmalloc 2/2

- `void *kzalloc(size_t size, gfp_t flags);`
  - Asigna un buffer inicializado a cero
- `void *kcalloc(size_t n, size_t size, gfp_t flags);`
  - Asigna memoria para un array de `n` elementos de tamaño `size`, y pone a cero sus contenidos.
- `void *krealloc(const void *p, size_t new_size, gfp_t flags);`
  - Cambia el tamaño del buffer apuntado por `p` a `new_size`, reasignando un nuevo buffer y copiando los datos, a menos que `new_size` quepa dentro de la alineación del buffer existente.

Funciones devm_ kmalloc

- Liberan automáticamente los buffers asignados cuando el dispositivo o módulo correspondiente se descarga.
- Necesitan tener una referencia a un `struct device`.
- `void *devm_kmalloc(struct device *dev, size_t size, int flags);`
- `void *devm_kzalloc(struct device *dev, size_t size, int flags);`
- `void *devm_kcalloc(struct device *dev, size_t n, size_t size, gfp_t flags)`
- `void *devm_kfree(struct device *dev, void *p);`
  - Útil para liberar inmediatamente un buffer asignado

Véase `Documentation/driver-model/devres.txt` para más detalles sobre recursos gestionados de dispositivos

Flags del asignador de página

- Los más comunes son:
  - `GFP_KERNEL`
    - Asignación estándar de la memoria del kernel. La asignación puede bloquear hasta encontrar suficiente memoria disponible. Adecuado para la mayoría de las necesidades, excepto en el contexto de los manejadores de interrupciones.
  - `GFP_ATOMIC`
    - RAM asignada desde código que no puede bloquear (manejadores de interrupciones o secciones críticas). Nunca bloquea, permite el acceso a pools de emergencia, pero puede fallar si no hay memoria libre inmediatamente disponible.
  - `GFP_DMA`
    - Asigna memoria en un área de memoria física utilizable para transferencias DMA.
  - Otras se definen en `include/linux/gfp.h`

Asignador vmalloc

- El asignador `vmalloc()` puede usarse para obtener zonas de memoria virtualmente contiguas, pero no físicamente contiguas. El tamaño de memoria requerido se redondea hacia arriba hasta la siguiente página.
- El área asignada está en la parte del espacio del kernel del espacio de direcciones, pero fuera del área identicamente mapeada
- Son posibles las asignaciones de áreas bastante grandes (casi tan grandes como la memoria total disponible, puesto que la fragmentación de la memoria física no es un problema, pero estas áreas no pueden usarse para DMA, puesto que DMA suele requerir buffers contiguos físicamente.
- API en `include/linux/vmalloc.h`
  - `void *vmalloc(unsigned long size);`
    - Devuelve una dirección virtual
  - `void vfree(void *addr);`

Memoria y puertos E/S

Laboratorio de Sistemas Inteligentes

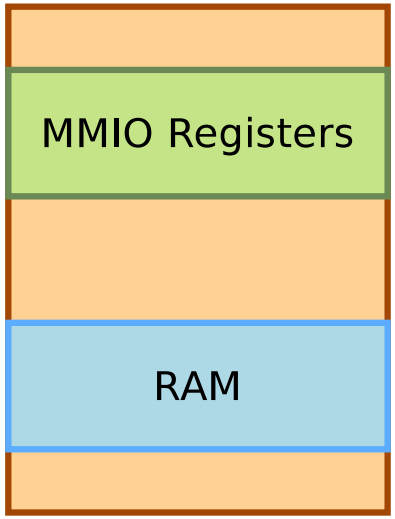
Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

Puerto E/S vs. E/S mapeada en memoria

- MMIO
  - Mismo bus de direcciones para la memoria de direcciones y dispositivos E/S
  - Acceso a los dispositivos E/S usando instrucciones normales
  - El método E/S más ampliamente utilizado en las diferentes arquitecturas soportadas por Linux
- PIO
  - Diferentes espacios de direcciones para la memoria y los dispositivos E/S
  - Usa una clase espacial de instrucciones CPU para acceder a dispositivos E/S
  - Ejemplo en x86: Instrucciones IN y OUT

MMIO vs PIO



Espacio de direcciones en memoria física, accesible mediante instrucciones de carga/almacenamiento normales

Espacio de direcciones E/S separado, accesible con instrucciones específicas

Pedir puertos E/S

- Decir al kernel qué driver está usando qué puertos E/S
- Permite evitar que otros drivers usen los mismos puertos E/S, pero es puramente voluntario.
- ```
  struct resource *request_region(
        unsigned long start,
        unsigned long len,
        char *name);
  ```
- Intenta reservar la región dada y devuelve `NULL` si no tiene éxito.
- `request_region(0x0170, 8, "ide1");`
- ```
  void release_region(
      unsigned long start,
      unsigned long len);
  ```

Ejemplo /proc/ioports (x86)

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
...
```

Acceder a puertos E/S

- Funciones para leer/escribir bytes (`b`), palabras (`w`) y "longs" (`l`) a puertos E/S:
  - `unsigned in[bwl](unsigned long port)`
  - `void out[bwl](value, unsigned long port)`
- Y las variantes string: a menudo más eficientes que el bucle en C correspondiente, si el procesador soporta estas operaciones
  - `void ins[bwl](unsigned port, void *addr, unsigned long count)`
  - `void outs[bwl](unsigned port, void *addr, unsigned long count)`
- Ejemplos
  - leer 8 bits
    - `oldlcr = inb(baseio + UART_LCR)`
  - escribir 8 bits
    - `outb(MOXA_MUST_ENTER_ENCHANCE, baseio + UART_LCR)`

Pedir memoria E/S

- Funciones equivalentes a `request_region()` y `release_region()`, pero para memoria E/S
- ```
  struct resource *request_mem_region(
      unsigned long start,
      unsigned long len,
      char *name);
  ```
- ```
  void release_mem_region(
      unsigned long start,
      unsigned long len);
  ```

Ejemplo /proc/iomem

```
00000000-0009efff : System RAM
0009f000-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000cffff : Video ROM
000f0000-000fffff : System ROM
00100000-3ffadfff : System RAM
00100000-0030afff : Kernel code
0030b000-003b4bff : Kernel data
3ffae000-3fffffff : reserved
40000000-400003ff : 0000:00:1f.1
40001000-40001fff : 0000:02:01.0
40400000-407fffff : PCI CardBus #03
40800000-40bfffff : PCI CardBus #03
a0000000-a0000fff : pcmcia_socket0
e8000000-efffffff : PCI Bus #01
...
```

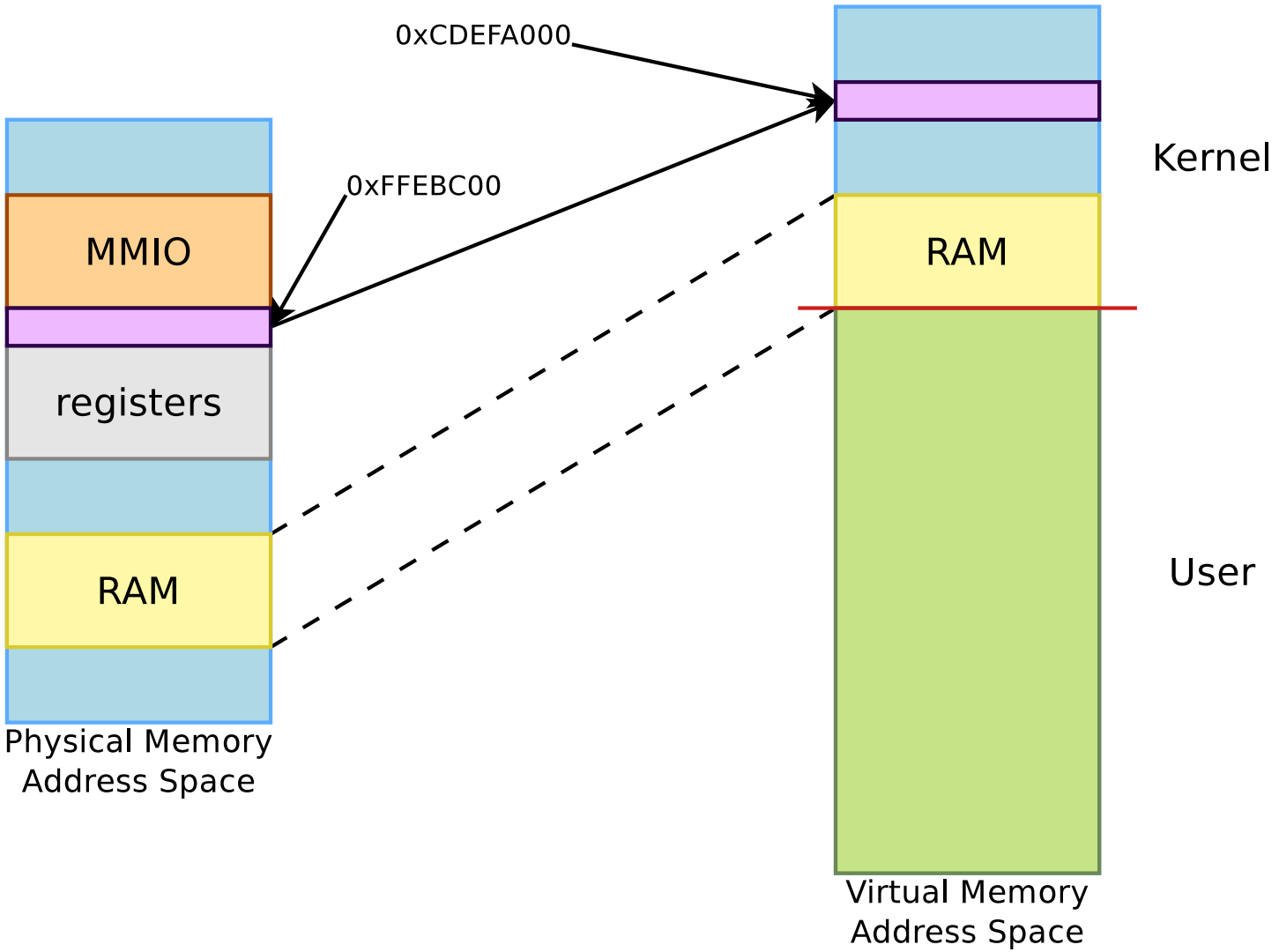
Mapear la memoria E/S en la memoria virtual

- Las instrucciones de carga/almacenamiento trabajan con direcciones virtuales
- Para acceder a la memoria E/S, los drivers tienen que tener una dirección virtual que el procesador pueda manejar, puesto que la memoria E/S no se mapea por defecto en la memoria virtual.
- La función `ioremap` satisface esta necesidad:

```
#include <asm/io.h>

void __iomem *ioremap(phys_addr_t phys_addr,
    unsigned long size);
void iounmap(void __iomem *addr);
```

- Cuidado: compribar que `ioremap()` no devuelve una dirección `NULL`

ioremap()




```
ioremap(0xFFEBC00, 4096) = 0xCDEFA000
```

API dirigida

Usar `request_mem_region()` y `ioremap()` en los drivers de dispositivo está ya obsoleto. Se deberían usar las siguientes funciones "dirigidas" en su lugar, las cuales simplifican la escritura del código y el manejo de errores:

- `devm_ioremap()`
- `devm_iounmap()`
- `devm_request_and_ioremap()`
  - Se ocupa tanto de las operaciones de petición como de las de remapeado

Acceder a dispositivos MMIO

- Leer directamente de o escribir a direcciones devueltas por `ioremap()` (*pointer dereferencing*) puede no funcionar en algunas arquitecturas.
- Para hacer accesos estilo PCI/little-endian, con la conversión hecha automáticamente

```
unsigned read[bwl](void *addr);
void write[bwl](unsigned val, void *addr);
```

- Para accesos en bruto, sin conversión endianness

```
unsigned __raw_read[bwl](void *addr);
void __raw_write[bwl](unsigned val, void *addr);
```

- Ejemplo
  - escribir 32 bits

```
__raw_writel(1 << KS8695_IRQ_UART_TX,
    membase + KS8695_INTST);
```

Nueva API para accesos mezclados

- Una nueva API permite escribir drivers que pueden funcionar tanto en dispositivos accedidos por PIO como por MMIO. Unos cuantos drivers los usan, peor no parece haber consenso en la comunidad del kernel acerca de ello.
- Mapeado
  - Para PIO: `ioport_map()` y `ioport_unmap()`. Realmente no mapean, sino que devuelven una cookie especial iomem.
  - Para MMIO: `ioremap()` y `iounmap()`. Como siempre.
- Acceso, funciona tanto en direcciones como en cookies devueltas por `ioport_map()` y `ioremap()`
  - `ioread[8/16/32]()` y `iowrite[8/16/32]` para accesos individuales
  - `ioread[8/16/32]_rep()` y `iowrite[8/16/32]_rep()` para accesos repetidos

Evitar problemas con los accesos E/S

- La caché en puertos o memoria E/S ya está desactivada
- Usar las macros, hacen lo correcto para la arquitectura en uso
- El compilador y/o la CPU pueden reordenar los accesos a memoria, lo que puede dar lugar a problemas con los dispositivos si esperan que se lea/escriba un registro antes que otro.
  - Hay disponibles barreras de memoria para evitar esta reordenación
  - `rmb()` es una barrera de memoria de lectura, evita que las lecturas crucen la barrera
  - `wmb()` es una barrera de escritura
  - `mb()` es una barrera de lectura-escritura
- Empieza a ser un problema con CPUs que reordenan las instrucciones y SMP.
- Véase `Documentation/memory-barriers.txt` para más detalles

/dev/mem

- Usado para proporcionar a las aplicaciones del espacio de usuario acceso directo a las direcciones físicas.
- Uso: abrir `/dev/mem` y leer o escribir en un offset dado. Lo que se lee o escribe es el valor en la dirección física correspondiente.
- Usado por aplicaciones como el servidor X para escribir directamente en la memoria del dispositivo.
- En `x86`, `arm`, `arm64`, `tile`, `powerpc`, `unicore32`, `s390`: opción `CONFIG_STRICT_DEVMEM` para restringir direcciones fuera de RAM en `/dev/mem`, por razones de seguridad (Linux 3.10).

Frameworks del kernel para drivers de dispositivo

Laboratorio de Sistemas Inteligentes

Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

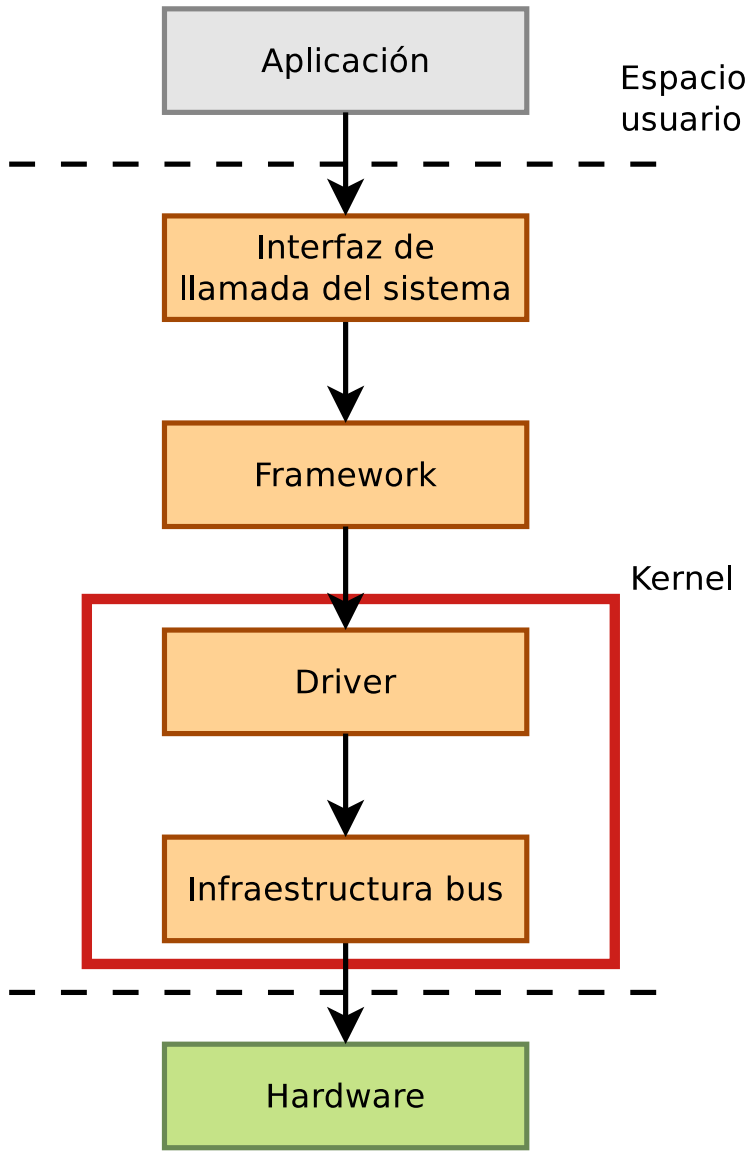
Última actualización: July 10, 2014.

Kernel and Drivers de Dispositivo

En Linux, un driver siempre tiene interfaz con:

- un **framework** que permite al driver exponer las características hardware de una forma genérica.
- una **infraestructura de bus**, parte del modelo de dispositivo, para detectar/comunicarse con el hardware.

Esta sección se centra en los *frameworks del kernel*, mientras que el *modelo de dispositivo* fue tratado anteriormente en este curso.



Visión de los dispositivos desde el espacio de usuario

Tipos de dispositivos

Bajo Linux, hay esencialmente tres tipos de dispositivos:

- **Dispositivos de red**. Se representan como interfaces de red, visibles en el espacio de usuario usando `ifconfig`.
- **Dispositivos orientados a/de bloques - Block devices**. Se usan para proporcionar acceso a los dispositivos de almacenamiento en bruto (discos duros, llaves USB) desde las aplicaciones del espacio de usuario. Son visibles para las aplicaciones como *archivos de dispositivo* en `/dev`.
- **Dispositivos orientados a/de caracteres - Character devices**. Se usan para proporcionar acceso a los demás dispositivos (entrada, sonido, gráficos, serie, etc.) desde las aplicaciones del espacio de usuario. Son visibles también para las aplicaciones como *archivos de dispositivo* en `/dev`.

$\rightarrow$ La mayoría de dispositivos son *dispositivos de caracteres*, por lo que estudiaremos estos con más detalle.

Números mayor y menor - Major/minor numbers

- Dentro del kernel, todos los dispositivos de bloques o caracteres se identifican usando un número *mayor* y un número *menor*.
- El *número mayor* normalmente indica la familia del dispositivo.
- El *número menor* normalmente indica el número del dispositivo (cuando son, por ejemplo, varios puertos serie)
- La mayoría de números mayores y menores se reservan estáticamente, y son idénticos en todos los sistemas Linux.
- Se definen en `Documentation/devices.txt` .

Dispositivos: todo es un archivo

- Una decisión de diseño muy importante de Unix fue representar la mayoría de "objetos del sistema" como archivos
- Permite a las aplicaciones manipular todos los "objetos del sistema con la API normal de archivos (`open`, `read`, `write`, `close`, etc.)
- Por tanto, los dispositivos tienen que representarse como archivos para las aplicaciones
- Esto se hace a través de un artefacto especial llamado **archivo de dispositivo** (**device file**)
- Es un tipo especial de archivo, que asocia un nombre de fichero visible a las aplicaciones del espacio de usuario a la tripleta *(tipo, mayor, menor)* que entiende el kernel
- Todos los *archivos de dispositivo* se almacenan por convención en el directorio `/dev`

Ejemplos de archivos de dispositivo

Ejemplo de archivos de dipositivo en un sistema Linux

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda1 /dev/sda2 /dev/zero
brw-rw---- 1 root disk     8,  1 2011-05-27 08:56 /dev/sda1
brw-rw---- 1 root disk     8,  2 2011-05-27 08:56 /dev/sda2
crw------- 1 root root     4,  1 2011-05-27 08:57 /dev/tty1
crw-rw---- 1 root dialout 4, 64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root     1,  5 2011-05-27 08:56 /dev/zero
```

Ejemplo de código C que usa la API de archivos normal para escribir datos en un puerto serie

```
int fd;
fd = open("/dev/ttyS0", O_RDWR);
write(fd, "Hello", 5);
close(fd);
```

Crear archivos de dispositivo

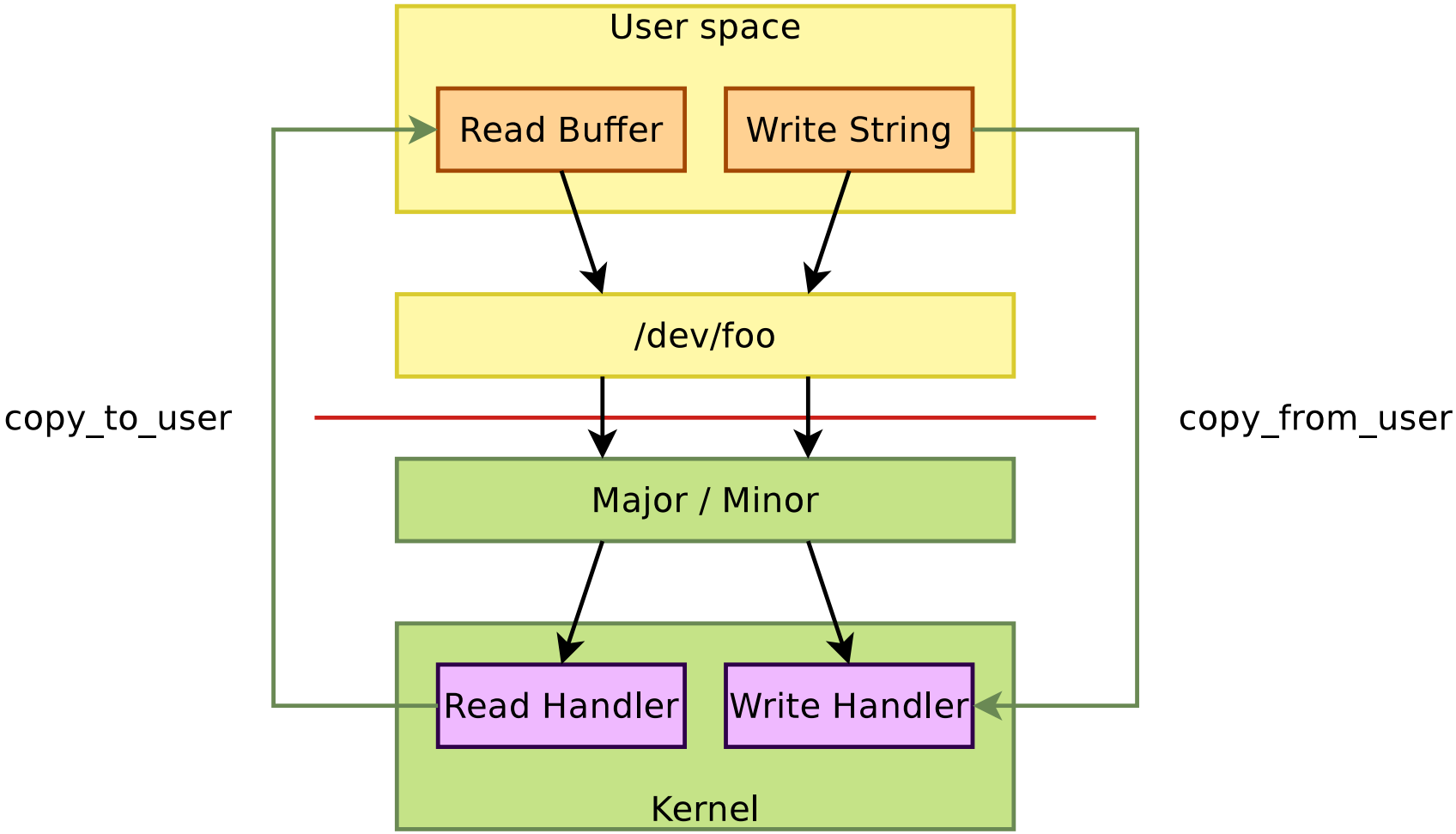
- En un sistema Linux básico, los archivos de dispositivo tienen que crearse manualmente usando el comando `mknod`
  - `mknod /dev/<device> [c|b] major minor`
  - Necesita privilegios root
  - La coherencia entre los archivos de dispositivo y los dispositivos manejados por el kernel es tarea del desarrollador del sistema
- En los sistemas Linux más elaborados, se pueden añador mecanismos para crearlos/eliminarlos automaticamente cuando aparecen y desaparecen los dispositivos
  - Sistema de ficheros virtual `devtmpfs`
  - Demonio `udev`, solución que se usa en los sistemas Linux de escritorio y servidores.
  - Programa `mdev`, una solución más ligera que `udev`

Drivers de caracteres

Un driver de caracteres en el kernel

- Desde el punto de vista de una aplicación, un *dispositivo de caracteres* es esencialmente un **archivo**.
- El driver de un dispositivo de caracteres debe, por tanto, implementar **operaciones** que permitan a las aplicaciones pensar que el dispositivo es un archivo: `open`, `close`, `read`, `write`, etc.
- Para conseguir esto, un driver de caracteres debe implementar las operaciones que se describen en la estructura `struct file_operations` y registrarlas.
- La capa del sistema de ficheros de Linux se asegurará de que las operaciones del driver sean llamadas cuando una aplicación del espacio de usuario haga la correspondiente llamada del sistema.

Desde el espacio de usuario al kernel



Operaciones de archivos

- Aquí están las operaciones más importantes para un driver de caracteres. Todas son opcionales.

```
#include <linux/fs.h>

struct file_operations {
    ssize_t (*read) (struct file *, char __user *,
        size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
        size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int,
        unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
};
```

open() and release()

- `int foo_open(struct inode *i, struct file *f)`
  - Se llama cuando el espacio de usuario abre el archivo de dispositivo.
  - `struct inode` es una estructura que representa de forma única un archivo en el sistema (sea un archivo normal, un directorio, un enlace simbólico, un dipositivo de bloques o caracteres)
  - `struct file` es una estructura que se crea cada vez que se abre un archivo. Varias estructuras `file` pueden apuntar a la misma estructura estructura `inode`.
    - Contiene información como la posición actual, el modo de apertura, etc.
    - Tiene un puntero `void *private_data` que puede usarse libremente.
    - Un puntero a la estructura `file` se pasa a todas las demás operaciones.
- `int foo_release(struct inode *i, struct file *f)`
  - Se llama cuando el espacio de usuario cierra el fichero.

read()

- `ssize_t foo_read(struct file *f, char __user *buf, size_t sz, loff_t *off)`
  - Se llama cuando el espacio de usuario usa la llamada del sistema `read()` en el dispositivo.
  - Debe leer datos del dispositivo, escribir como mucho `sz` bytes en el buffer del espacio de usuario `buf`, y actualizar la posición actual en el `off` del fichero. `f` es un puntero a la misma estructura `file` que se pasó en la operación `open()`
  - Debe devolver el número de bytes leídos
  - En UNIX, las operaciones `read()` típicamente bloquean cuando no hay suficientes datos para leer del dispositivo

write()

- `ssize_t foo_write(struct file *f,`

  `const char __user *buf, size_t sz, loff_t *off)`

  - Se llama cuando el espacio de usuario usa la llamada del sistema `write()` en el dispositivo
  - Lo contrario de `read`, debe leer como mucho `sz` bytes de `buf`, escribirlos al dispositivo, actualizar `off` y devolver el número de bytes escrito.

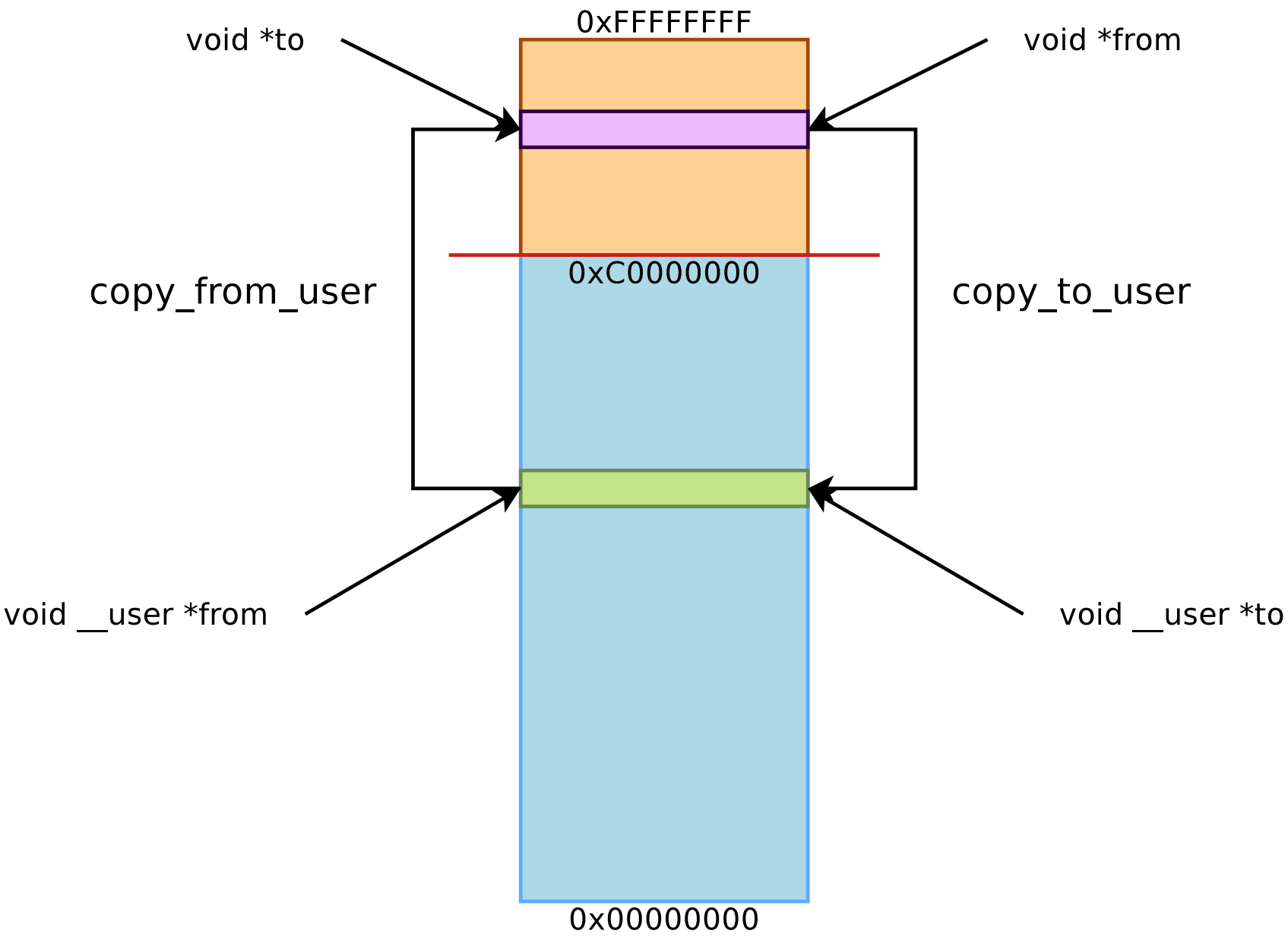
Intercambiar datos con el espacio de usuario 1/3

- El código del kernel no puede acceder directamente a la memoria del espacio de usuario, usando `memcpy()` o mediante acceso directo a los datos señalados por un puntero
  - Hacer esto no funciona en algunas arquitecturas
  - Si la dirección pasada por la aplicación es inválida, dará segfault.
- Para mantener el código del kernel portable y tener un manejo de error adecuado, el driver debe usar funciones especiales del kernel para intercambiar datos con el espacio de usuario.

Intercambiar datos con el espacio de usuario 2/3

- Un único valor
  - `get_user(v, p);`
    - La variable del kernel `v` obtiene el valor al que apunta el puntero del espacio de usuario `p`
  - `put_user(v, p);`
    - El valor al que apunta el puntero del espacio de usuario `p` se rellena con los contenidos de la variable del kernel `v`.
- Un buffer
  - ```
    unsigned long copy_to_user(void __user *to,
    const void *from, unsigned long n);
    ```
  - ```
    unsigned long copy_from_user(void *to,
    const void __user *from, unsigned long n);
    ```
- El valor de retorno debe comprobarse. Cero en caso de éxito, distinto de cero en caso de error. Si es distinto de cero, la convención es devolver `-EFAULT`.

Intercambiar datos con el espacio de usuario 3/3



Acceso sin copia a la memoria de usuario

- Tener que copiar datos a o desde un buffer intermedio del kernel puede volverse costoso cuando la cantidad de datos a transferir es grande (vídeo).
- Son posibles opciones sin copia (*Zero copy*):
  - La llamada del sistema `mmap()` permite al espacio de usuario acceder directamente al espacio de memoria E/S mapeado.
  - `get_user_pages()` para obtener un mapeado de las páginas de usuario sin tener que copiarlas. Esta API es más difícil de usar.

unlocked_ioctl()

- `long unlocked_ioctl(struct file *f, unsigned int cmd, unsigned long arg)`
  - Asociado a la llamada del sistema `ioctl()`.
  - Permite extender las capacidades del driver más allá de la limitada API de lectura/escritura.
  - Por ejemplo: cambiar la velocidad de un puerto serie, especificar el formato de salida del vídeo, consultar un número de dispositivo serie...
  - `cmd` es un número que identifica la operación a realizar
  - `arg` es el argumento opcional que se pasa como tercer argumento de la llamada del sistema `ioctl()`. Puede ser un entero, una dirección, etc.
  - La semántica de `cmd` y `arg` es específica del driver.

Ejemplo de ioctl(): lado del kernel

```
static long phantom_ioctl(struct file *file, unsigned int cmd,
    unsigned long arg)
{
    struct phm_reg r;
    void __user *argp = (void __user *)arg;

    switch (cmd) {
    case PHN_SET_REG:
        if (copy_from_user(&r, argp, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    case PHN_GET_REG:
        if (copy_to_user(argp, &r, sizeof(r)))
            return -EFAULT;
        /* Do something */
        break;
    default:
        return -ENOTTY;
    }

    return 0; }
```

Extracto seleccionado de `drivers/misc/phantom.c`

Ejemplo de ioctl(): lado de la aplicación

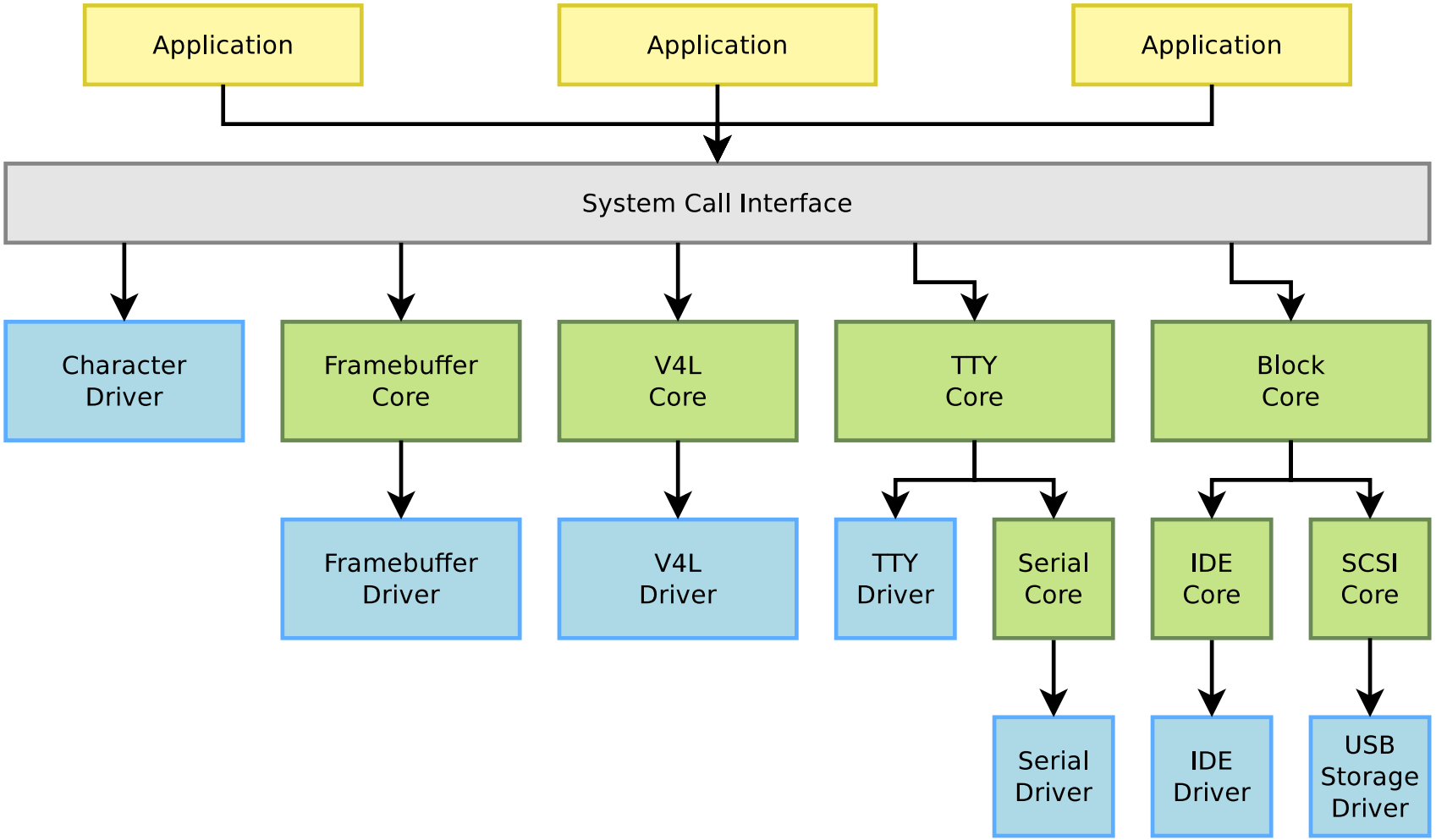
```
int main(void)
{
    int fd, ret;
    struct phm_reg reg;

    fd = open("/dev/phantom");
    assert(fd > 0);

    reg.field1 = 42;
    reg.field2 = 67;

    ret = ioctl(fd, PHN_SET_REG, & reg);
    assert(ret == 0);

    return 0;
}
```

El concepto de los frameworks del kernel

Más allá de los drivers de caracteres

- Muchos drivers de dispositivo no se implementan directamente como drivers de caracteres
- Se implementan bajo un *framework* específico a un tipo de dispositivo dado (framebuffer, V4L, serial, etc.)
  - El framework permite factorizar las partes comunes de los drivers para el mismo tipo de dispositivos
  - Desde el espacio de usuario, se siguen viendo como dispositivos de caracteres por las aplicaciones
  - El framework permite proporcionar una interfaz de usuario coherente (`ioctl`, etc.) para cada tipo de dispositivo, independientemente del driver

Frameworks del kernel



Ejemplo del framework framebuffer

Ejemplo: Framework Framebuffer

- Opción del kernel `CONFIG_FB`
  - `menuconfig FB`
    - `tristate "Support for frame buffer devices"`
- Implementado en `drivers/video/`
  - `fb.c`, `fbmem.c`, `fbmon.c`, `fbcmap.c`, `fbsysfs.c`, `modedb.c`, `fbcvt.c`
- Implementa un driver de caracteres único y define la API del usuario/kernel
  - Primera parte de `include/linux/fb.h`
- Define el conjunto de operaciones que un driver framebuffer debe implementar y las funciones de ayuda para los drivers
  - `struct fb_ops`
  - Segunda parte de `include/linux/fb.h` (en `ifdef __KERNEL__`)

Esqueleto de driver Framebuffer

- Esqueleto de driver en `drivers/video/skeletonfb.c`
- Implementa el conjunto de operaciones específicas framebuffer definidas por la estructura `struct fb_ops`

- `xxxfb_open()`
- `xxxfb_read()`
- `xxxfb_write()`
- `xxxfb_release()`
- `xxxfb_checkvar()`
- `xxxfb_setpar()`
- `xxxfb_setcolreg()`
- `xxxfb_blank()`
- `xxxfb_pan_display()`
- `xxxfb_fillrect()`
- `xxxfb_copyarea()`
- `xxxfb_imageblit()`
- `xxxfb_cursor()`
- `xxxfb_rotate()`
- `xxxfb_sync()`
- `xxxfb_ioctl()`
- `xxxfb_mmap()`

Esqueleto de driver Framebuffer

- Después de la implementación de las operaciones, definición de una estructura `struct fb_ops`

```
static struct fb_ops xxxfb_ops = {
    .owner = THIS_MODULE,
    .fb_open = xxxfb_open,
    .fb_read = xxxfb_read,
    .fb_write = xxxfb_write,
    .fb_release = xxxfb_release,
    .fb_check_var = xxxfb_check_var,
    .fb_set_par = xxxfb_set_par,
    .fb_setcolreg = xxxfb_setcolreg,
    .fb_blank = xxxfb_blank,
    .fb_pan_display = xxxfb_pan_display,
    .fb_fillrect = xxxfb_fillrect,   /* Necesario */
    .fb_copyarea = xxxfb_copyarea,   /* Necesario */
    .fb_imageblit = xxxfb_imageblit, /* Necesario */
    .fb_cursor = xxxfb_cursor,       /* Opcional */
    .fb_rotate = xxxfb_rotate,
    .fb_sync = xxxfb_sync,
    .fb_ioctl = xxxfb_ioctl,
    .fb_mmap = xxxfb_mmap,
};
```

Esqueleto de driver Framebuffer

- En la función `probe()`, registro del dispositivo y las operaciones framebuffer

```
static int xxxfb_probe (struct pci_dev *dev,
    const struct pci_device_id *ent)
{
    struct fb_info *info;
    [...]
    info = framebuffer_alloc(sizeof(struct xxx_par), device);
    [...]
    info->fbops = &xxxfb_ops;
    [...]
    if (register_framebuffer(info) > 0)
        return -EINVAL;
    [...]
}
```

- `register_framebuffer()` crea el dispositivo de caracteres que puede usarse desde las aplicaciones del espacio de usuario con la API genérica framebuffer.

Estructura de datos específica del driver

- Cada *framework* define una estructura que un driver de dispositivo debe registrar para ser reconocido como dispositivo en este framework
  - `struct uart_port` para el puerto serie, `struct netdev` para los dispositivos de red, `struct fb_info` para los framebuffers, etc.
- Además de esta estructura, el driver normalmente necesita almacenar información adicional sobre su dispositivo
- Esto se hace típicamente
  - Mediante una subclase de la estructura framework apropiada
  - Almacenando una referencia a la estructura framework apropiada
  - O incluyendo la información en la estructura framework

Ejemplos de datos específicos del driver 1/2

- Driver serie i.MX: `struct imx_port` es una subclase de `struct uart_port`

```
struct imx_port {
    struct uart_port port;
    struct timer_list timer;
    unsigned int old_status;
    int txirq, rxirq, rtsirq;
    unsigned int have_rtscts:1;
    [...]
};
```

- Driver RTC ds1305: `struct ds1305` tiene una referencia a `struct rtc_device`

```
struct ds1305 {
    struct spi_device       *spi;
    struct rtc_device       *rtc;
    [...]
};
```

Ejemplos de datos específicos del driver 2/2

- Driver de red rtl8150: `struct rtl8150` tiene una referencia a `struct net_device` y se reserva dentro de esa estructura framework.

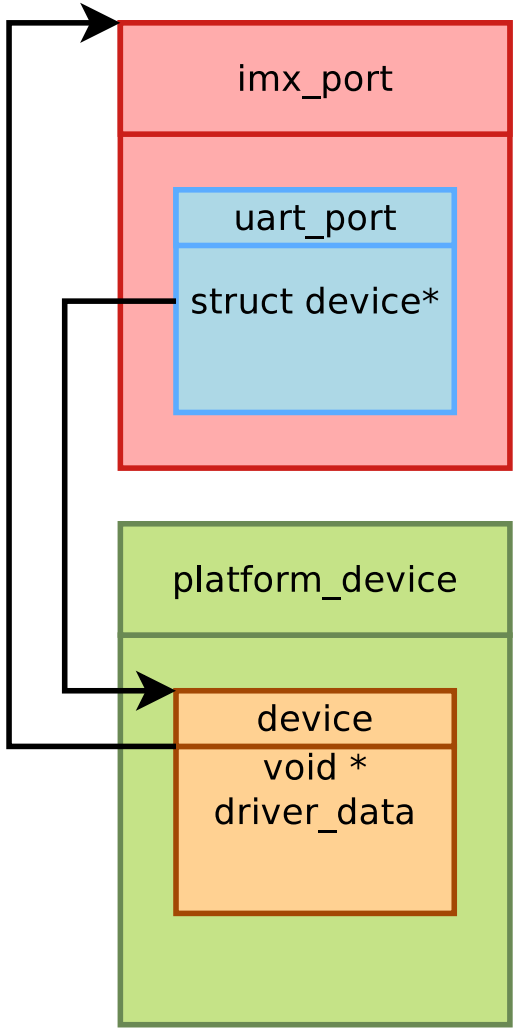
```
struct rtl8150 {
    unsigned long flags;
    struct usb_device *udev;
    struct tasklet_struct tl;
    struct net_device *netdev;
    [...]
};
```

Enlaces entre estructuras 1/4

- El framework suele contener un puntero `struct device *` que el driver debe apuntar a la correspondiente estructura de dispositivo
  - Es la relación entre el dispositivo lógico (por ejemplo una interfaz de red) y el dispositivo físico (por ejemplo el adaptador de red USB)
- La estructura de dispositivo también contiene un puntero `void *` que el driver puede usar libremente.
  - A menudo se usa para enlazar de vuelta el dispositivo a la estructura de alto nivel del framework.
  - Permite, por ejemplo, desde la estructura `struct platform_device`, encontrar la estructura que describe el dispositivo lógico

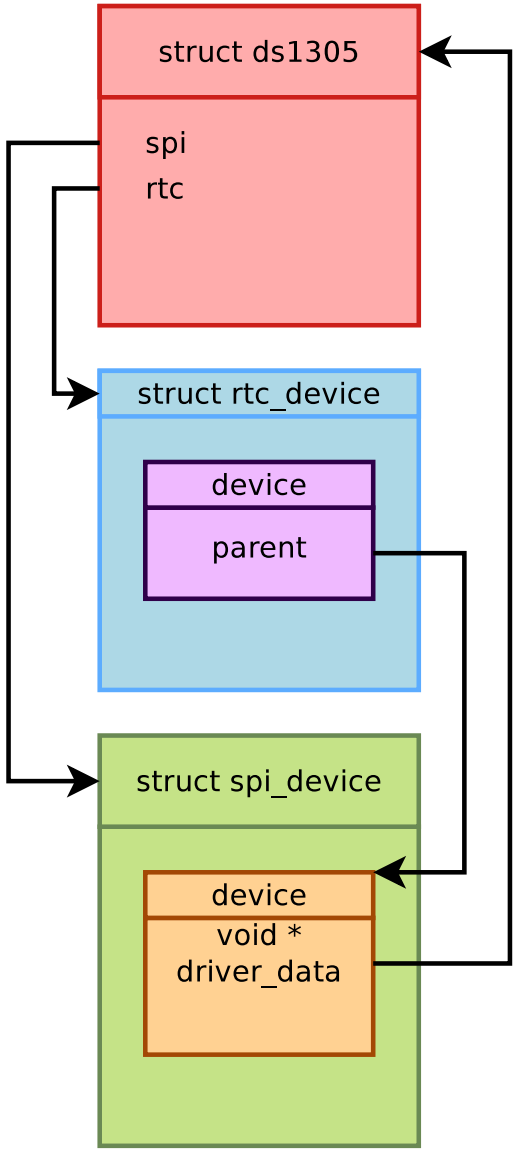
Enlaces entre estructuras 2/4

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport;
    [...]
    /* setup the link between uart_port and the struct
     * device inside the platform_device */
    sport->port.dev = &pdev->dev;
    [...]
    /* setup the link between the struct device inside
     * the platform device to the imx_port structure */
    platform_set_drvdata(pdev, &sport->port);
    [...]
    uart_add_one_port(&imx_reg, &sport->port);
}

static int serial_imx_remove(struct platform_device *pdev)
{
    /* retrieve the imx_port from the platform_device */
    struct imx_port *sport = platform_get_drvdata(pdev);
    [...]
    uart_remove_one_port(&imx_reg, &sport->port);
    [...]
}
```



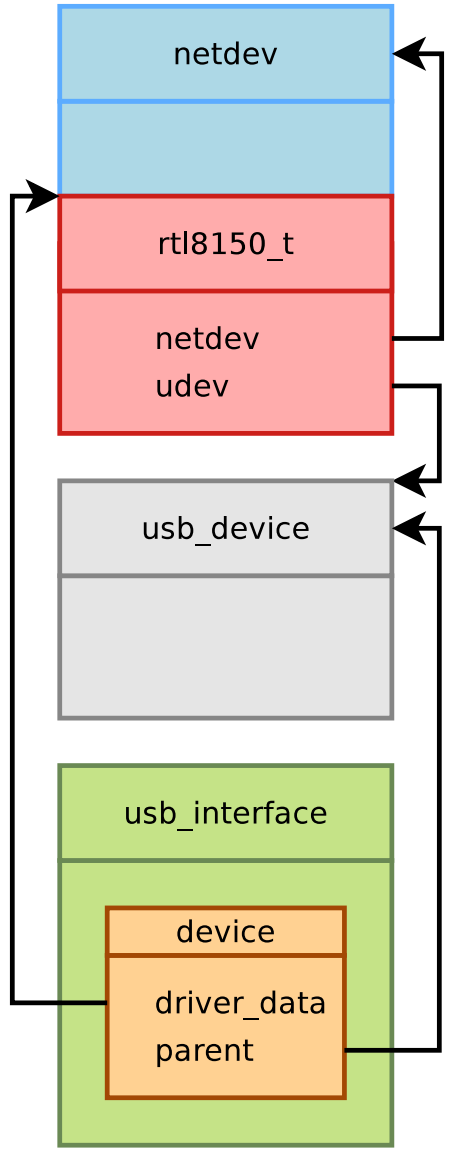
Enlaces entre estructuras 3/4

```
static int ds1305_probe(struct spi_device *spi)
{
    struct ds1305                   *ds1305;

    [...]

    /* set up driver data */
    ds1305 = devm_kzalloc(&spi->dev, sizeof(*ds1305), GFP_KERNEL);
    if (!ds1305)
            return -ENOMEM;
    ds1305->spi = spi;
    spi_set_drvdata(spi, ds1305);

    [...]

    /* register RTC ... from here on, ds1305->ctrl needs locking */
    ds1305->rtc = devm_rtc_device_register(&spi->dev, "ds1305",
                    &ds1305_ops, THIS_MODULE);

    [...]
}

static int ds1305_remove(struct spi_device *spi)
{
    struct ds1305 *ds1305 = spi_get_drvdata(spi);

    [...]
}
```



Enlaces entre estructuras 4/4

```
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    struct usb_device *udev = interface_to_usbdev(intf);
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    dev = netdev_priv(netdev);

    [...]

    dev->udev = udev;
    dev->netdev = netdev;

    [...]

    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);

    [...]
}

static void rtl8150_disconnect(struct usb_interface *intf)
{
    rtl8150_t *dev = usb_get_intfdata(intf);

    [...]
}
```



Laboratorio - Puertos y memoria E/S

- Añadir dispositivo led al ODROID
- Acceder a los registros E/S para controlar el dispositivo y enviarle instrucciones

El subsistema misc

Laboratorio de Sistemas Inteligentes

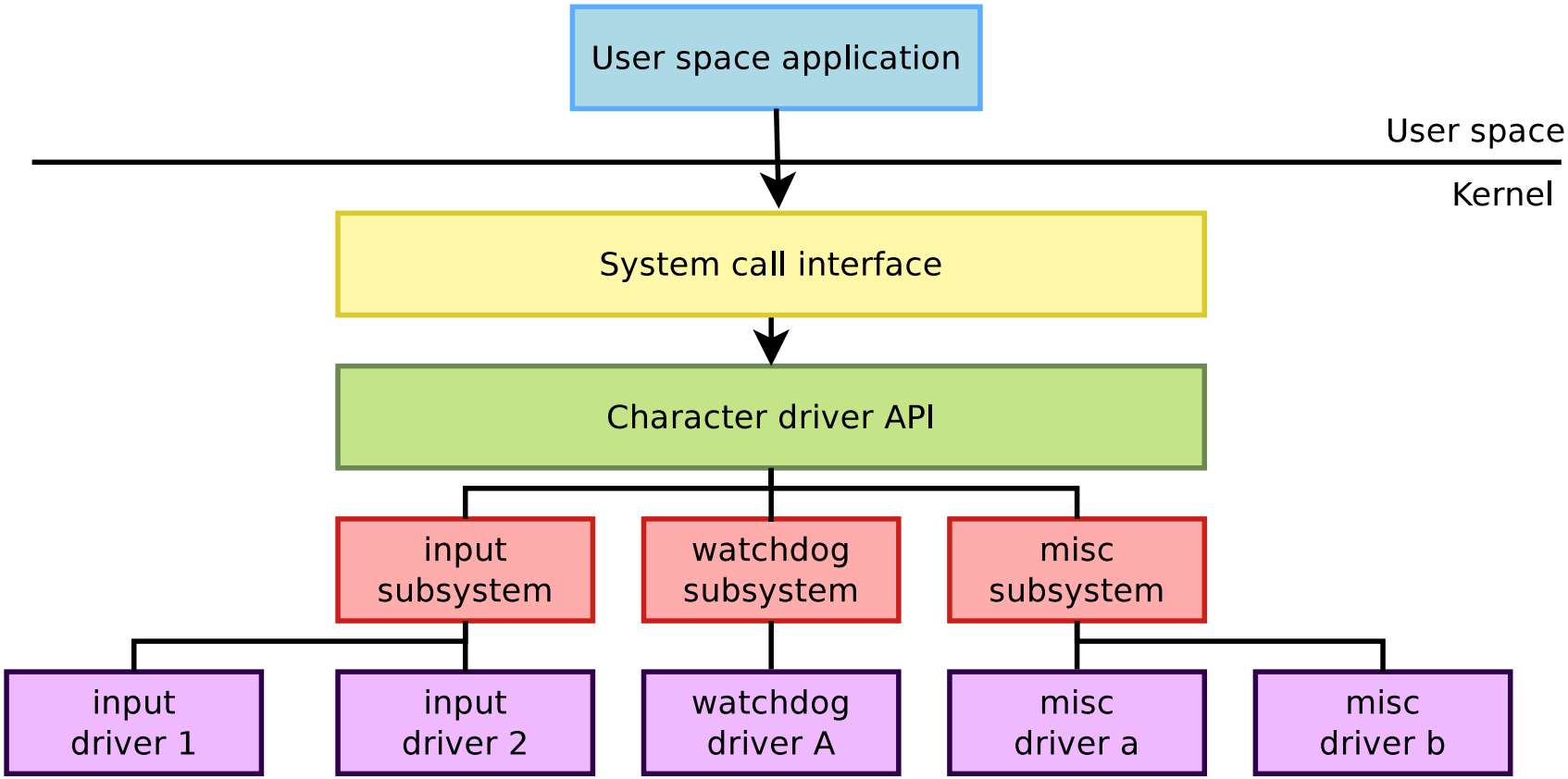
Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

¿Por qué un subsistema *misc*?

- El kernel ofrece un gran número de **frameworks** que cubren un amplio rango de tipos de dispositivo: entrada, red, vídeo, audio, etc.
  - Estos frameworks permiten factorizar funcionalidades comunes entre drivers y ofrecer una API consistente a las aplicaciones del espacio de trabajo.
- Sin embrago, hay algunos dispositivos que **realmente no se ajustan a ninguno de los frameworks existentes**.
  - Dispositivos altamente personalizados implementados en una FPGS, u otros dispositivos raros para los cuales no es útil implementar un framework completo.
- Los drivers para esos dispositivos pueden implementarse directamente como *drivers de caracteres* brutos.
- Pero hay un subsistema que hace este trabajo un poco más sencillo: el **subsistema misc**.
  - Realmente solo es una **fina capa** por encima de la API del *driver de caracteres*.

Diagrama del subsistema misc



API del subsistema misc (1/2)

- La API del subsistema misc principalmente proporciona dos funciones, registrar y desregistrar **un único** *dispositivo misc*:
  - `int misc_register(struct miscdevice * misc);`
  - `int misc_deregister(struct miscdevice *misc);`
- Un *dispositivo misc* se describe con una estructura `struct miscdevice`:

```
struct miscdevice  {
        int minor;
        const char *name;
        const struct file_operations *fops;
        struct list_head list;
        struct device *parent;
        struct device *this_device;
        const char *nodename;
        umode_t mode;
};
```

API del subsistema misc (2/2)

Los campos principales que hay que llenar en `struct miscdevice` son:

- `minor`, número menor para el dispositivo, o `MISC_DYNAMIC_MINOR` para obtener un número menor automáticamente asignado.
- `name`, nombre del dispositivo, que será usado para crear el nodo de dispositivo si se usa *devtmpfs*.
- `fops`, puntero a una estructura `struct file_operations`, que describe que funciones implementan las operaciones *read*, *write*, *ioctl*, etc.
- `parent`, la estructura `struct device` que representa el dispositivo hardware expuesto por este driver.

API del espacio de usuario para dispositivos misc

- Los *dispositivos misc* son dispositivos normales de caracteres
- Las operaciones que soportan en el espacio de usuario dependen de las operaciones que implementa el driver del kernel:
  - Las llamadas del sistema `open()` y `close()` para abrir/cerrar el dispositivo.
  - Las llamadas del sistema `read()` y `write()` para leer/escribir del/al dispositivo.
  - La llamada del sistema `ioctl()` para llamar a algunas operaciones específicas del driver.

Laboratorio - Driver led

- Extender el driver comenzado en el ejercicio anterior registrándolo en el subsistema *misc*.
- Implementar las funcionalidades de entrada / salida a través del subsistema *misc*.
- Probar el funcionamiento del driver.

El subsistema input

Laboratorio de Sistemas Inteligentes

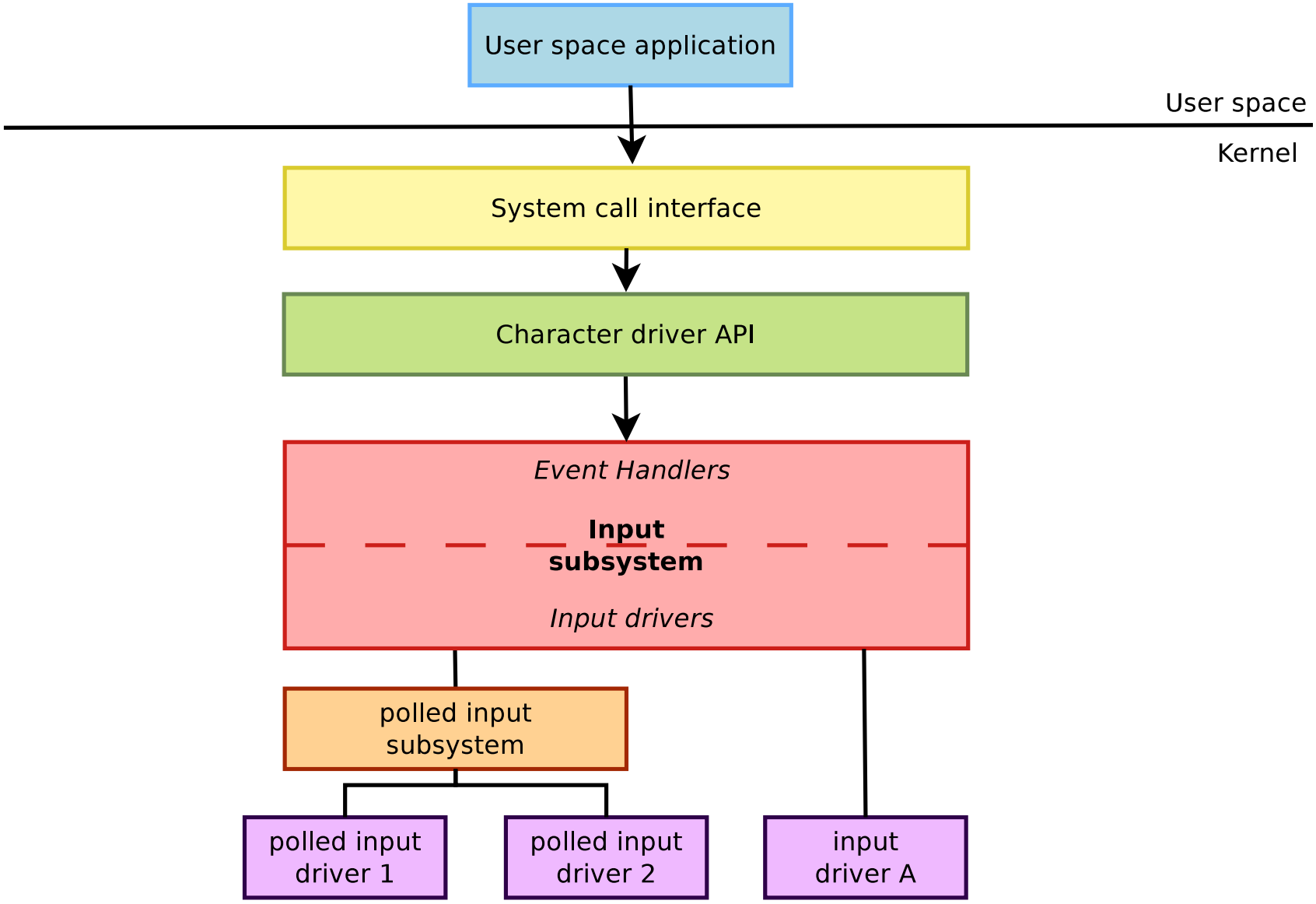
Basado en el trabajo de Free Electrons:
`http://free-electrons.com/docs/`

Última actualización: July 10, 2014.

¿Qué es el subsistema input?

- El subsistema input se encarga de todos los eventos de entrada que vienen del usuario humano.
- Inicialmente escrito para soportar los dispositivos USB *HID* (Human Interface Device), rápidamente creció para manejar todo tipo de entrada (usen USB o no): teclado, ratón, joysticks, pantallas táctiles, etc.
- El subsistema input se divide en dos partes:
  - **Drivers de dispositivos**: hablan al hardware (por ejemplo vía USB) y proporcionan eventos (pulsaciones de teclas, movimientos de ratón, coordenadas en las pantallas táctiles) al núcleo input
  - **Manejadores de eventos**: reciben eventos de los drivers y los pasan adonde sea necesario a través de varias interfaces (la mayoría de las veces a través de `evdev`)
- En el espacio de usuario, habitualmente lo usa el paquete gráfico, como *X.Org*, *Wayland* o *Android InputManager*.

Diagrama del subsistema Input



Vista general del subsistema Input

- Opción del kernel `CONFIG_INPUT`
  - `menuconfig INPUT`
    - `tristate "Generic input layer (needed for keyboard, mouse, ...)"`
- Implementado en `drivers/input/`
  - `input.c`, `input-polldev.c`, `evbug.c`
- Implementa un driver de caracteres único y define la API del usuario/kernel
  - `include/uapi/linux/input.h`
- Define el conjunto de operaciones que un driver de entrada debe implementar y funciones de ayuda para los drivers
  - `struct input_dev` para la parte del driver de dispositivo
  - `struct input_handler` para la parte del manejador de eventos
  - `include/linux/input.h`

API del subsistema Input 1/3

- Un *dispositivo input* se describe con una estructura muy larga `struct input_dev`, un extracto es:

```
struct input_dev {
    const char *name;
    [...]
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
    [...]
    int (*getkeycode)(struct input_dev *dev,
                      struct input_keymap_entry *ke);
    [...]
    int (*open)(struct input_dev *dev);
    [...]
    int (*event)(struct input_dev *dev, unsigned int type,
                 unsigned int code, int value);
    [...]
};
```

- Antes de ser usada, esta estructura tiene que ser reservada en memoria e inicializada:
  `struct input_dev *input_allocate_device(void);`
- Despues de desregistrar `struct input_dev`, debe ser liberada:
  `void input_free_device(struct input_dev *dev);`

API del subsistema Input 2/3

- Dependiendo del tipo de evento que se genere, los campos de bit de entrada `evbit` y `keybit` deben configurarse: Por ejemplo, para un botón, solo generamos eventos de tipo `EV_KEY`, y de estos, solo eventos `BTN_0`:

```
set_bit(EV_KEY, myinput_dev.evbit);
set_bit(BTN_0, myinput_dev.keybit);
```

- `set_bit()` es una operación atómica que permite poner un bit particular a `1`.
- Una vez el *dispositivo input* se localiza en memoria y se rellena, la función para registrarlo es:
  `int input_register_device(struct input_dev *);`
- Cuando se descargue el driver, el *dispositivo input* se desregistra usando:
  `void input_unregister_device(struct input_dev *);`

API del subsistema Input 3/3

- Los eventos se envían por parte del driver al manejador de eventos usando `input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);`
  - Los tipos de eventos están documentados en `Documentation/input/event-codes.txt`
  - Un evento se compone de uno o varios cambios de los datos de entrada (cambios de paquetes o de datos de entrada) como el estado del botón, la posición relativa o absoluta a lo largo de un eje, etc.
  - Después de entregar eventos potencialmente múltiples, el núcleo *input* debe ser notificado llamando a: `void input_sync(struct input_dev *dev)`:
  - El subsistema input proporciona otras envolturas como `input_report_key()`, `input_report_abs()`, ...

Subclase Polled input

- El subsistema input proporciona una subclase que soporta dispositivos simples de entrada que *no generan interrupciones* sino que tienen que ser periódicamente consultados (*polled*) para detectar cambios en su estado.
- Un dispositivo *polled input* se describe mediante una estructura `struct input_polled_dev`:

```
struct input_polled_dev {
        void *private;
        void (*open)(struct input_polled_dev *dev);
        void (*close)(struct input_polled_dev *dev);
        void (*poll)(struct input_polled_dev *dev);
        unsigned int poll_interval; /* msec */
        unsigned int poll_interval_max; /* msec */
        unsigned int poll_interval_min; /* msec */
        struct input_dev *input;
/* private: */
        struct delayed_work work;
}
```

API del subsistema polled input

- La reserva/liberación de la estructura `struct input_polled_dev` se hace usando
  - `input_allocate_polled_device()`
  - `input_free_polled_device()`
- Entre los manejadores de `struct input_polled_dev` solo el método `poll()` es obligatorio, esta función consulta el dispositivo y postea eventos de entrada.
- Los campos `id`, `name`, `phys`, `bits` de `struct input` también deben ser inicializados.
- Si no se rellena ninguno de los campos `poll_interval`, el intervalo de consulta por defecto es de 500ms.
- El registro/desregistro del dispositivo se hace con:
  - `input_register_polled_device(struct input_polled_dev *dev)`.
  - `input_unregister_polled_device(struct input_polled_dev *dev)`

Interfaz del espacio de usuario *evdev*

- La principal interfaz del espacio de usuario con los *dispositivos input* es la **interfaz de eventos**
- Cada *dispositivo input* se representa como un dispositivo de caracteres `/dev/input/event<X>`
- Una aplicación del espacio de usuario puede usar lecturas bloqueantes y no bloqueantes, pero también `select()` (para ser notificado de los eventos) después de abrir el dispositivo.
- Cada lectura devolverá estructuras `struct input_event` del siguiente formato:

```
struct input_event {
        struct timeval time;
        unsigned short type;
        unsigned short code;
        unsigned int value;
};
```

- Una aplicación muy útil para comprobar el funcionamiento de un *dispositivo input* es `evtest`, `http://cgit.freedesktop.org/evtest/`