S. empotrados y ubicuos

Programación de dispositivos

Fernando Pérez Costoya
Despacho 4201
fperez@fi.upm.es

Contenido

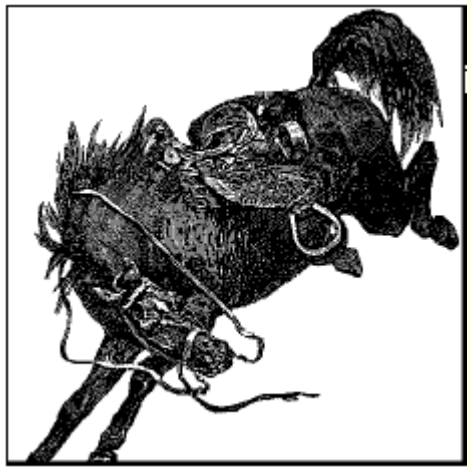
- **Introducción**
- El hardware de E/S visto desde el software
- Aspectos generales de la programación de dispositivos
- Programación de manejadores de dispositivos
  - Caso práctico: programación de manejadores en Linux

Introducción

- Computador incluye dispositivos de E/S para:
  - Almacenamiento de información
  - Interacción con mundo exterior
    - Usuarios, componentes físicos (sensores, actuadores,…)
  - Comunicación con otros equipos
- Software de E/S muy complejo y heterogéneo
- Precisamente por eso surgieron los SS.OO.
  - Ofrecen interfaz uniforme para todos los dispositivos
  - Manejadores (*drivers*) ocultan complejidad y heterogeneidad
- Programación de manejadores infrecuente en sist. propósito gral.

Introducción

- Sistema empotrado controla un sistema externo
  - Gran interacción con componentes físicos
  - Con frecuencia mediante hardware *ad hoc*
  - Necesidad de programar este hardware
- Menos habitual desarrollo de manejadores de E/S para:
  - Almacenamiento, interacción con usuarios o comunicación
- Progr. de dispositivos muy compleja
  - Hay que domar a "la bestia"



***Linux Device Drivers*, 3ª Edición. O'Reilly, 2005**

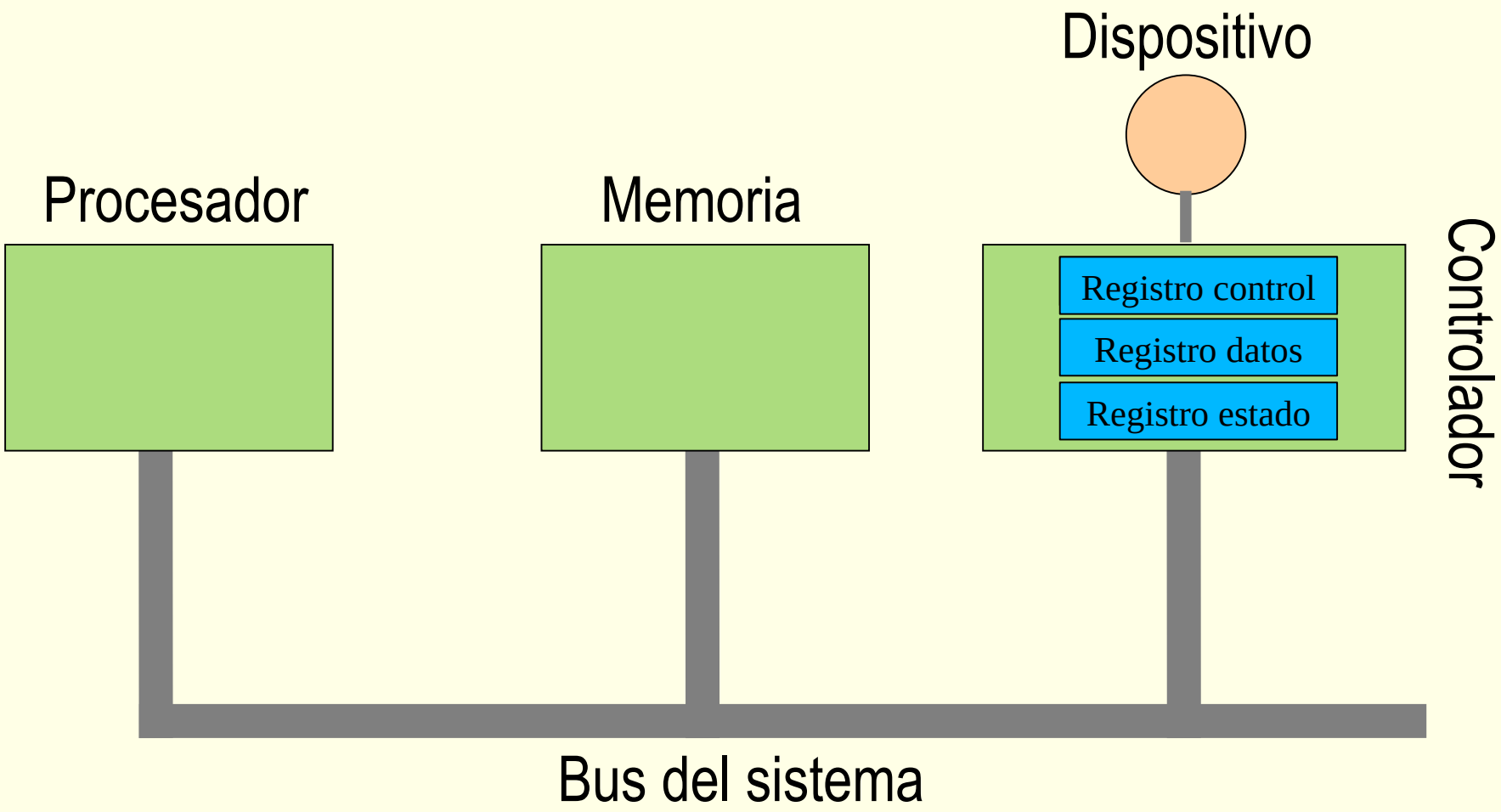
Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

https://lwn.net/Kernel/LDD3/

Contenido

- Introducción
- **El hardware de E/S visto desde el software**
- Aspectos generales de la programación de dispositivos
- Programación de manejadores de dispositivos
  - Caso práctico: programación de manejadores en Linux

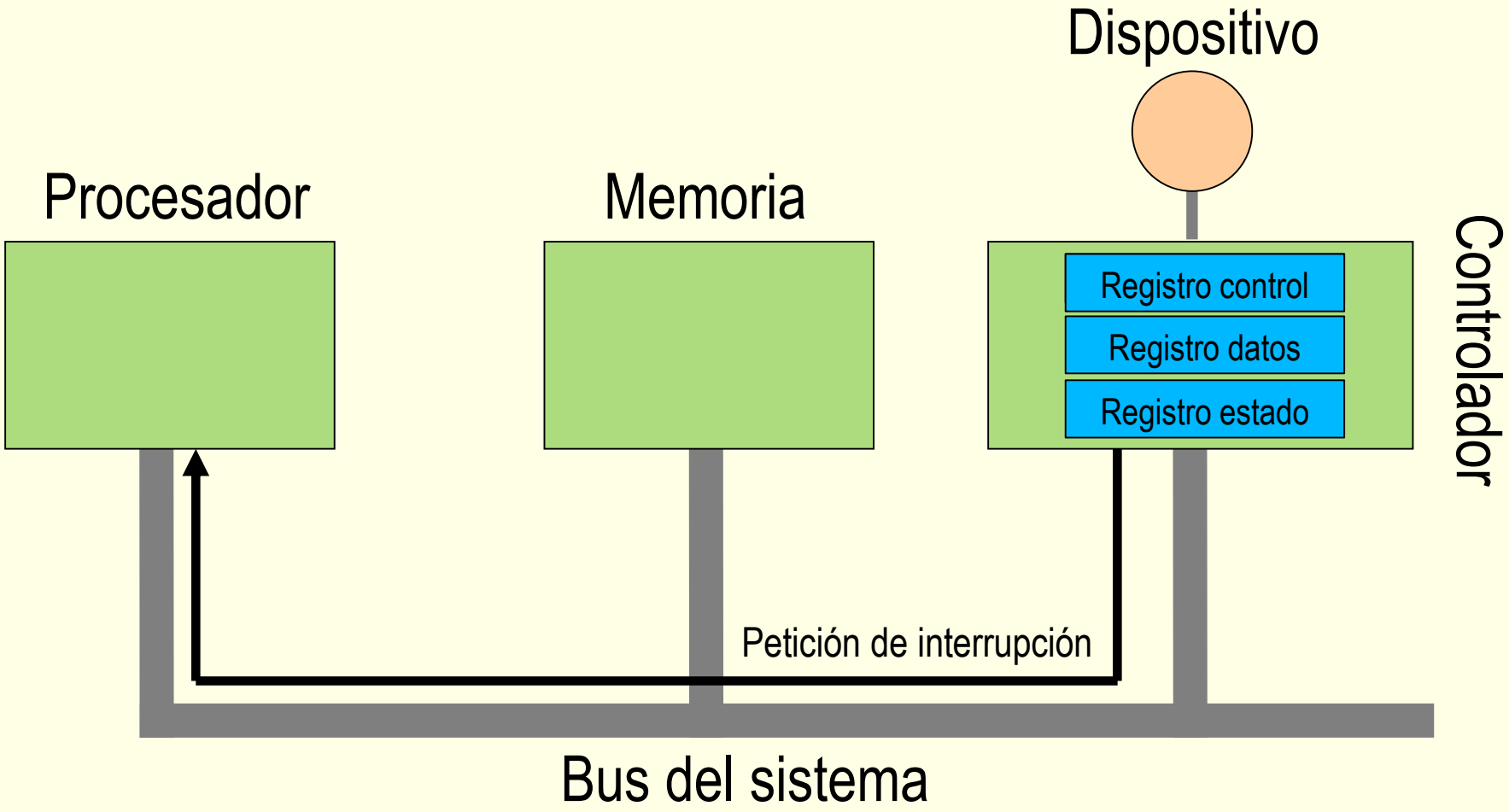
Modelo simplificado de dispositivo básico



E/S programada

- Esquema básico de operación lectura de N datos. Repetir N veces:
  1. Escritura en **reg. control** de código de operación de lectura
  2. Lectura repetida de **reg. estado** hasta fin de operación o error
  3. Si fin de operación → lectura de **reg. datos** del dato leído
- HW sencillo pero UCP monopolizada por la operación
- Aceptable solo en sistema dedicado
- En otros sistemas, UCP debe ocuparse también de otras labores
- Posible opción: lectura periódica de r. estado. Repetir N veces:
  1. Escritura en **reg. control** de código de operación de lectura
  2. Activa temporizador y pasa a otras labores
  3. Se cumple temporizador: Lectura de **reg. estado**
  4. Si fin operación → lectura **r. datos** del dato leído y vuelve a 1
  5. Si no → activa temporizador y pasa a otras labores
- Mejor opción: uso de interrupciones

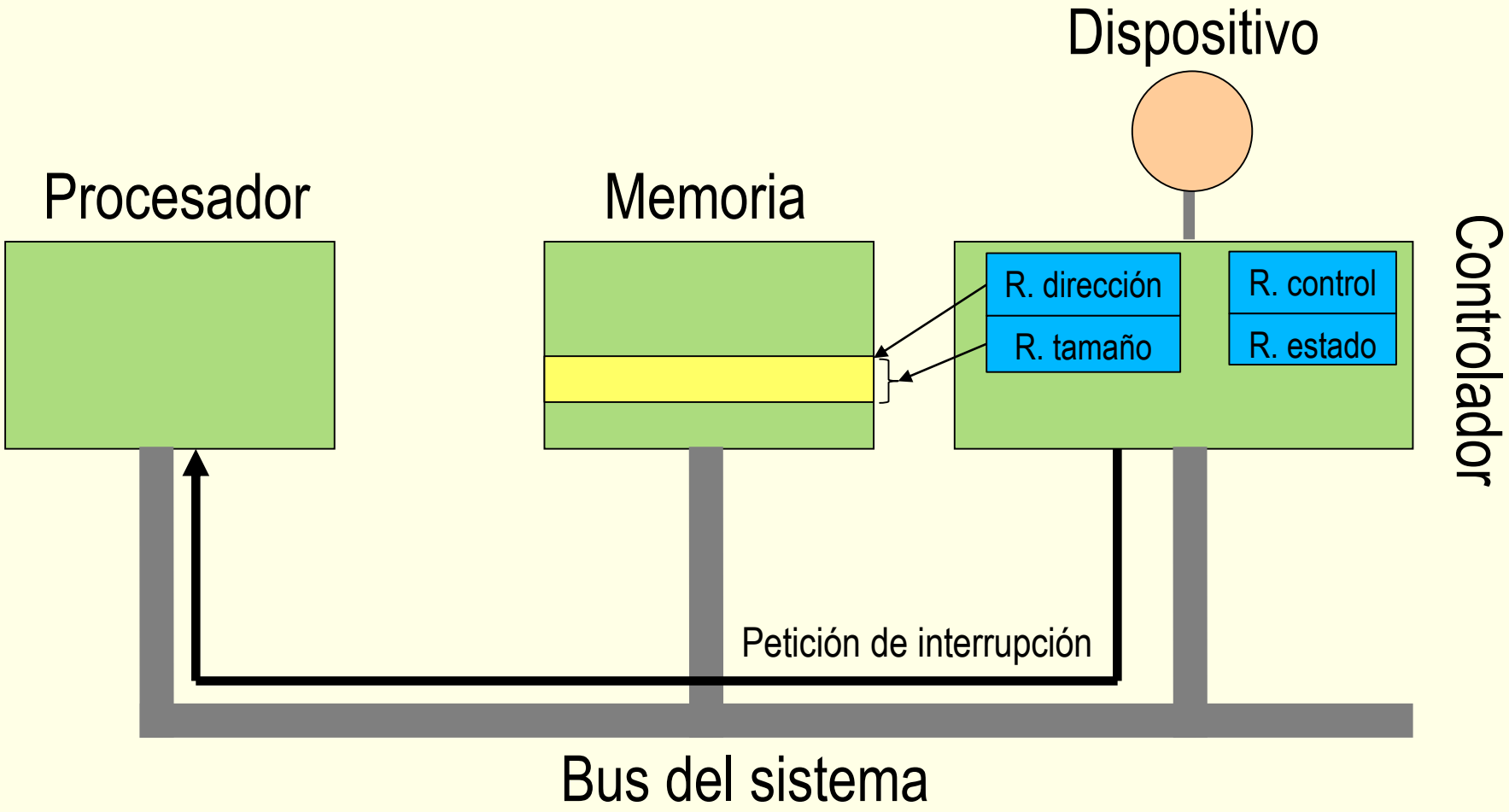
Modelo simplificado de dispo. con interrupciones



E/S con interrupciones

- Controlador de dispo. genera interrupción al completar operación
  - Proporciona un ID (vector) cuando UCP reconoce interrupción
  - Se activa rutina de interrupción correspondiente a vector
- HW más complejo pero solución más eficiente
  - UCP involucrada solo cuando es necesario
- Esquema básico de operación lectura de N datos. Repetir N veces:
  1. Escritura en **reg. control** de código de operación de lectura
  2. Pasa a otras labores
  3. Se produce interrupción: Lectura de **reg. estado**
  4. Si no error → lectura **reg. datos** del dato leído
  5. Vuelve a 1
- UCP involucrada en obtener cada dato
  - Problema si muchos datos y/o dispositivo de alta velocidad
- Mejor opción en ese caso: uso de DMA

Modelo simplificado de dispositivo con DMA



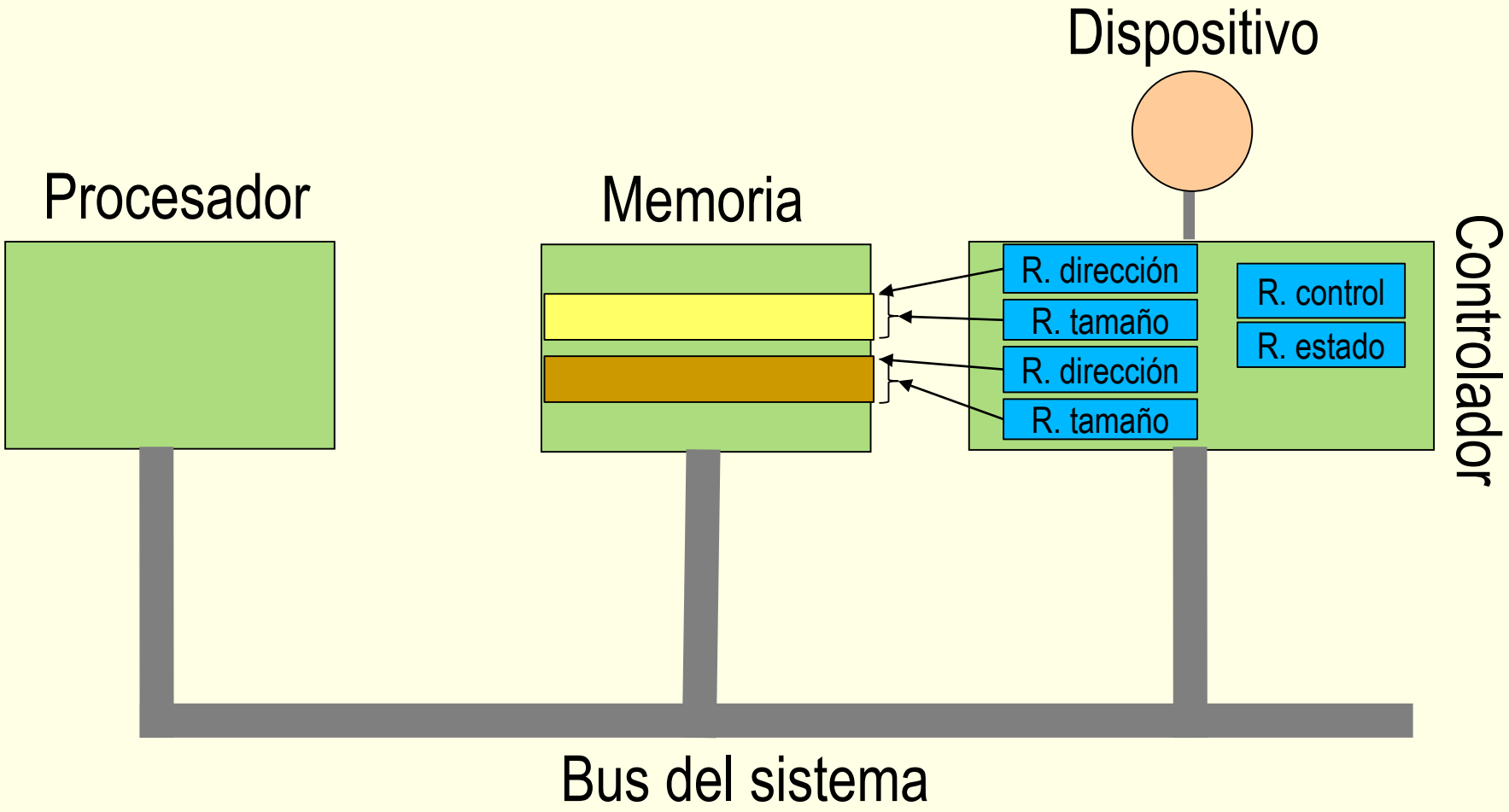
E/S con DMA

- Controlador puede copiar datos de dispo. a memoria y viceversa
  - Sin intervención de la UCP
  - Dispositivo envía interrupción al completar **toda** la operación
- HW más complejo pero solución más eficiente
  - UCP solo involucrada al inicio para programar OP de DMA
  - y al final para tratar la interrupción
- Esquema básico de operación lectura de N datos. Solo una vez:
  1. Escritura en **reg. tamaño** del valor N
  2. Escritura en **r. dirección** de dir. de mem. donde dejar los datos
  3. Escritura en **reg. control** de código de operación de lectura
  4. Pasa a otras labores
  5. Se produce interrupción: Operación total completada
     - Lectura de **reg. estado** para comprobar si ha habido un error

DMA: Aspectos adicionales

- Controlador con o sin *scatter-gather DMA (vectored I/O)*
  - Controlador con múltiples pares [reg. dirección; reg. tamaño]
  - Permite múltiples transferencias en una sola operación
- Controlador con o sin coherencia entre caché y memoria
  - *Non-coherent DMA*: transferencias DMA no afectan a la caché
  - Si *Non-coherent DMA*, posible uso incorrecto de datos:
    - Operación de lectura: programa obtiene datos obsoletos
    - Operación de escritura: Se envían a dispositivo datos obsoletos
    - Debe resolverse por software
- Uso de IO-MMU (*aka* virtual DMA)
  - Controlador ve memoria con direcciones ≠ UCP (p.e. SPARC)
  - Posibilita ver como contiguo buffer no contiguo en m. física.
    - Puede permitir múltiples transferencias en una sola operación
  - Facilita virtualización de la E/S

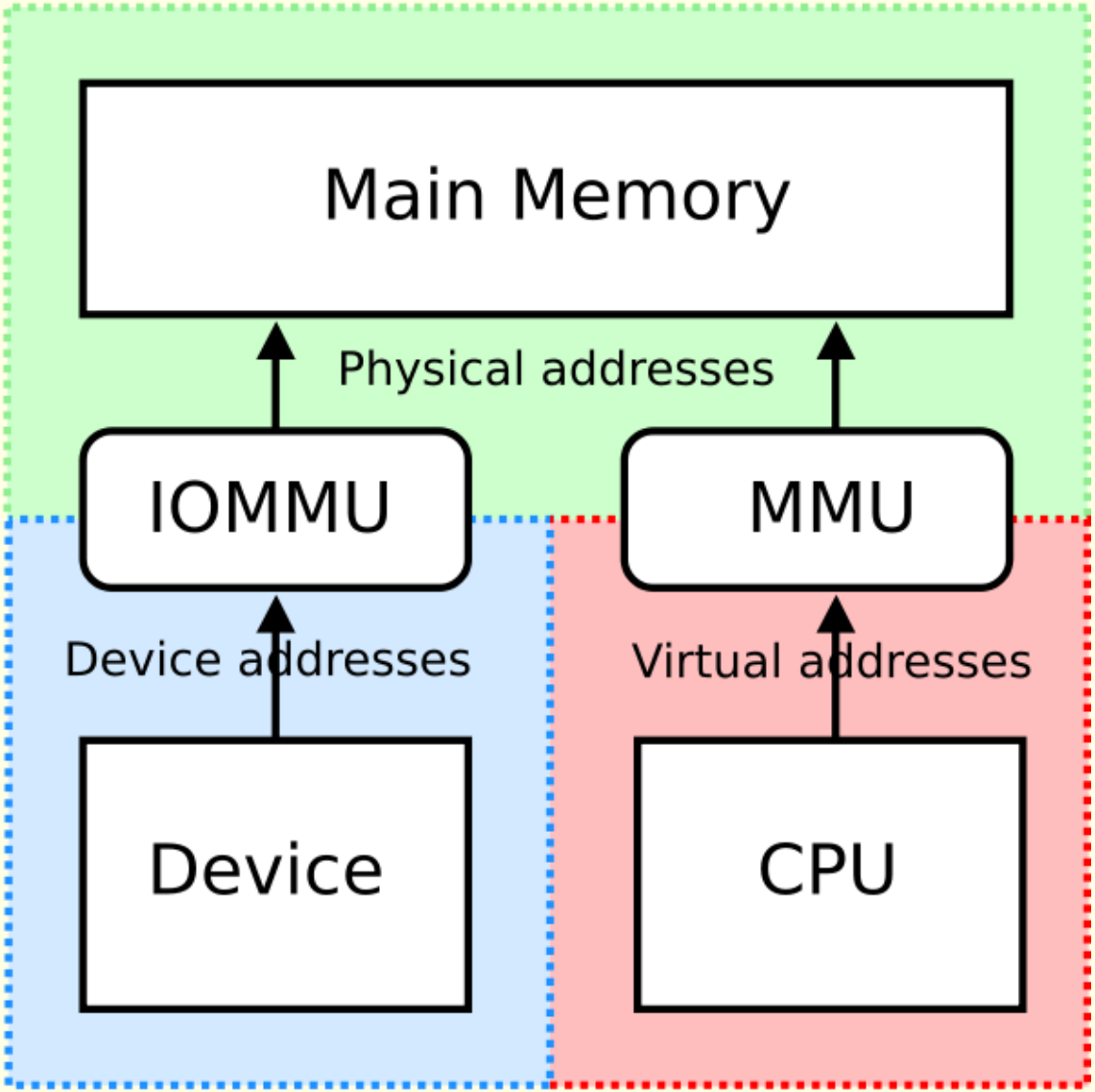
Controlador con *scatter-gather* DMA



Non-coherent DMA (wikipedia)

Operación de lectura

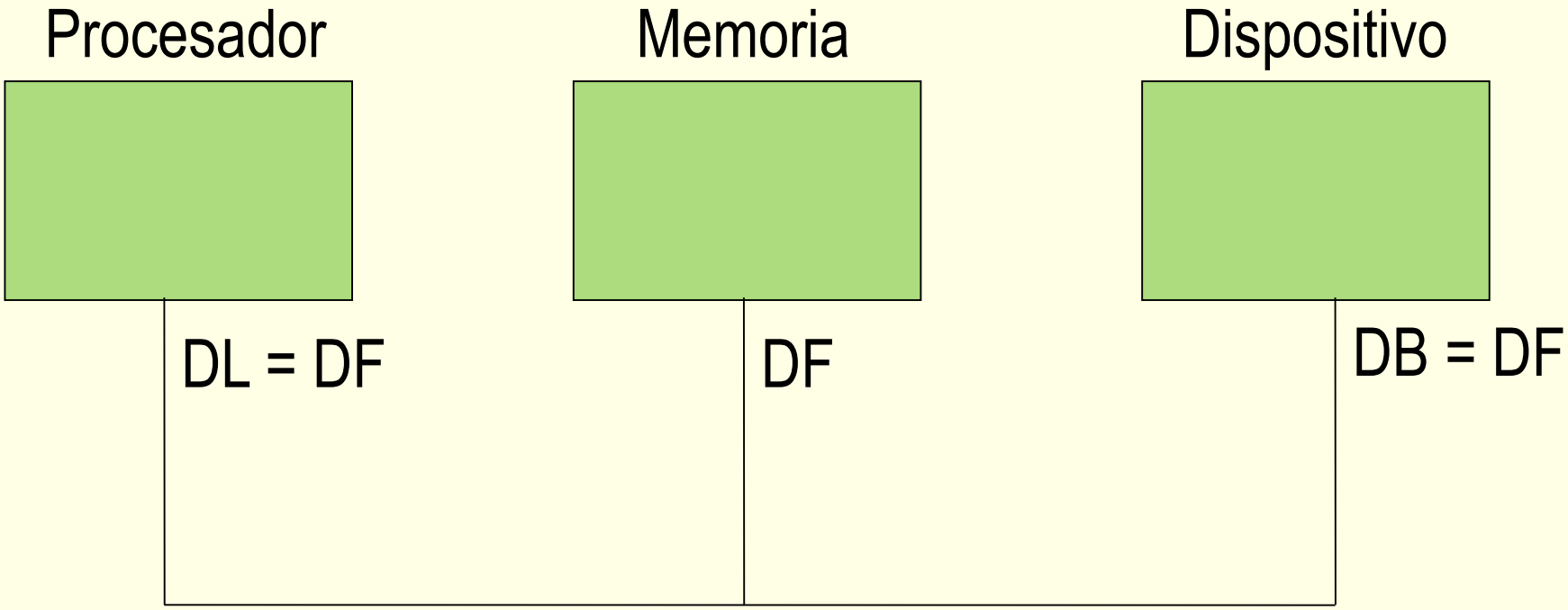
IO-MMU (wikipedia)



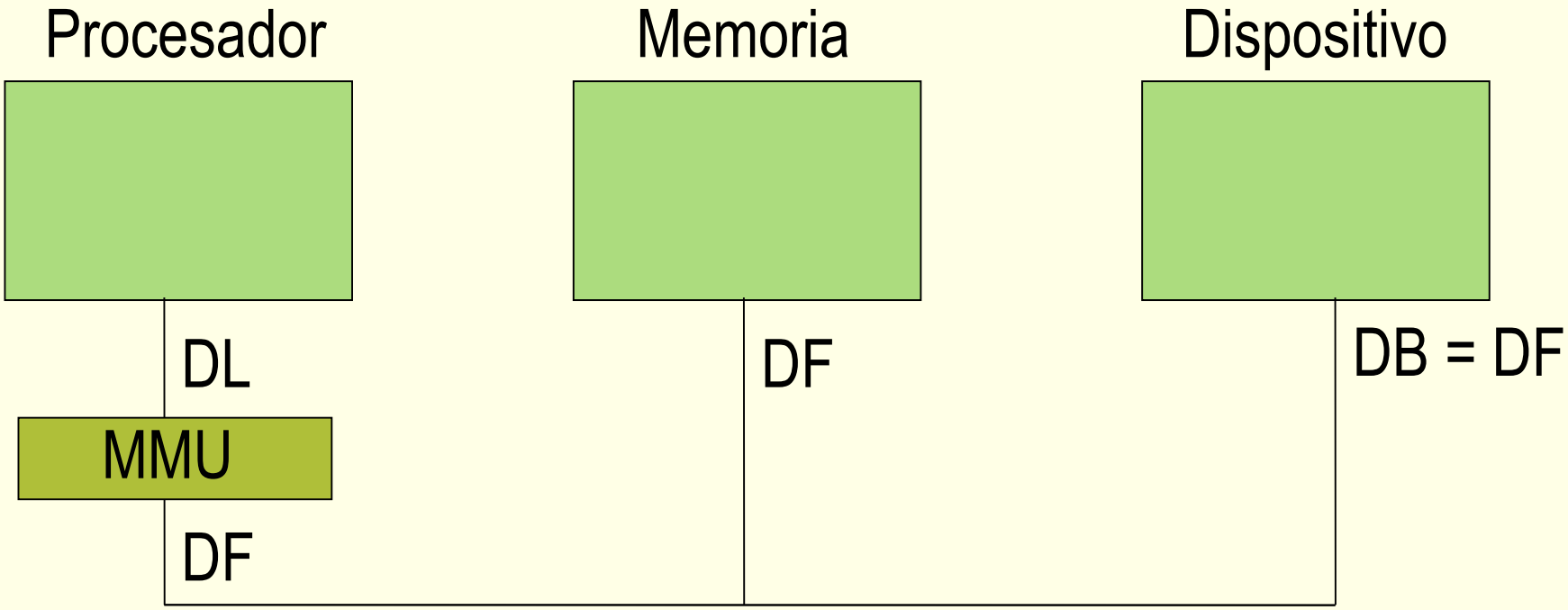
Tipos de direcciones

- Debido a la existencia de MMU e IO-MMU
- Tres tipos:
  - Lógicas (DL) (o virtual): usadas por procesador
    - Si UCP distingue modos de ejecución: DL usuario o DL sistema
  - Físicas (DF): las que llegan a la memoria
  - De bus (DB): usadas por un dispositivo
- En sistema sin MMU ni IO-MMU
  - DL = DF = DB
- En sistema con MMU pero sin IO-MMU
  - DL ≠ DF = DB
- En sistema con MMU e IO-MMU
  - DL ≠ DF ≠ DB
- ***https://static.lwn.net/images/pdf/LDD3/ch15.pdf***

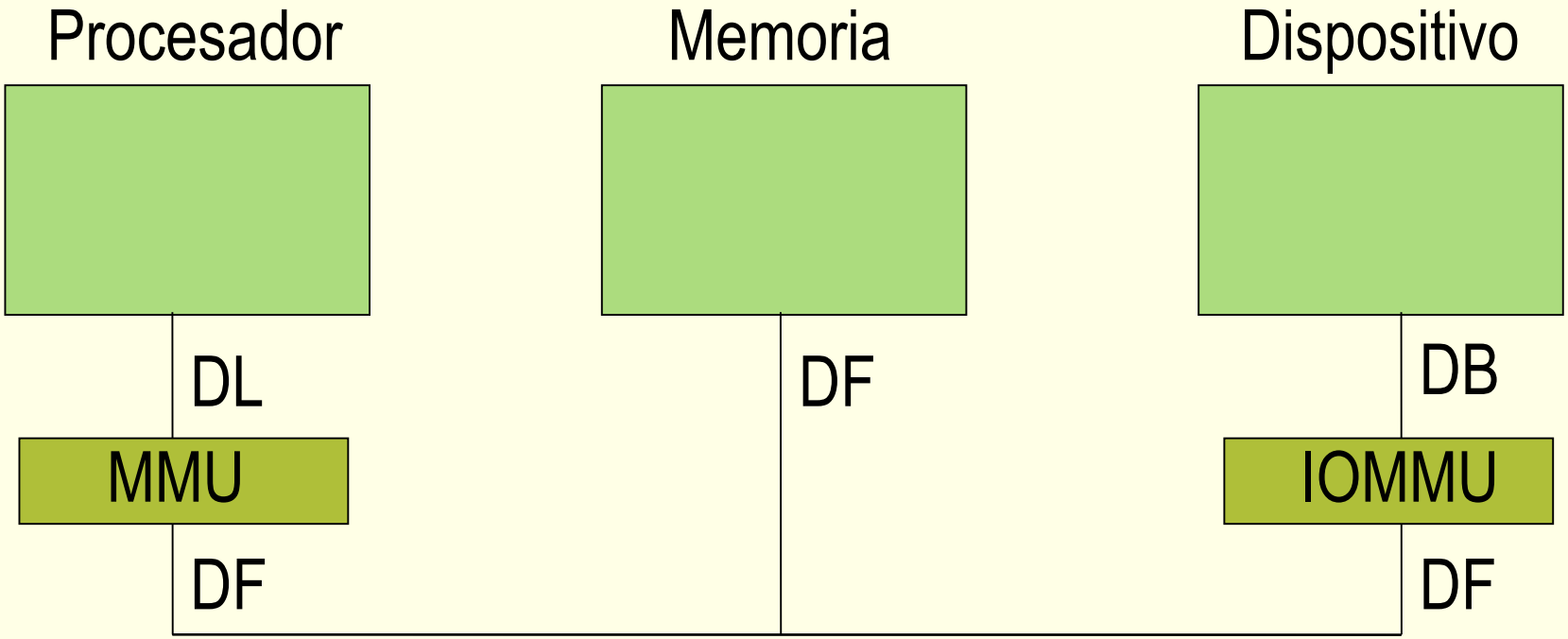
Sistema sin MMU ni IO-MMU



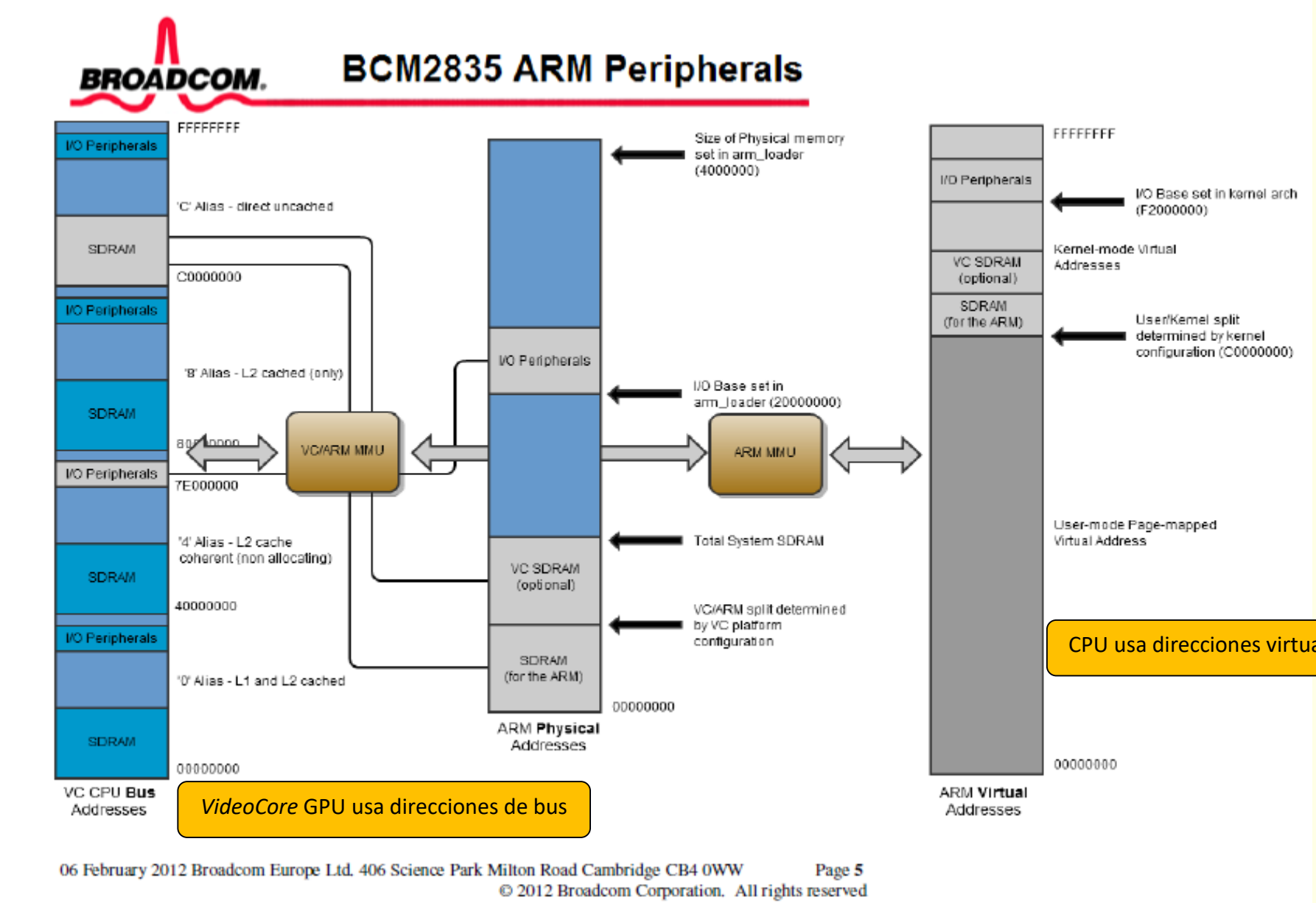
Sistema con MMU pero sin IO-MMU



Sistema con MMU e IO-MMU



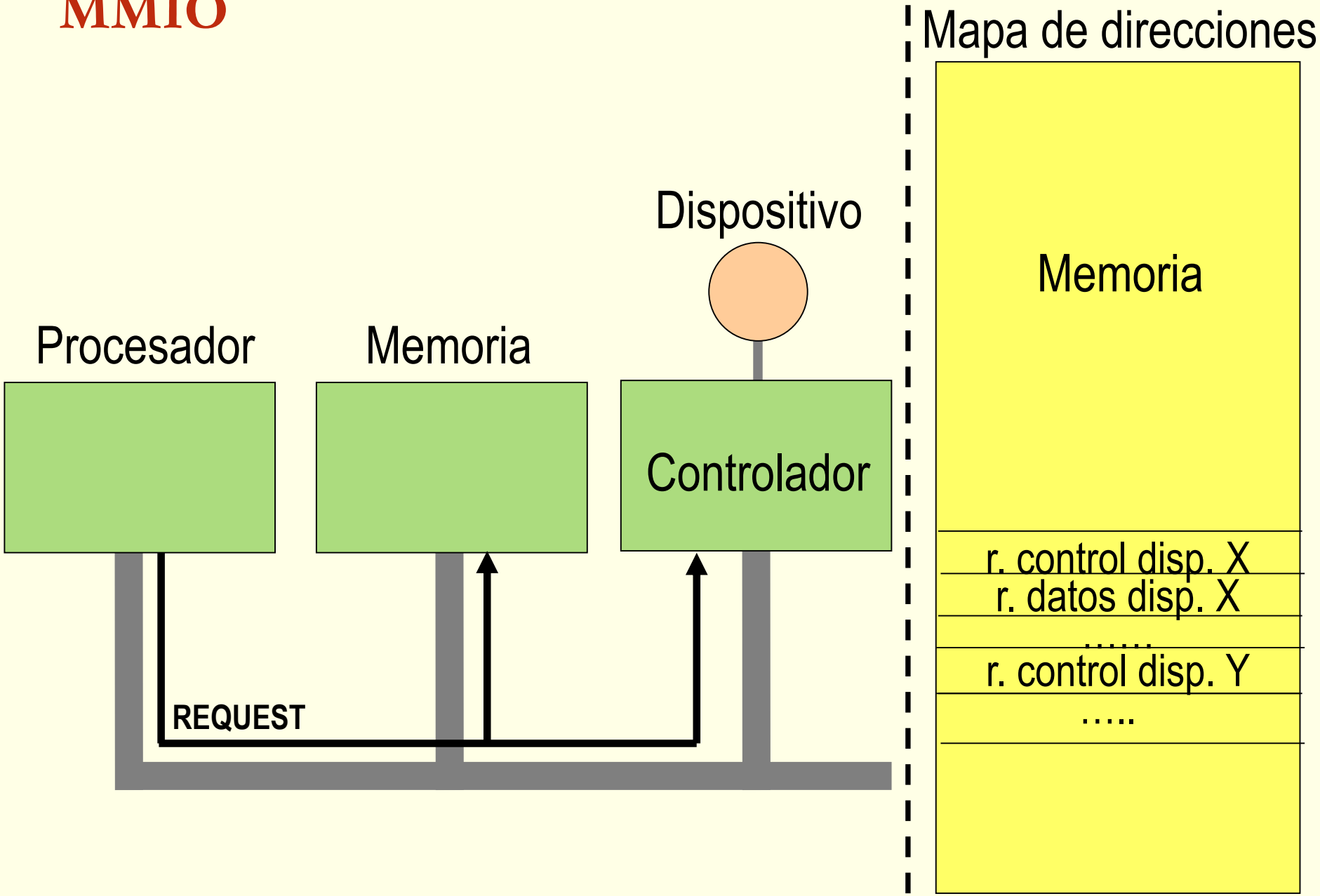
Raspberry: direcciones de bus, físicas y virtuales



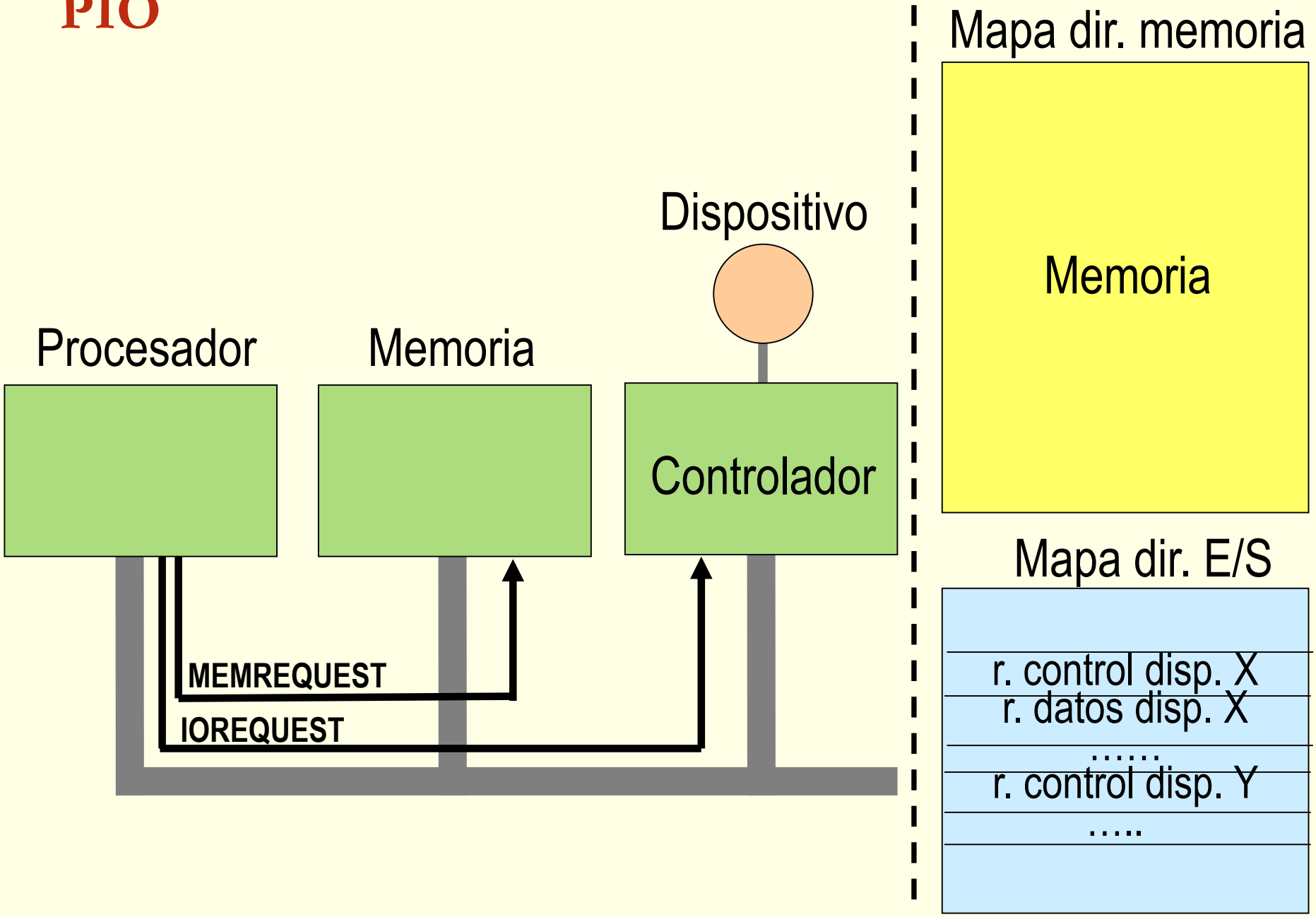
Memory-mapped I/O (MMIO) vs. *Port I/O* (PIO)

- MMIO: Mismo espacio de dir. e instrucciones para mem. y E/S
  - Instrucciones acceso a memoria convencionales: LOAD/STORE
- PIO: Distintos espacios de dir. e instrucciones para mem. y E/S
  - Bus incluye señal para discriminarlos (MEMREQ vs IOREQ)
  - Instrucciones de acceso específicas: IN/OUT
- Ventajas de MMIO en la programación de dispositivos
  - Procesador más sencillo
  - Puede usar cualquier tipo de direccionamiento en acceso a dispo
  - No requiere ensamblador
- Ventajas de PIO en la programación de dispositivos
  - Menos problemas de coherencia
- PIO no habitual excepto familia x86
  - Aunque también usa MMIO
  - Linux: ficheros /proc/ioports y /proc/iomem

MMIO


Mapa de direcciones
Dispositivo
Memoria
Procesador
Memoria
Controlador
r. control disp. X
r. datos disp. X
……
r. control disp. Y
…..
REQUEST

PIO


Mapa dir. memoria
Memoria
Dispositivo
Procesador
Memoria
Controlador
MEMREQUEST
IOREQUEST
Mapa dir. E/S
r. control disp. X
r. datos disp. X
……
r. control disp. Y
…..

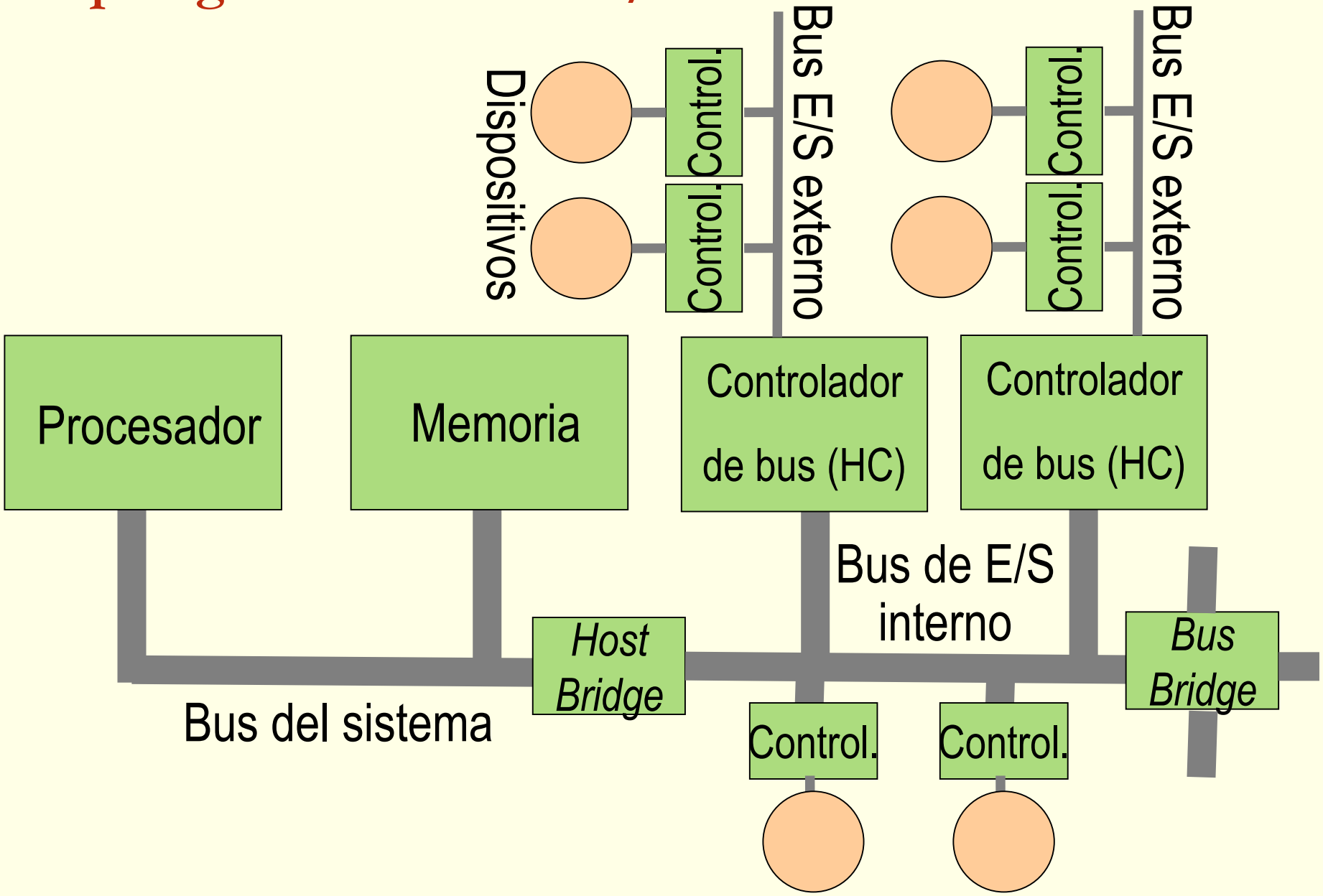
Configuración de dispositivos

- Cada dispositivo usa diversos recursos:
  - Rango dir E/S (puertos y/o memoria) + línea(s) de interrup.
- Asignación de recursos estática ("de fábrica") → colisiones
  - Solución usada en sistemas dedicados (p.e. RaspBerry)
- Asignación mediante *jumpers/switches* → poco flexible
- Mejor configurables por software
- ¿Cómo configurar dispositivo si no tiene dirección asignada?
  - Uso direccionamiento geográfico → posición (*slot*) en el bus
- Deseable *plug* & *play* (*P&P*) y *hot-plugging* (*H-P*)
    - *P&P*: Configuración automática de dispositivos en arranque
    - *H-P*: Conexión de dispositivo con el sistema arrancado
  - Requiere reconocer de qué tipo de dispositivo se trata
    - Para configurar y cargar su manejador en tiempo de ejecución
    - Dispo. ofrece información en sus registros de configuración
      - ID único, vendedor, clase, nº dir. E/S e interrupciones requeridas,…

Conexión al sistema de dispositivos: buses de E/S

- En sistemas primitivos o de baja gama
  - Controladores de dispositivo conectados a bus de sistema
  - Problemas de rendimiento y capacidad del bus sistema
  - Gran variedad en prestaciones de los dispositivos
- Uso habitual de buses de E/S
  - Bus de sistema conecta procesadores con memoria (y gráficos)
  - Controladores de dispositivo conectados a bus de E/S
  - Componente actúa de adaptador entre bus de sistema y de E/S
    - *Bus Host Bridge: HB* (*Bus Host Controller: HC*)
- Tipos de buses de E/S:
  - "internos": Dispositivos accesibles mediante MMIO/PIO
    - *HB* proporciona acceso directo transparente a dispositivos
  - "externos": UCP accede solo a *HC* usando MMIO/PIO
    - *HC* interacciona con dispositivos usando direcciones internas

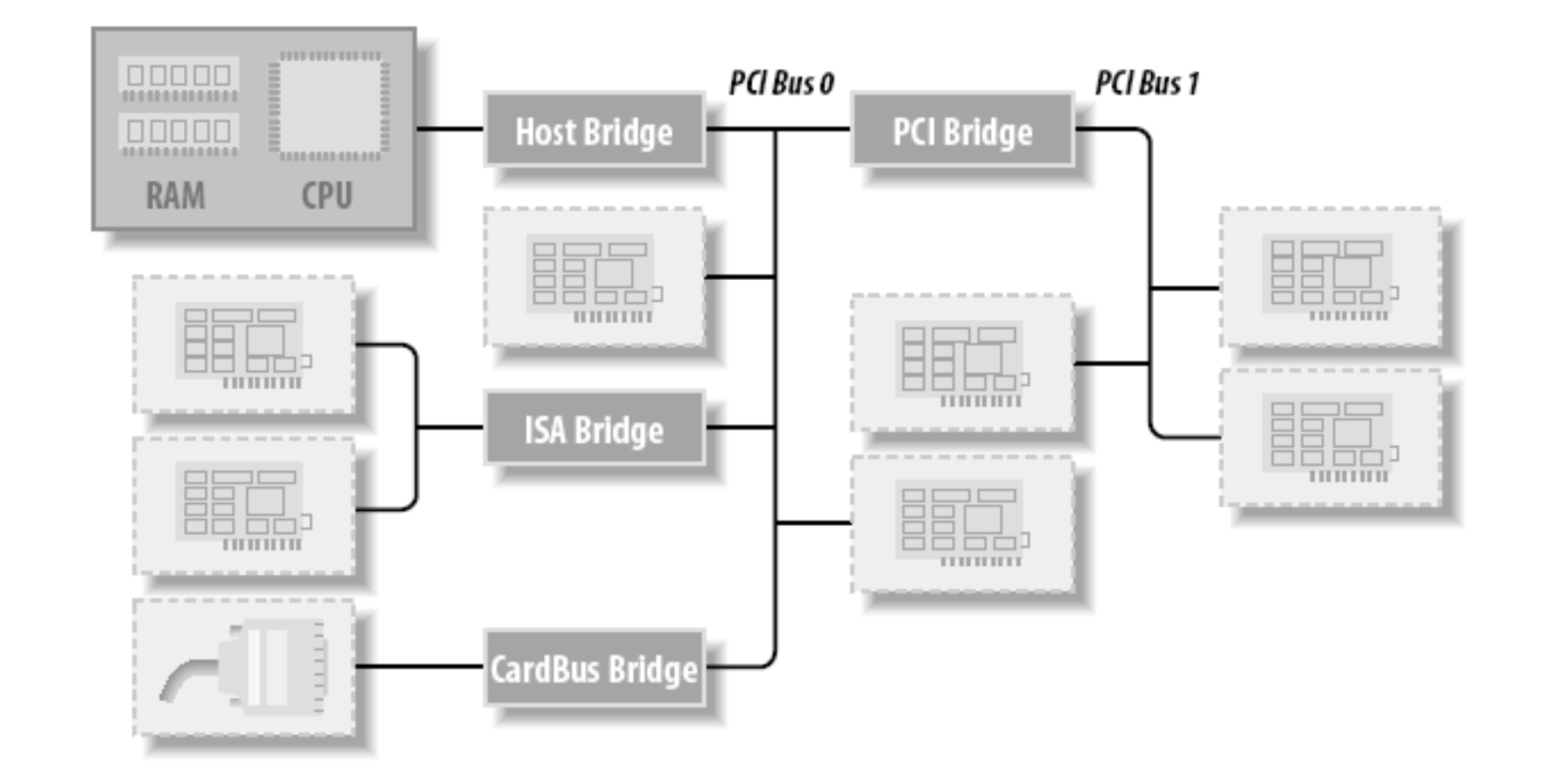
Topología de buses de E/S



Buses de E/S "internos" (PCI, ISA, ...)

- Controladores de dispositivos conectados a bus de E/S "interno"
  - Controladores directamente accesibles mediante PIO o MMIO
    - Como si estuvieran conectados al bus del sistema
  - *HC* de buses "externos" conectados a bus de E/S "interno"
- Jerarquía de buses: bus "interno" formado por varios "tramos"
  - Por limitaciones del bus (nº máx. dispositivos/tramo)
  - Por rendimiento (mayor paralelismo)
  - Por compatibilidad (conexión dispositivos incompatibles)
    - Uso de "tramo" que implementa otro tipo de bus "interno"
  - Puentes (*Bus Bridges*) permiten su interconexión
- Proceso de enumeración de dispositivos (si el bus lo permite)
  - "Descubrimiento" mediante direccionamiento geográfico
    - Accede a cada posición del bus atravesando *Bridges* encontrados
    - Obtención de info. de dispositivo (vendedor, ID, clase, ...)
    - Configuración de dirs. E/S (PIO o MMIO) e IRQs

Jerarquía de buses "internos"



Linux Device Drivers, 3ª Edición

Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

Bus PCI

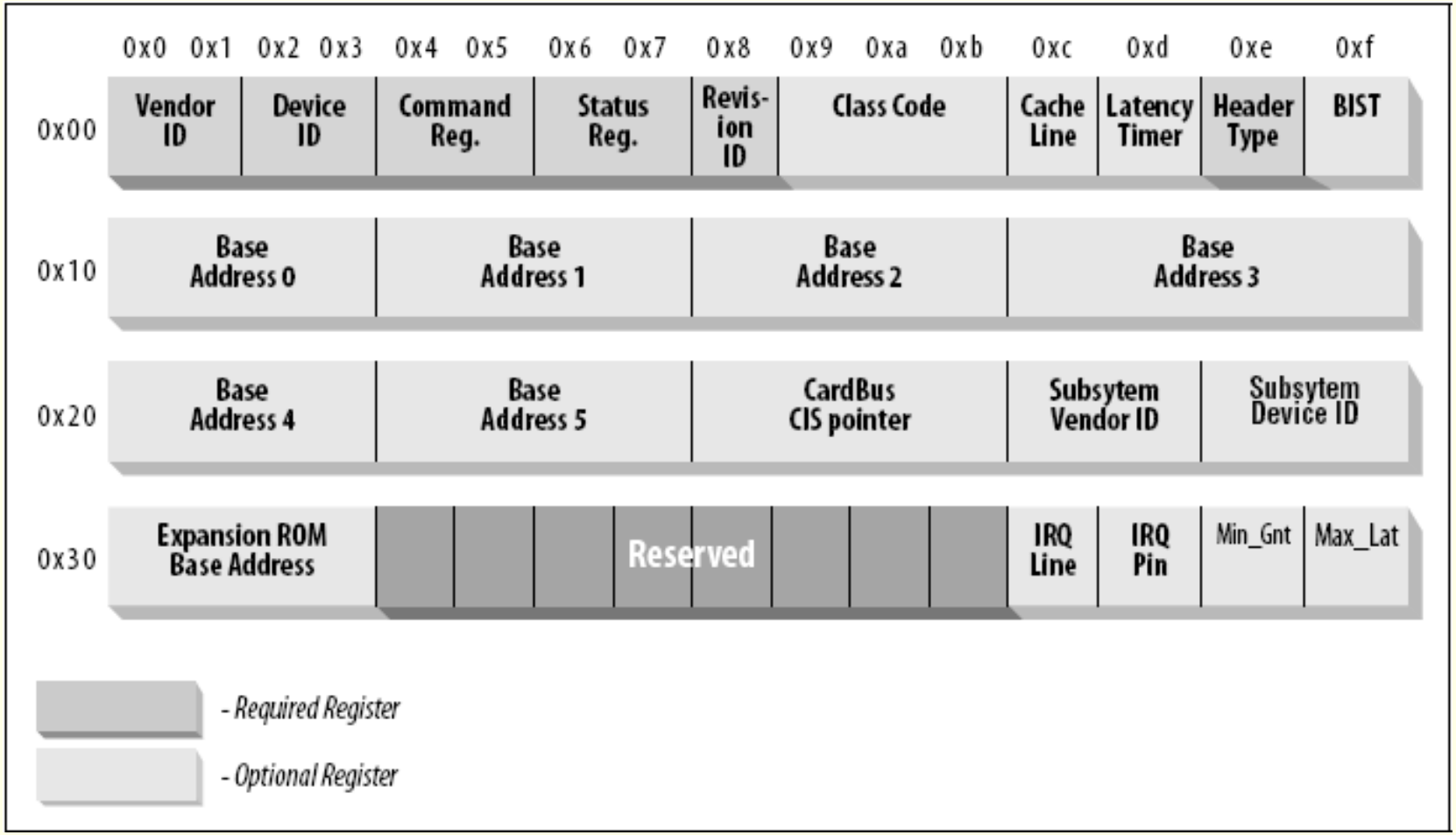
- Sustituto de ISA (desde *desktops* a grandes servidores)
  - Mucho más rápido, permite autoconfiguración y es "neutral"
  - PCI convencional, PCI-X, PCI Express (bus serie)
- Cada *slot* del bus: "componente" con múltiples "funciones"
  - Realmente, cada función es un dispositivo
- Bus jerárquico: formado por varios tramos conectados mediante
  - *PCI-PCI Bridges* (*PPB*) (Linux `lspci -tv`)
  - 256 buses (8 bits)
  - 32 *slots*/bus (5 bits)
  - 8 funciones/*slot* (3 bits)
- CPU dialoga con *Host Bridge* (*HB*)
  - *HB* controla acceso a un bus PCI (jerárquico)
  - Puede haber varios *HBs*: múltiples buses PCI
    - terminología Linux: múltiples dominios

Acceso al bus PCI

- Dispositivo PCI provee 2 tipos de accesos:
  - Configuración: acceso RW geográfico por posición *slot* en bus
    - Cada "función" proporciona registros de configuración (256B)
    - Acceso de configuración lee o escribe un registro
  - Una vez configurado: acceso convencional MMIO/PIO

- SW no puede realizar accesos de configuración
  - Solución habitual: *HB* proporciona dos puertos PIO
    - SW especifica dirección de acceso en CONFIG_ADDRESS (0xCF8)
      - nº bus + nº *slot* + nº función + nº r. configuración
    - SW lee/escribe valor r. configuración en CONFIG_DATA (0xCFC)

Info. de configuración de dispositivo en PCI



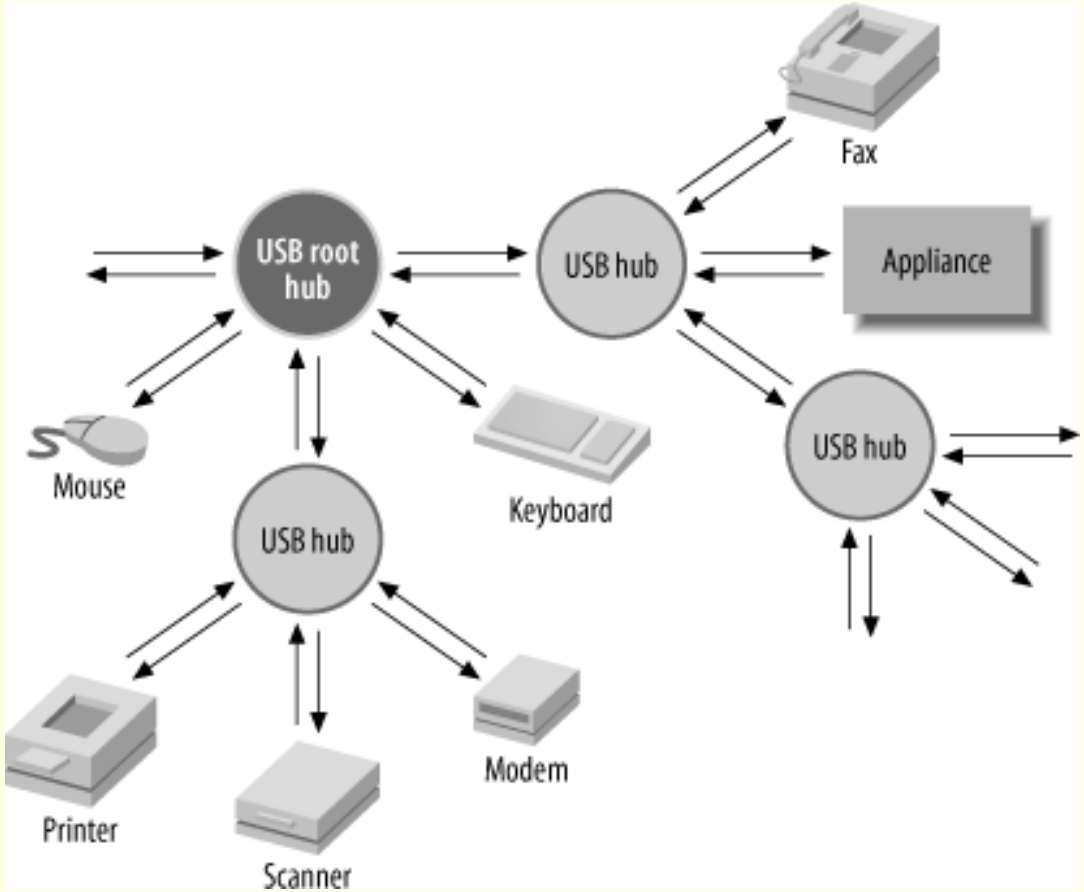
***Linux Device Drivers*, 3ª Edición**

Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

Buses de E/S "externos" (SCSI, USB, I²C, SPI,...)

- Conectados a bus sistema o de E/S interno usando *Host controllers*
  - Si conectado a un bus de E/S interno configurable
    - *HC* descubierto y configurado en enumeración de buses internos
  - Cada *HC* gestiona múltiples dispositivos
- SW interacciona (PIO/MMIO) con *HC*
  - *HC* interacciona con dispositivos conectados al bus externo
  - Dispositivo tiene asignada una dirección interna dentro del bus
  - SW indica esa dir. interna en operaciones MMIO/PIO con *HC*
  - Direcciones internas fijas ("de fabrica") o configurables por SW
- Enumeración de bus externo (si lo permite, como USB)
  - Descubrimiento y conf. dispositivos (no dirs. E/S ni IRQs)
    - Asigna una dirección interna (de 7 bits en USB)
    - Puede obtenerse info. de dispositivo (vendedor, ID, clase, ...)
      - Linux: mandato `lsusb -tv`

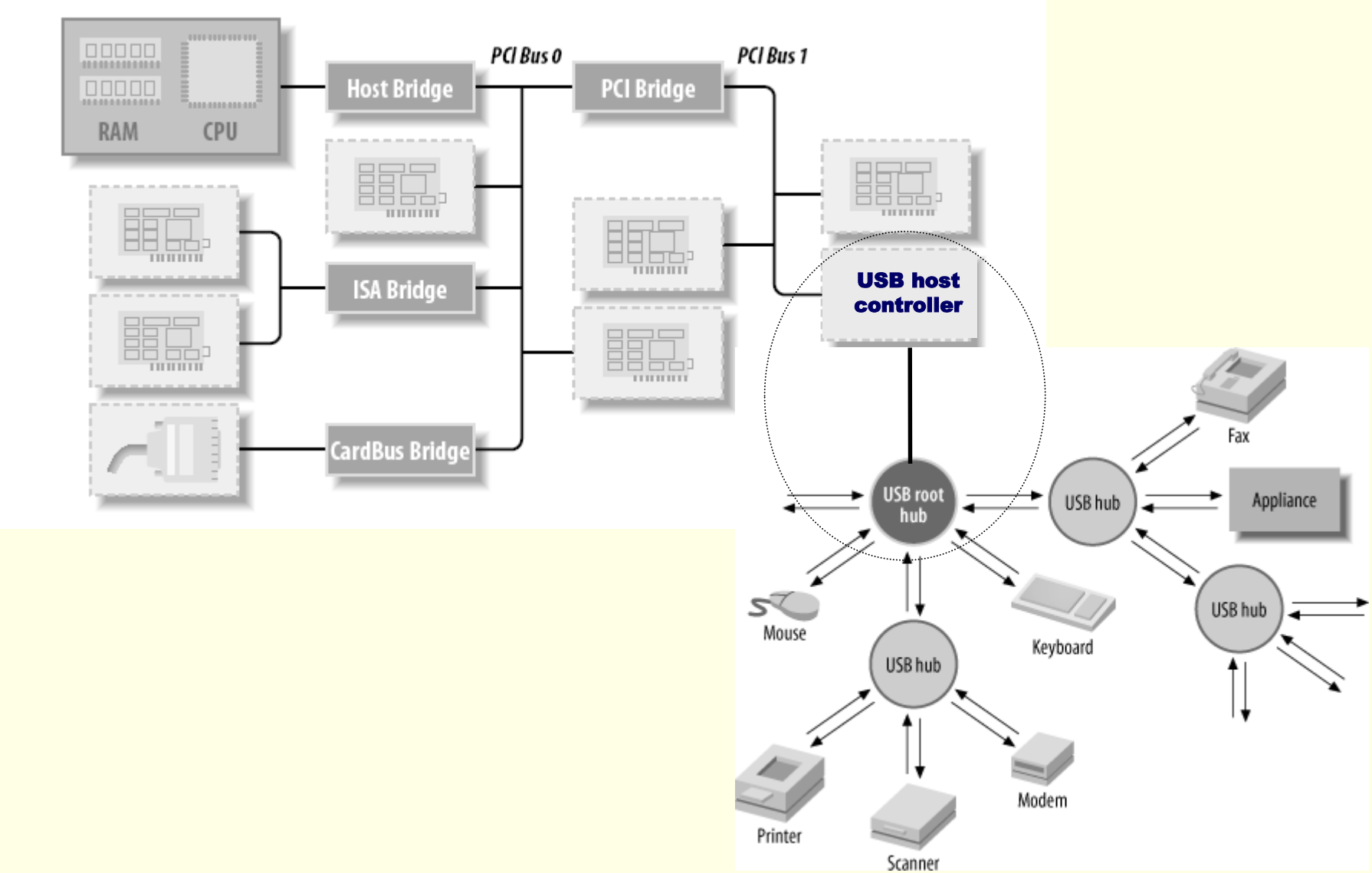
Bus externo USB



Designing Embedded Hardware

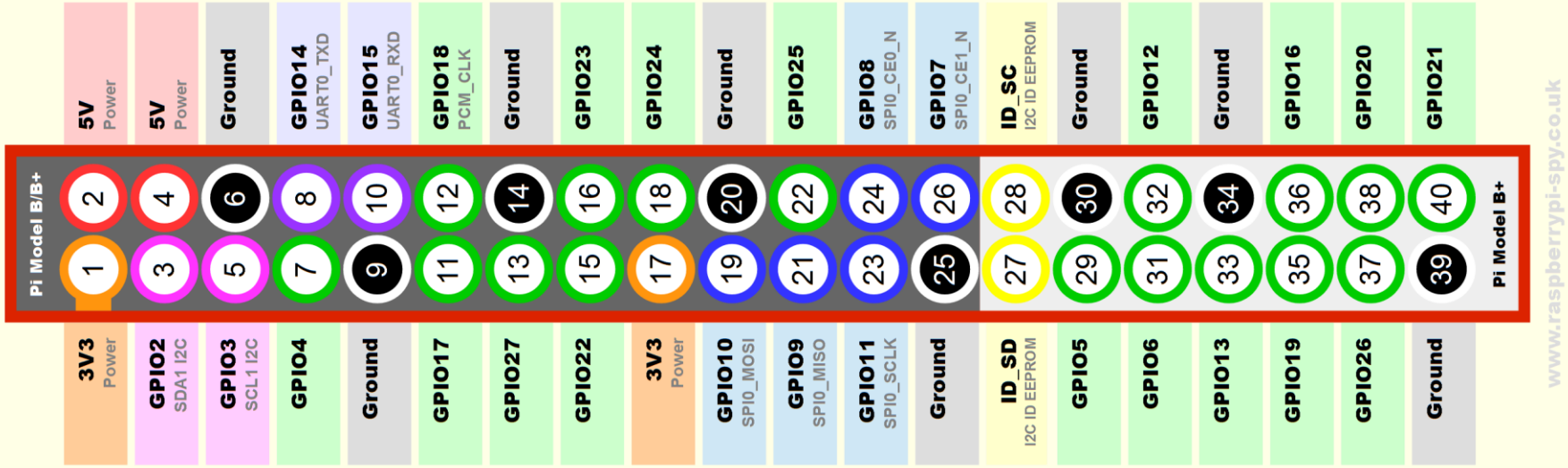
John Catsoulis

Jerarquía de buses de E/S



General-purpose input/output (GPIO)

- Sistemas dedicados y empotrados suelen ofrecer GPIO
- Colección de pines tal que cada uno acarrea una señal digital
- Uso de propósito general: sensor, LED, motor, botón,…
- Se pueden configurar como de entrada o salida
- Pueden tener asociadas interrupciones
- Multiplexación de pines: algunos pines pueden configurarse
  - Uso GPIO o específico (PWM, SPI, I^{2}C,…)



https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	<reserved>			
GPIO1	High	SCL0	SA4	<reserved>			
GPIO2	High	SDA1	SA3	<reserved>			
GPIO3	High	SCL1	SA2	<reserved>			
GPIO4	High	GPCLK0	SA1	<reserved>			ARM_TDI
GPIO5	High	GPCLK1	SA0	<reserved>			ARM_TDO
GPIO6	High	GPCLK2	SOE_N / SE	<reserved>			ARM_RTCK
GPIO7	High	SPI0_CE1_N	SWE_N / SRW_N	<reserved>			
GPIO8	High	SPI0_CE0_N	SD0	<reserved>			
GPIO9	Low	SPI0_MISO	SD1	<reserved>			
GPIO10	Low	SPI0_MOSI	SD2	<reserved>			
GPIO11	Low	SPI0_SCLK	SD3	<reserved>			
GPIO12	Low	PWM0	SD4	<reserved>			ARM_TMS
GPIO13	Low	PWM1	SD5	<reserved>			ARM_TCK
GPIO14	Low	TXD0	SD6	<reserved>			TXD1
GPIO15	Low	RXD0	SD7	<reserved>			RXD1
GPIO16	Low	<reserved>	SD8	<reserved>	CTS0	SPI1_CE2_N	CTS1
GPIO17	Low	<reserved>	SD9	<reserved>	RTS0	SPI1_CE1_N	RTS1
GPIO18	Low	PCM_CLK	SD10	<reserved>	BSCSL SDA / MOSI	SPI1_CE0_N	PWM0
GPIO19	Low	PCM_FS	SD11	<reserved>	BSCSL SCL / SCLK	SPI1_MISO	PWM1
GPIO20	Low	PCM_DIN	SD12	<reserved>	BSCSL / MISO	SPI1_MOSI	GPCLK0
GPIO21	Low	PCM_DOUT	SD13	<reserved>	BSCSL / CE_N	SPI1_SCLK	GPCLK1
GPIO22	Low	<reserved>	SD14	<reserved>	SD1_CLK	ARM_TRST	
GPIO23	Low	<reserved>	SD15	<reserved>	SD1_CMD	ARM_RTCK	
GPIO24	Low	<reserved>	SD16	<reserved>	SD1_DAT0	ARM_TDO	
GPIO25	Low	<reserved>	SD17	<reserved>	SD1_DAT1	ARM_TCK	
GPIO26	Low	<reserved>	<reserved>	<reserved>	SD1_DAT2	ARM_TDI	
GPIO27	Low	<reserved>	<reserved>	<reserved>	SD1_DAT3	ARM_TMS	
GPIO28	-	SDA0	SA5	PCM_CLK	<reserved>		
GPIO29	-	SCL0	SA4	PCM_FS	<reserved>		
GPIO30	Low	<reserved>	SA3	PCM_DIN	CTS0		CTS1
GPIO31	Low	<reserved>	SA2	PCM_DOUT	RTS0		RTS1
GPIO32	Low	GPCLK0	SA1	<reserved>	TXD0		TXD1
GPIO33	Low	<reserved>	SA0	<reserved>	RXD0		RXD1
GPIO34	High	GPCLK0	SOE_N / SE	<reserved>	<reserved>		
GPIO35	High	SPI0_CE1_N	SWE_N / SRW_N		<reserved>		
GPIO36	High	SPI0_CE0_N	SD0	TXD0	<reserved>		
GPIO37	Low	SPI0_MISO	SD1	RXD0	<reserved>		
GPIO38	Low	SPI0_MOSI	SD2	RTS0	<reserved>		
GPIO39	Low	SPI0_SCLK	SD3	CTS0	<reserved>		
GPIO40	Low	PWM0	SD4		<reserved>	SPI2_MISO	TXD1
	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5

Modo de operación de GPIO de Raspberry

- Acceso a registros de configuración mediante MMIO
- Primero: Selección de función del pin → GPFSEL0 a GPFSEL5
  - Direcciones físicas: 0x7E20000 a 0x7E20004
  - GPFSEL0 controla pines de 0 a 9, GPFSEL1 pines 10 a 19 bits,…
  - Cada pin controlado por 3 bits: entrada, salida, función alternat. (0-5)
  - P.e. selección de función del pin 0 como salida
    - Escribir 001 en tres primeros bits de GPFSEL0
- Poner 1 en un pin→GPSET0 y GPSET1 (0x7E2001C y 0x7E20020)
  - Escribe 1 en bit correspondiente (pin 0 → bit 0 de GPSET0,…)
- Poner 0 en un pin→GPCLR0 y GPCLR1 (0x7E20028 y 0x7E2002C)
  - Escribe 1 en bit correspondiente (pin 0 → bit 0 de GPCLR0,…)
- Leer pin→ GPLEV0 y GPLEV1 (0x7E20034 y 0x7E20038)
  - Lee bit correspondiente (pin 0 → bit 0 de GPLEV0,…)

Contenido

- Introducción
- El hardware de E/S visto desde el software
- **Aspectos generales de la programación de dispositivos**
- Programación de manejadores de dispositivos
  - Caso práctico: programación de manejadores en Linux

PIO

- Requiere instrucciones en ensamblador. En x86:
  - IN|INS|INSB|INSW|INSD; OUT|OUTS|OUTSB|OUTSW|OUTSD
- En sistema propósito general, acceso dispositivo solo para SO
  - En UCP con modos de ejecución → ops. PIO privilegiadas
  - x86 por defecto ops. PIO no disponibles en modo usuario pero
    - Campo IOPL de r. estado define nivel privilegio para acceso PIO
      - Si igual a 3, en modo usuario accesibles todos los puertos
    - Mapa de bits de permisos de E/S en TSS (*Task State Segment*)
      - Especifica qué puertos son accesibles para el proceso en modo usuario
- Para acceso PIO en modo usuario sin ensamblador, Linux ofrece:
  - Funciones *inline* inb, inw, outb, outw,…
  - Llamadas iopl/ioperm para modificar IOPL/mapa permisos E/S
    - solo para proceso *root* o con la *capability* CAP_SYS_RAWIO

Ejemplo PIO en modo usuario en Linux

fperez@fperez-VirtualBox: ~/SEU/PIO

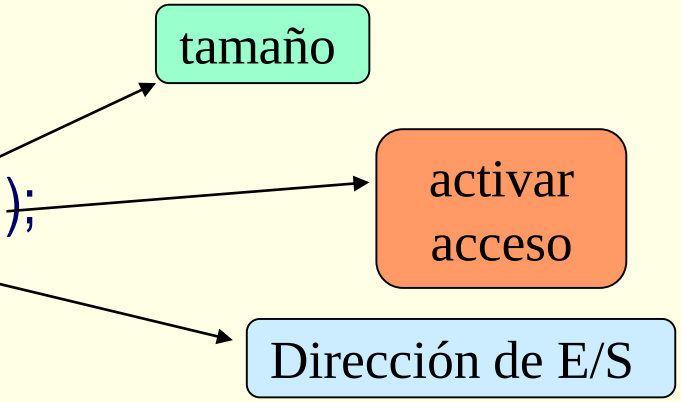
Archivo Editar Ver Buscar Terminal Ayuda

```
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>
#define KEYBOARD 0x60
#define MASC (1<<7)
int main() {
        if (ioperm(KEYBOARD, 1, 1) < 0) {
                perror("ioperm");
                return 1;
        }
        while(1){
                uint8_t codigo= inb(KEYBOARD);
                printf("Código de tecla %d", codigo & ~MASC);
                if (codigo | MASC)
                        printf(" %s\n", (codigo & MASC) ? "liberada" : "pulsada");
                sleep(5);
        }
        return 0;
}
                                                        20,1          Final
```

Ejemplo PIO en modo usuario en Linux

Acceso disp. RTC que usa puertos 0x70 y 0x71 de 1 byte

```
#define RTC_REG 0x70
#define RTC_DAT 0x71
int main() {
    uint8_t valor, dato;
    ioperm(RTC_REG, 2, 1);
    .........
    outb(valor, RTC_REG);
    ..........
    dato = inb(RTC_DAT));
    .........
```



MMIO

- MMIO permite acceso convencional con puntero a dir. registro

```
uint32_t volatile *p = (uint32_t volatile *)(0x40000000); // si no MMU
*p = 1;
```

- Normalmente, acceso solo en modo privilegiado
  - Las direcciones de E/S solo aparecen en el mapa del SO
  - Si procesador usa MMU, necesidad de crear *mapping*
    - Hacer que una DL corresponda a DF deseada (Linux ioremap)
- Para acceso modo usuario: dir. E/S dentro del mapa del proceso
  - En Linux mediante *mmap* de /dev/mem (solo para proceso *root*)
    - Proporciona acceso a memoria física del sistema
    - *Offset* corresponde a dirección física de los registros del dispo.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Ejemplo MMIO en modo usuario en Linux

Local-APIC, por defecto, ofrece sus registros de 4 bytes a partir de d. física 0xfee00000 ocupando hasta 4K. Acceso a registro ubicado a *X* bytes con respecto a esa dirección.

```
int main() {
    int fd, tam=4096;
    uint32_t dato, desp = X;
    uint32_t volatile *p;
    fd = open("/dev/mem", O_RDONLY);
    p = mmap(NULL, tam, PROT_READ, MAP_SHARED, fd, 0xfee00000);
    .......
    dato = *(p + desp/4);
    .........
```

offset:
Debe ser múltiplo del tamaño de la página

Acceso GPIO Raspberry en modo usuario en Linux

- Modos de acceso alternativos

  https://elinux.org/RPi_GPIO_Code_Samples

- Uso de */dev/mem*

  https://elinux.org/RPi_GPIO_Code_Samples#Direct_register_access

  - Debe gestionar todos los detalles de bajo nivel

- Uso de */dev/gpiomem* (no requiere *root*)
  - Complejidad similar a */dev/mem* pero
    - *mmap* devuelve dirección base de GPIO (no requiere *offset*)
- A través de *sysfs* quedan ocultos los detalles de bajo nivel

  https://elinux.org/RPi_GPIO_Code_Samples#sysfs

  https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

- Mediante */dev/gpiochipN* (sustituye a *sysfs*)

  https://archive.fosdem.org/2018/schedule/event/new_gpio_interface_for_linux/attachments/slides/2094/export/events/attachments/new_gpio_interface_for_linux/slides/2094/FOSDEM_18_New_GPIO_interface_for_linux.

- Bibliotecas específicas para distintos lenguajes
  - Que internamente usan alguno de los mecanismos anteriores

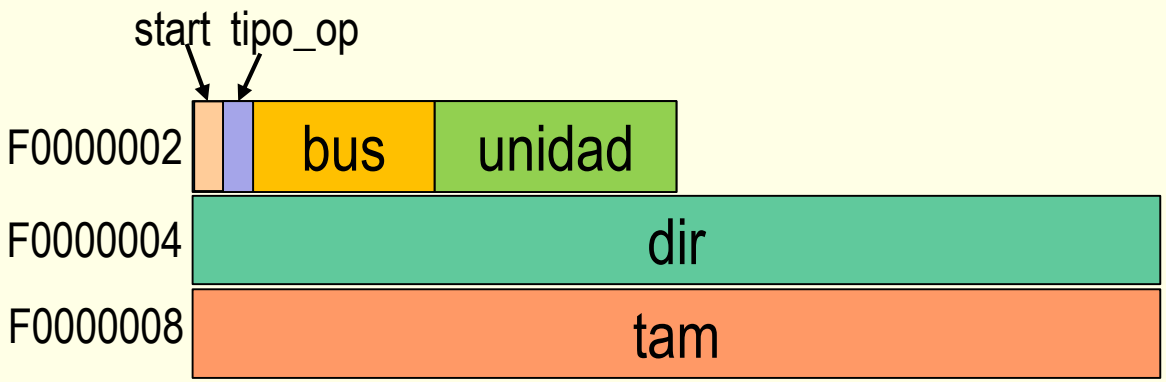
Acceso MMIO a los registros del controlador de E/S

- Información en registros de dispositivo suele tener una estructura
  - Para facilitar su manejo → definir tipo de datos que la refleje
- Gestión de campos a nivel de bits. Alternativas en C:
  - Uso de *bitfields* vs Uso de máscaras de bits
    - Programación más "limpia" pero dependencia del compilador
- Compilador puede incluir relleno entre campos
  - Impediría la correspondencia entre tipo y registros del disp.-
    - Algunos compiladores permiten eliminarlo
- *Endianness* (orden de los bytes en una palabra)
  - Problema en comunicación; también en programación de dispo.
    - Dispositivo utiliza un determinado *endian*
    - P.ej. información de dispositivo de PCI es *little-endian*

```
le16toh(VendorID);
```

Acceso MMIO con *bitfields*

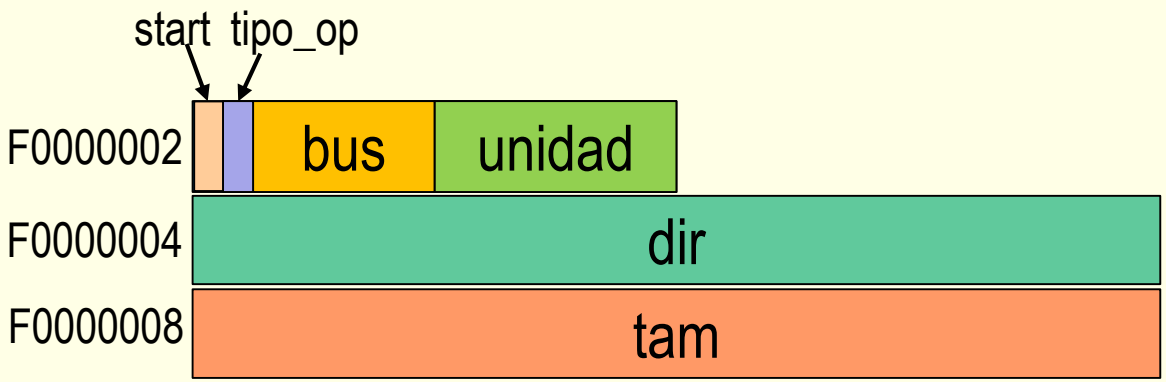
```
#define WRITE 1
struct info_dispo {
        uint16_t unidad:8;
        uint16_t bus:6;
        uint16_t tipo_op:1;
        uint16_t start:1;
        uint32_t dir;
        uint32_t tam;
} __attribute__ ((packed)); // si no ocuparía 12 bytes y no estaría alineado con registros del dispositivo
```




```
struct info_dispo *d = (struct info_dispo *)0xF0000002; // suponiendo que no hay MMU
d->dir = 0x……..; d->tam = ……..;
d->unidad = unidad;
d->bus = bus;
d->tipo_op = WRITE;
d->start = 1;
```

Acceso MMIO con máscaras de bits

```
#define WRITE 1
struct info_dispo {
        uint16_t registro;
        uint32_t dir;
        uint32_t tam;
} __attribute__ ((packed));
```




```
struct info_dispo *d = (struct info_dispo *)0xF0000002; // suponiendo que no hay MMU
d->dir = 0x……..;
d->tam = …….;
d->registro =  1<<15 | WRITE << 14 | bus << 8 | unidad;
;
```

Acceso MMIO: el compilador nos la juega

```
char *p = (char  *)(0x40000000);
*p = 1;                    // arranca operación en dispositivo
while (*p == 0);           // espera que op. se complete; ¡pero no espera!
```

```
uint32_t  *timer = (uint32_t  *)(0x60000000);
tini= *timer;              // toma de tiempo inicial
x=a*b*c*d/e/f;             // operación a medir
ttot= *timer - tini;       // ¡Devuelve 0!
```

- El compilador no sabe que esas variables pueden cambiar de valor

Optimizaciones del compilador y del procesador

- Las optimizaciones del compilador pueden:
  - Escribir/leer en/de registros del procesador y no en memoria
    - Operación de escritura en dispo. se queda en registro de UCP
  - Eliminar sentencias
    - el while porque nunca se cumple
    - En 2 escrituras sucesivas a la misma posición solo dejar la 2ª
  - Reordenar instrucciones
- El procesador tampoco ayuda:
  - Usa caché
    - Solución: desactivar cache (p.e. *flag* PCD entrada de TP de x86)
  - Reordena instrucciones y accesos a memoria
    - Asegurando que no cambia semántica del programa
    - Pero afecta a la programación de dispositivos

Volatile en C

```
char volatile *p = (char  volatile *)(0x40000000);
uint32_t const volatile *timer = (uint32_t const volatile *)(0x60000000);
```

- Indica al compilador que no se optimice acceso a esa variable
  - Lectura/Escritura de variable → LOAD/STORE en memoria
  - No puede eliminar accesos a esa variable
  - No reordenar accesos a variables de este tipo
    - Pero no especifica orden accesos volátiles y no volátiles
- Definición en el estándar ambigua: obliga a compilador
  - Pero no a procesador: este puede reordenar accesos a volátiles

---

- PREGUNTA: ¿tiene sentido const y volatile?
- NOTA: *volatile* de Java además implica aspectos adicionales:
  - Atomicidad en los accesos (el de C no lo asegura)
  - Crea relación de causalidad (*happens before*) en accesos

Acceso MMIO: reordenamiento de instrucciones

- Escenario típico de prog. dispositivo: implica varias operaciones:

```
dev->reg_addr =        io_destination_address;
dev->reg_size =        io_size;
dev->reg_operation =   READ;
dev->reg_control =     START;
```

- Compilador/UCP pueden optimizar reordenando instrucciones
  - Para ellos son independientes entre sí
  - START debe ejecutar después de completadas otras 3 sentencias
- ¿Solución?: marcar como volatile puntero dev

```
struct dev_t volatile *dev; // en struct: se aplica a todos sus campos
```

  - Indica a compilador no reordenar accesos de ese tipo
  - No suficiente: UCP puede optimizar reordenando instrucciones
- Uso de barreras (de compilador y de memoria):
  - Crean una sincronización en el punto donde aparecen

Barreras del compilador

- Barrera de optimización del compilador:
  - Al llegar a barrera, valores de registros → variables en memoria
  - Después, primeros accesos a variables → a memoria
  - No movimiento de instrucciones entre lados de la barrera
  - GNU cc: asm volatile("" ::: "memory");
- Puede ser más eficiente que volatile. En el ejemplo:
  - Basta con asegurar que START se ejecuta al final
  - Compilador podría optimizar sentencias previas
- Como volatile, barrera de compilador no es suficiente en el ejemplo
  - Procesador puede optimizar reordenando instrucciones
    - Solución: añadir barrera de memoria (BM; mecanismo hardware)
- NOTA: PIO usa ensamblador → no aplicable barreras compilador

Barrera de memoria

- Establece un orden parcial en accesos previos y posteriores
  - Barrera completa (Pentium MFENCE):
    - LDs|STs antes BM globalmente visibles antes que LDs|STs después
      - NOTA: LD Instrucción LOAD; ST instrucción STORE
  - Barrera de lectura (Pentium LFENCE):
    - LDs antes BM globalmente visibles antes que LDs de después
  - Barrera de escritura (Pentium SFENCE):
    - STs antes BM globalmente visibles antes que STs de después
- Normalmente, si se necesita BM, también barrera de compilador
  - Linux rmb, wmb, mb incluyen ambos tipo de barreras
- NOTA: PIO no accede a memoria → no aplicable barreras memoria
  - En ejemplo solo requiere que UCP no reordene instrucciones
  - *Flush pipeline* de instrucciones (Motorola NOP; x86 CPUID)

Acceso MMIO y PIO: Solución

- MMIO:

```
dev->reg_addr =          io_destination_address;
dev->reg_size =          io_size;
dev->reg_operation =     READ;
wmb();      // barrera de compilador + barrera de memoria de escritura
dev->reg_control =       START;
```

- PIO:

```
OUT #io_destination_address, dev->reg_addr
OUT #io_size, dev->reg_size
OUT #READ, dev->reg_operation
flush_pipeline();
OUT #START, dev->reg_control
```

Nota sobre Raspberry

- Sistema de E/S de Raspberry también nos la juega

  "*Accesses to the same peripheral will always arrive and return in-order. It is only when switching from one peripheral to another that data can arrive out-of-order. The simplest way to make sure that data is processed in-order is to place a memory barrier instruction at critical positions in the code. You should place:*

  - *A memory write barrier before the first write to a peripheral.*
  - *A memory read barrier after the last read of a peripheral.*

  For example:

  ```
  a_status = *pointer_to_peripheral_a;
  b_status = *pointer_to_peripheral_b;
  ```

  Without precautions the values ending up in the variables a_status and b_status can be swapped around."

Programación de interrupciones

- Si rutina interrupción y código interrumpido comparten variable:
  - Debe ser volatile
  - Pero requiere además ser de actualización atómica (sig_atomic_t)
    - Escritura en variable solo requiera una instrucción

      volatile sig_atomic_t v;
  - No solo para interrupciones: también en manejo señales UNIX
- Puede ser necesario crear sección crítica (SC) con respecto a int.
    - P.e. código interrumpido y rut.int. insertan/borran nodos en lista
  - En UP prohibir interrupción conflictiva en la SC
  - En MP además *spinlocks*
  - Minimizar tiempo SC → minimiza latencia de activación int.
- "Regla de oro" en el procesamiento de interrupciones
  - Minimizar duración de rutina de tratamiento de interrupción
    - En rutina solo operaciones críticas
  - Mejor tiempo respuesta y menos problemas de sincronización

Programación de DMA

- Supongamos operación de lec. o esc. con múltiples *buffers*
  - Escritura: ***N*** *buffers* → Dispositivo
  - Lectura: ***N*** *buffers* ← Dispositivo
- Alternativas:
  - Sin IO-MMU ni *scatter-gather DMA*: *N* ops. DMA
    - Reg. dir. de cont. DMA: DF de cada sucesivo *buffer*
  - Sin IO-MMU pero con *scatter-gather*: 1 op. DMA
    - Regs. dir. de cont. DMA: DF de *buffers*
  - Con IO-MMU: 1 op. DMA
    - Programar IO-MMU para ver *buffers* como DB contiguas
    - Reg. dir. de cont. DMA: DB de *buffer*
- Si HW no asegura coherencia de memoria, SW debe hacerlo
  - Antes de escritura: volcado caché de datos involucrados
  - Después de lectura: invalidación caché de datos involucrados

Aspectos de configuración

- Con dispositivos no configurables por software
  - SW no debe usar dir. E/S ni IRQ fijas sino parametrizables
  - *Device Trees*: fichero de config. proporcionado por usuario
- Con dispositivos configurables por software
  - En arranque (o en *plug*) recorre jerarquía de buses internos
    - Descubre dispositivos usando direccionamiento geográfico
    - Les asigna dir. de E/S e IRQs evitando conflictos
    - Labor realizada por *firmware* o el SO
      - o la propia aplicación si máquina desnuda
    - Si *firmware*, después SO averigua dir. E/S e IRQs dispo.
      - Leyendo regs. config. correspondientes mediante dir. geográfico
  - En arranque (o en *plug*) "enumera" buses externos
    - Les asigna dirección interna en el bus
    - Realizado por *HC* + SW (*firmware*, SO o la aplicación)

Introducción a la programación del bus PCI

- Sistema inicialmente desconfigurado. SW "enumera" buses:
  - for (n=0; n<32; n++) → acceso de lectura de configuración a:
    - Bus 0, *slot* n, función 0, registro de configuración 0
    - Si no existe, lectura devuelve todos los bits a 1; continue
    - Si existe y multifunción, prueba las otras 7 funciones
    - Por cada función encontrada, si es *PCI-PCI Bridge* (PPB)
      - Asigna valor a bus secundario y repite enumeración para ese bus
    - Si no es un PPB → configuración de función (dispositivo)
- Configuración dispo: Múltiples regiones MMIO o PIO asignadas
  - Asignación direcciones usando *Base Address Registers* (BAR)
    - Escribiendo palabra con 1s en BAR y leyendo a continuación
      - Tamaño requerido por la región
    - Escribir en BAR dirección MMIO o PIO asignada
- http://wiki.osdev.org/PCI

Ejemplo hipotético de programación de dispositivos

- Programación de operación de escritura por DMA
- DMA sin coherencia de cache
- Dispositivo acceso MMIO con 3 registros de E/S en direcciones:
  - X registro dirección DMA (32 bits)
  - X+1 registro tamaño transferencia DMA (32 bits)
  - X+2 registro control (32 bits): Escribiendo un 1 inicia escritura
- Estructura que representa los registros del dispositivo:

```
struct dispo {
    uint32_t dir;       // dirección de DMA
    uint32_t tam;       // tamaño de transferencia de DMA
    uint32_t rctl;      // registro de control
};
```

Ejemplo en sistema sin MMU ni IO-MMU

```
struct dispo *dis;      // para referenciar los registros del dispositivo
struct dato_a_escribir v;
…….. // incluye en variable v la información que se pretende a escribir
flush_cache(&v, sizeof(v));   // vuelca a memoria datos asociados a v
dis = X;                      // dirección del primer registro del dispositivo
dis->dir = &v;                // dirección del buffer a escribir por DMA
dis->tam = sizeof(v);         // tamaño de los datos a escribir por DMA
wmb();                        // barrera compilador + b. memoria de escritura
dis->rctl=1;                  // inicia escritura por DMA
```

Ejemplo en sistema con MMU pero sin IO-MMU

```
struct dispo *dis;      // para referenciar los registros del dispositivo
uint32_t *dirf;         // para guardar dir. física de buffer a escribir
struct dato_a_escribir v;
.......     // incluye en variable v la información que se pretende a escribir
flush_cache(&v, sizeof(v));   // vuelca a memoria datos asociados a v
dis = mmu_map(X,12);          // crea en MMU dir. lógicas asociadas X,X+1,...
dirf = virt_to_phys(&v);      // dirección física del buffer a escribir por DMA
dis->dir = dirf;              // dirección del buffer a escribir por DMA
dis->tam = sizeof(v);         // tamaño de los datos a escribir por DMA
wmb();                        // barrera compilador + b. memoria de escritura
dis->rctl=1;                  // inicia escritura por DMA
```

Ejemplo en sistema con MMU e IO-MMU

```
struct dispo *dis;      // para referenciar los registros del dispositivo
uint32_t *dirb;         // para guardar dir. de bus de buffer a escribir
struct dato_a_escribir v;
.......     // incluye en variable v la información que se pretende a escribir
flush_cache(&v, sizeof(v));   // vuelca a memoria datos asociados a v
dis = mmu_map(X,12);          // crea en MMU dir. lógicas asociadas X,X+1,...
// crea en IOMMU direcciones de bus asociadas a buffer a escribir
dirb = iommu_map(virt_to_phys(&v),sizeof(v));
dis->dir = dirb;              // dirección del buffer a escribir por DMA
dis->tam = sizeof(v);         // tamaño de los datos a escribir por DMA
wmb();                        // barrera compilador + b. memoria de escritura
dis->rctl=1;                  // inicia escritura por DMA
```

Contenido

- Introducción
- El hardware de E/S visto desde el software
- Aspectos generales de la programación de dispositivos
- **Programación de manejadores dispositivos**
  - Programación de manejadores dispositivos en sistemas sin SO
  - Programación en SS.OO. monolíticos
  - Programación en SS.OO. basados en micronúcleos
- Caso práctico: programación de manejadores en Linux

Programación de dispositivos en máquinas sin SO

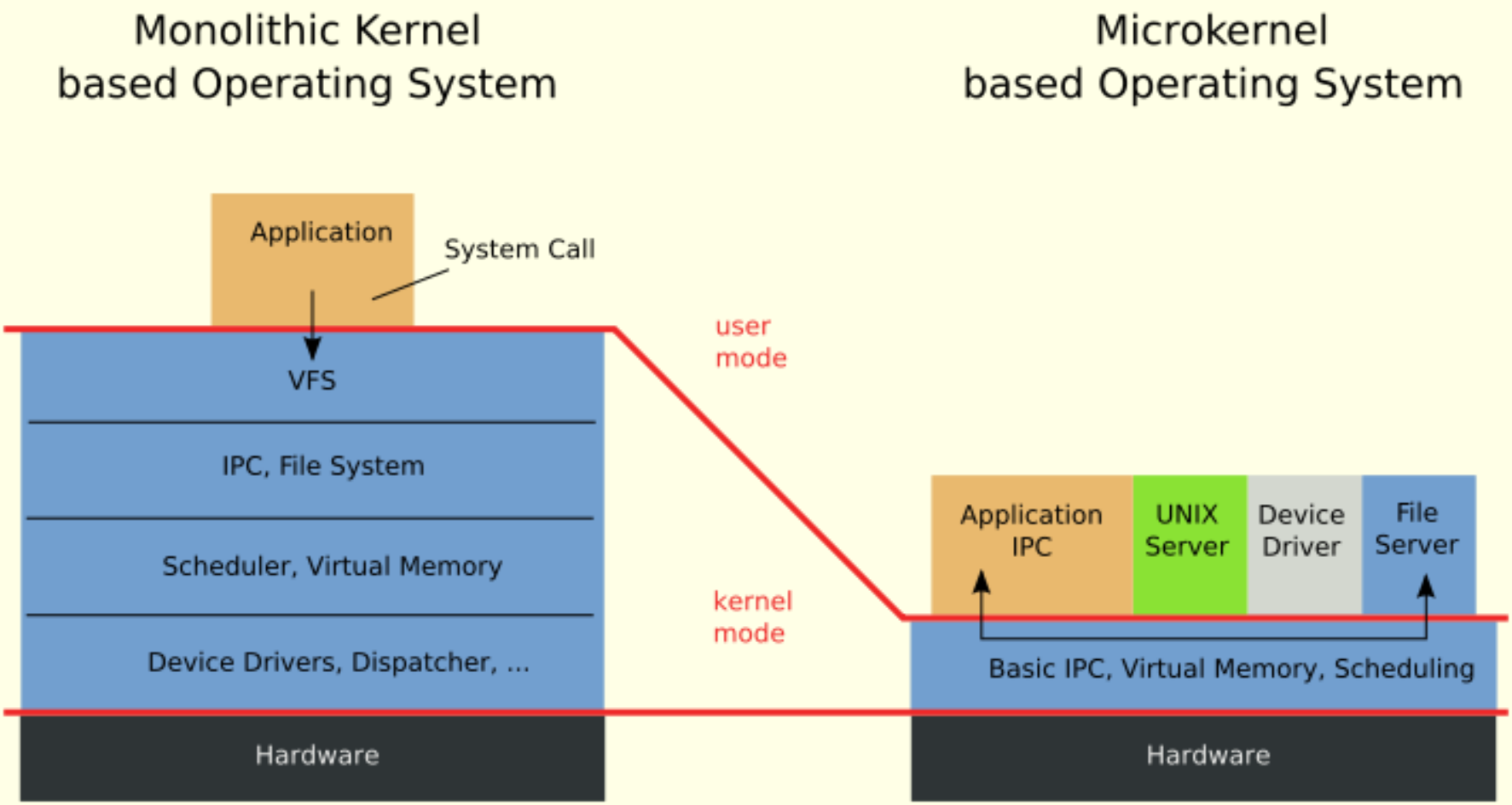
- Programador debe enfrentarse con toda la problemática identificada
  - solo con la ayuda del *runtime* del lenguaje

- Alternativas en el diseño de app que gestiona múltiples dispositivos
  - Ejecutivo cíclico con espera activa y *polling* de dispositivos
  - Ejecutivo cíclico con interrupciones
    - Minimizar duración rutinas de interrupción
  - Aplicación concurrente si *runtime* del lenguaje lo permite
    - Ada, Java,…

Programación de dispositivos en sistemas con SO

- SO ofrece enorme soporte para desarrollo de nuevos manejadores
  - API para su desarrollo
  - Abstracciones proceso y *thread* como modelo de concurrencia
- Tipo de arquitectura del SO:
  - Basado en un micronúcleo (p.e. Mach, QNX, Google Fuchsia):
    - Manejadores de E/S fuera del SO (ejecutando modo usuario)
  - Monolítico (p.e. familia UNIX, Linux):
    - Manejadores de E/S dentro del SO (ejecutando modo privilegiado)
    - SO monolíticos actuales con módulos cargables
      - Permite adaptar núcleo a plataforma (p.ej. sistema empotrado)
      - Posibilita técnicas como *hot-plugging*

Monolítico versus micronúcleo (wikipedia)



Manejador de dispositivo (*Device Driver*)

- Módulo que gestiona una clase de dispo. (varias unidades)
  - Esconde particularidades específicas de cada clase
  - Provee interfaz común independiente de disp. a resto de sistema
- Manejador en sistemas monolíticos:
  - Módulo que se enlaza con el resto del SO
    - De forma estática o de forma dinámica (módulo cargable)
  - Proporciona acceso a dispositivos mediante llamadas al sistema
  - Puede hacer uso de todas funciones internas del SO
  - Resto del SO puede usar sus funciones
- Manejadores en sistemas basados en micronúcleos
  - Procesos de usuario que reciben mensajes
- Presentación se centra en sistemas monolíticos (Linux)
  - Aunque al final se recogen aspectos específicos de micronúcleos

Desarrollo manejadores en sistemas monolíticos

- Se trata de una labor de programación pero con peculiaridades...
- Se usa una versión "recortada" de la biblioteca del lenguaje
  - P.e. en caso de C no debe incluir fread pero sí strcmp
- Pila de tamaño limitado (p.e. 8K) y sin control de desbordamiento
  - Evitar vars. de pila grandes y mucho anidamiento de llamadas
- Economía en el uso de memoria: no es memoria virtual
- Error en código → cuelgue del sistema
  - Opción: desarrollo sobre máquina virtual
- Difícil depuración (errores no reproducibles al tratar con HW):
  - Falta herramientas equivalentes a depuradores de aplicaciones
  - Habitual: Imprimir por consola y análisis de trazas
- Tipo de desarrollo con baja productividad y propenso a errores

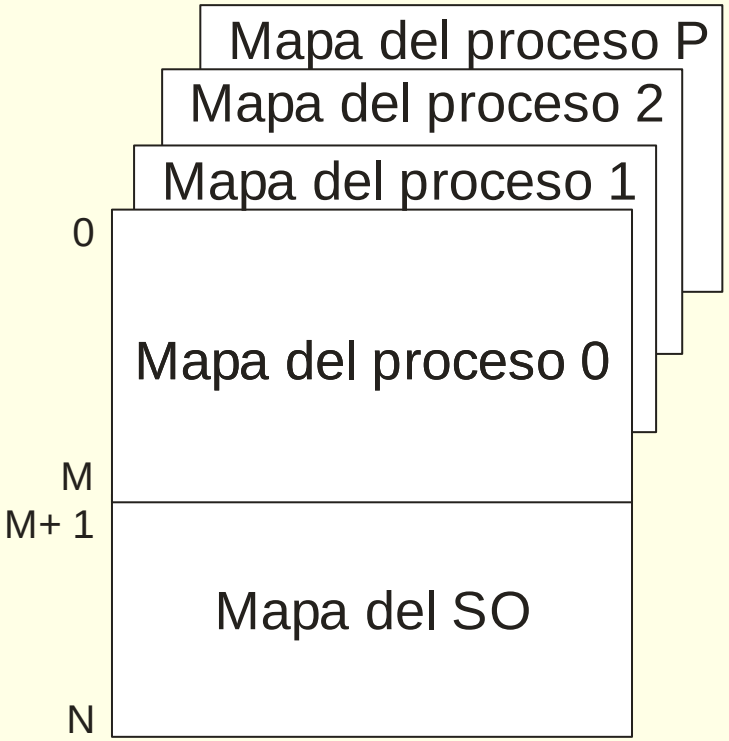
¿Manejadores en modo usuario en monolíticos?

- Mejor si se puede manejar dispo. desde software en modo usuario
  - Todas las ventajas del micronúcleo
- ¿Cómo acceder a E/S desde modo usuario?
- En Linux:
  - PIO: iopl, ioperm, inb, inw, outb, outw,…
  - MMIO: mmap de /dev/mem
- Pero...
  - No es posible manejo de interrupciones
  - Y no suficiente eficiente para dispositivos de altas prestaciones
    - Sobrecarga por cambios de modo y de contexto, código y datos del manejador expulsables de memoria (posible uso de mlock), …

Contextos de ejecución

- SO programa dirigido por eventos:
  - Una vez iniciado el sistema, solo se ejecuta código del SO si:
    - Interrupción, llamada al sistema, excepción (p.e. fallo de página)
      - Int. es asíncrona, llamada (y excepción) síncrona

- Modelo de memoria del proceso



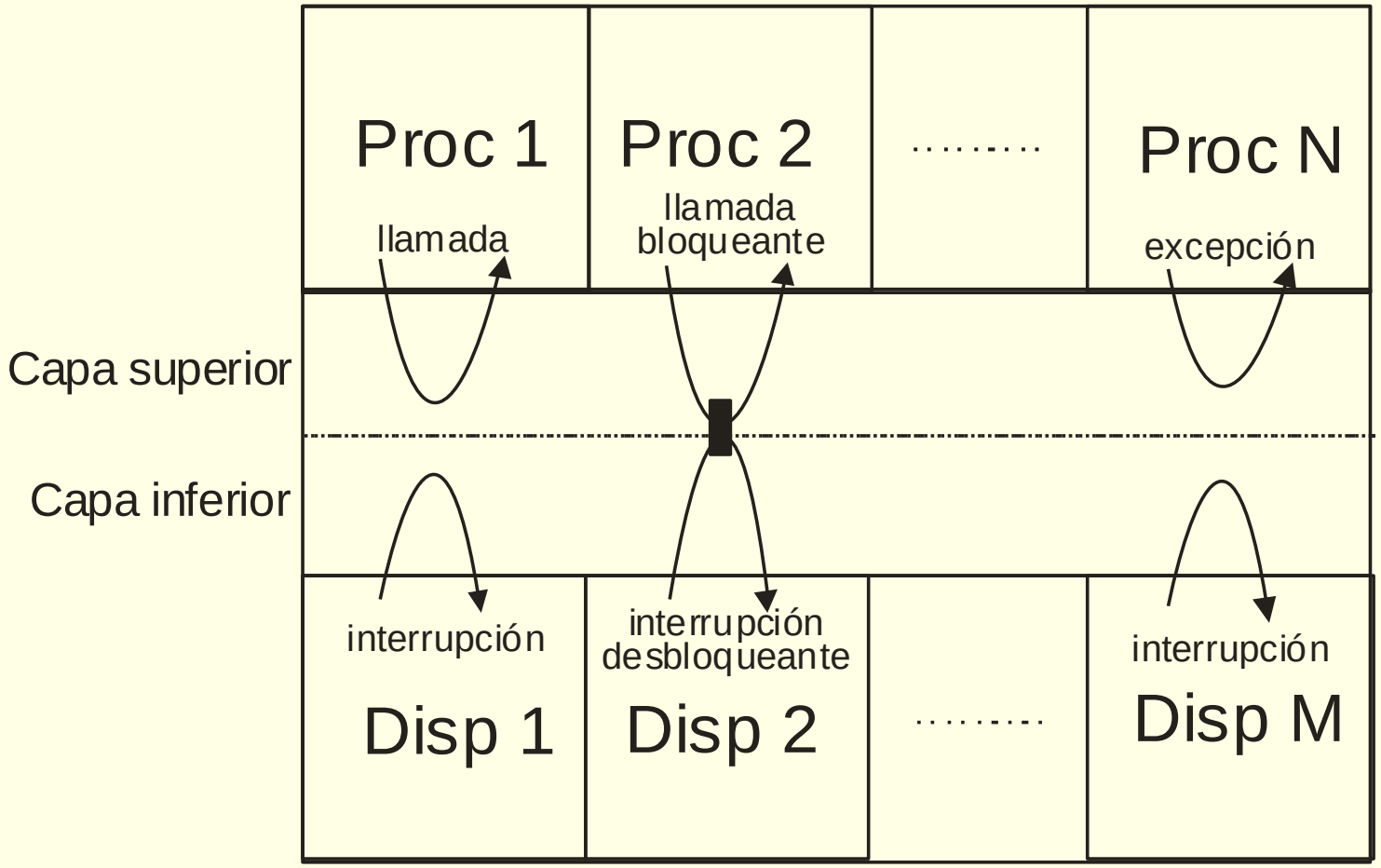
Contexto de ejecución de funciones del manejador

- Importante distinción entre funciones del manejador:
  - Su contexto de ejecución
- Funciones de acceso al dispo. (abrir, leer, escribir, cerrar, ...)
  - Ejecución en el contexto del proceso “solicitante”
  - Se puede acceder a mapa de memoria de proceso actual
  - Se puede realizar una operación de bloqueo
- Funciones de interrupción
  - Ejecución en contexto de proceso no relacionado con la int.
  - No se puede acceder a mapa de memoria de proceso actual
  - No se puede realizar una operación de bloqueo

Operación de lectura errónea

```
char *dirb;
tipo_cola_procesos cola_espera_entrada;
int lectura(char *dir, int tam) {
    dirb = dir;
    while (tam--) {
        out(R_CONTROL_X, LECTURA);
        Bloquear(cola_espera_entrada_X);
    }
}
void interrupcion_entrada_X() {
    *(dirb++) = in(R_DATOS_LEC_X);    /*  ERROR: acceso a mapa usuario
                                          desde rutina de interrupción */
    Desbloquear(cola_espera_entrada);
}
```

Organización del SO



Diseño de manejador de dispositivo de caracteres

- Pautas de diseño usando un dispositivo de salida hipotético X:
  - Opera en modo carácter, dirigido por interrupciones y sin DMA
    - dato → r. datos; orden → r.control; interrupción → fin operación
- **<u>No se tienen en cuenta aspectos de sincronización</u>**
- Objetivos del manejador hipotético:
  - Minimizar cambios modo (Usuario→Sistema | Sistema→Usuario)
  - Minimizar cambios de contexto (cambios de proceso)
  - Maximizar paralelismo entre SW y HW:
    - Operación concurrente de aplicación y dispositivo
    - Esquema productor-consumidor concurrente:
      - App. escritora produce datos; dispositivo los consume
  - Priorizar uso de dispositivo sobre ejecución de aplicaciones:
    - Dispo. sirve peticiones aunque en ejecución proc. alta prioridad

Versión escritura sin *buffering*: ineficiente

```
tipo_cola_procesos cola_espera_salida;
int escritura(char *dir, int tam) {
    while (tam-- > 0) {
        out(R_DATOS, *dir++);  // puede causar un fallo de página
        out(R_CONTROL, ESCR);
        Bloquear(&cola_espera_salida); // demasiados cambios de contexto: 1/byte
    }
}
void interrupcion_salida_X() {
    Desbloquear(&cola_espera_salida);
}
```

// Déficit: entre bytes, dispo. "parado" hasta que vuelva a ejecutar el proceso

// Solución: programación de dispositivo en rutina de interrupción

Versión escritura con *buffer* de N bytes

```
tipo_buffer buf;
tipo_cola_procesos cola_espera_salida;
int tam_datos;
int a_esc;
int escritura(char *dir, int tam) {
    tam_datos = tam;
    while (tam_datos>0) {
        a_esc = (min(tam_datos, tam(&buf)));
        copiar_de_usuario_a_buf(dir, &buf, a_esc); // puede causar un fallo de página
        tam_datos -= a_esc;
        dir+= a_esc;
        programar();
        Bloquear(&cola_espera_salida); // nº cambios de contexto: tam/buf.tam
    }
}
```

Versión escritura con *buffer* de N bytes

```
void interrupcion_salida_X() {
    if (vacio(&buf))
        Desbloquear(&cola_espera_salida);
    else
        programar();
}

void programar(){
    char c = extraer(&buf); // buf.nelem--
    out(R_DATOS, c);
    out(R_CONTROL, ESCRITURA);
 }
```

// Mejora: entre bytes, dispo. sigue operando aunque no ejecute el proceso

// Déficit: si varias llamadas a *escritura*, entre llamadas dispositivo "parado"

// Solución: escritura diferida: llamada a *escritura* termina al copiar datos a *buffer*

Versión escritura diferida

```
tipo_buffer buf;
tipo_cola_procesos cola_espera_salida; int tam_datos, a_esc, dispo_activo=0;
int escritura(char *dir, int tam) {
    tam_datos = tam;
    while (tam_datos>0) {
        if (lleno(&buf))
            Bloquear(&cola_espera_salida);
        a_esc = (min(tam_datos, espacio_libre(&buf)));
        copiar_de_usuario_a_buf(dir, &buf, a_esc); // puede causar un fallo de página
        tam_datos -= a_esc;
        dir+= a_esc;
        if (!dispo_activo) {
            dispo_activo = 1;
            programar();
        }
    }
}
```

Versión escritura diferida

```
void interrupcion_salida_X() {
    if (vacio(&buf))
        dispo_activo = 0;
    else
        programar();
    if (!vacia(cola_espera_salida) && (espacio_libre(&buf)>= min(tam_datos, tam(&buf))))
        Desbloquear(&cola_espera_salida);
}
void programar(){
    char c = extraer(&buf); // buf.nelem--
    out(R_DATOS, c);
    out(R_CONTROL, ESCRITURA);
}
```

ALTERNATIVA

```
// despertar antes al proceso: en cuanto haya hueco de tamaño superior a un cierto umbral
    if (!vacia(cola_espera_salida) && (espacio_libre(&buf)>=min(tam_datos, UMBRAL))
// intenta evitar buffer vacío si hay escritor y permite paralelismo entre copia y dispositivo
```

Estructura interna del manejador

- Manejador exporta al resto del SO funciones para:
  - Iniciarse y terminar (al cargarse y descargarse)
  - Añadir y eliminar los dispositivos a manejar si PnP
    - Invocadas en conexión y desconexión del dispositivo
  - Ofrecer a las aplicaciones acceso a los dispositivos
  - Tratar interrupciones de dispos. e interrupciones software
- Manejador usa API interno ofrecido para:
  - gestión de bloqueos, acceso a dispositivos,
  - uso de memoria, uso de DMA, control de tiempo
  - sincronización, concurrencia, ...
- Presentamos primero manejadores no PnP y después PnP
  - Al final, el API interno

Ciclo de vida de manejador dispositivos no PnP

- Enlazado estáticamente con SO o cargable en tiempo de ejecución
  - En Linux configurable al generar la imagen SO
  - Normalmente, recomendable que sea módulo cargable
    - Incluso aunque sea para dispositivos no PnP
  - Ofrece funciones para su inicio (module_init) y fin (module_exit)
- Carga módulo dinámico que maneja dispositivos no PnP:
    - Carga del manejador como parte del arranque del SO
    - Carga a petición del superusuario (insmod)
- Descarga módulo dinámico que maneja dispositivos no PnP:
    - Descarga a petición del superusuario (rmmod)
    - Descarga del manejador como parte de parada del SO

Dispositivos en UNIX

- Distingue entre dispositivos de bloques, caracteres y red
  - Dispositivos de bloques y caracteres mismo esquema de acceso
    - Nos centramos en dispositivos de caracteres
- Cada manejador tiene un ID único interno (*major*) por cada tipo
- Y recibe como argumento de sus ops. un nº unidad (*minor*)
- SO ofrece a aplicaciones fichero especial (por convención en /dev)
    - /dev/nombre → car. o bl. + *major* + *minor* → (manejador + unidad)
  - $ ls -l /dev/sda1 /dev/sda2 /dev/tty0 /dev/tty1
  - brw-r----- 1 root disk 8, 1 nov 23 09:47 /dev/sda1
  - brw-r----- 1 root disk 8, 2 nov 23 09:47 /dev/sda2
  - crw-rw---- 1 root root 4, 0 nov 23 09:47 /dev/tty0
  - crw------- 1 root root 4, 1 nov 23 08:47 /dev/tty1

Func. inicial manejador no PnP: recursos HW

- Manejador conoce a priori qué dispositivos gestiona
  - Y qué recursos hardware requieren
  - Esa información debería recibirla como parámetro
    - O mejor como *Device Trees* (véase más adelante)
  - No deberían incluirse datos fijos en su código
- PIO: manejador debe indicar rango puertos usados (request_region)
  - Permite que núcleo detecte conflictos entre manejadores
- MMIO:manejador indica rango dir E/S usadas (request_mem_region)
  - Permite que núcleo detecte conflictos entre manejadores
  - Solicita crear rango dir. lógicas asociadas a las físicas (ioremap)
- Instala manejadores de interrupción (request_irq)

Func. inicial manejador no PnP: recursos SW

- Reserva IDs internos para dispositivos (UNIX *major* y *minor*)
  - *Major* seleccionado por manejador (register_chrdev_region)
    - Debería recibirlo como parámetro
  - O por el núcleo (alloc_chrdev_region)
- Registra dispositivo de caracteres (cdev_init y cdev_add)
  - Especificando ops. acceso al dispositivo (struct file_operations)
- Hace visible cada dispositivo a aplicaciones de usuario. Linux:
  - Añadirlo a *sysfs* : class_create y device_create
  - Demonio de sistema (udev) detecta nueva entrada en *sysfs*
  - Crea fichero especial con *major* y *minor* asignados
  - Versiones antiguas: creación “manual” con mknod

Devres API

- Manejador va reservando y liberando muchos recursos
  - Es fácil olvidarse de liberar alguno
    - Sobretodo al tratar condiciones de error
- API alternativo para algunas funciones que “recuerda” las reservas
  - Y las libera automáticamente al cerrar el dispositivo
  - Algunos ejemplos:
    - devm_request_irq
    - devm_ioremap
    - devm_request_region
    - devm_kzalloc

Funciones acceso al dispositivo struct file_operations

- Ops. manejador proporciona a aplicaciones para acceso a dispos.
  - Abrir, leer, escribir, operaciones control (p.e. rebobinar cinta)
  - Pueden bloquear al proceso que las invoca si es preciso
  - Suelen ser las mismas para todos los dispos. de un manejador
    - Contraejemplo de Linux: manejador mem (/dev/zero, /dev/null, …)
- Todos los manejadores ofrecen misma API
  - ¿Cómo incluir operaciones de control?
    - En UNIX función única "cajón de sastre" (ioctl)
- struct file_operations
  - https://elixir.bootlin.com/linux/v4.18.11/source/include/linux/fs.h#L1713
- Funcionalidad principal en las operaciones de lectura y escritura
  - Variedad de tipos de operaciones de lectura/escritura

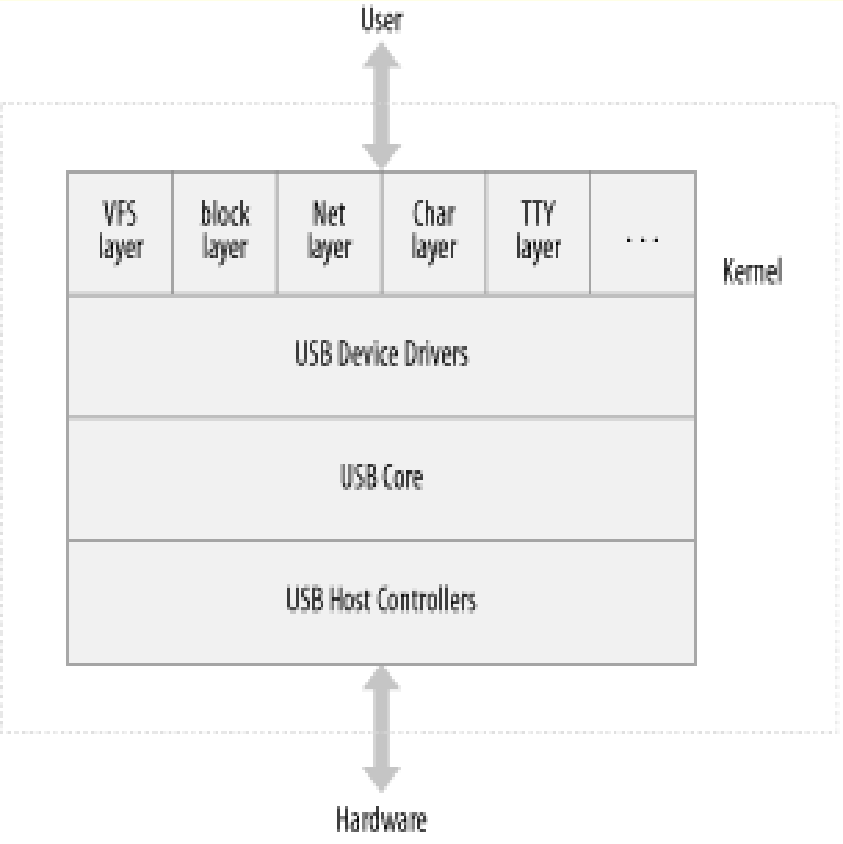
Operaciones de lectura y escritura

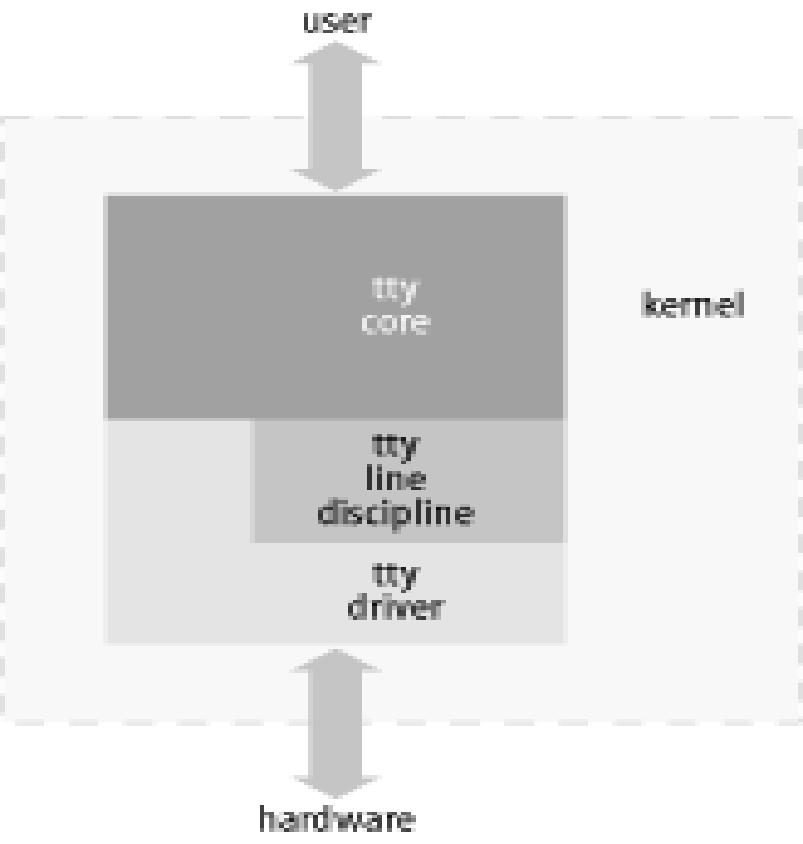
- Convencionales:
  - Programa especifica *buffer* para leer/escribir
  - SO devuelve control cuando datos leídos o escritos
  - Manejador puede usar *buffer* interno como almacén temporal
- No bloqueantes
  - Si operación no puede completarse inmediatamente → error
- Sin *buffers* intermedios
  - disp ↔ espacio de usuario
- Asíncronas
  - Manejador inicia oper. y SO devuelve control inmediatamente
- Con *scatter-gather* a nivel de usuario
  - Programa puede especificar varios *buffers*
- Uso de mmap en vez de read y write

Jerarquía de manejadores de dispositivos

- Conjunto de manejadores no es plano sino jerárquico
  - Código común y necesidad de apilamiento de manejadores
  - Manejadores de buses y de dispositivos
- Ej: webcam USB requiere man. de buses y del propio dispositivo
  - Manejadores de buses:
    - Manejador bus PCI
      - Manejador general de PCI + específico del *HB* PCI
    - Manejador del HC USB conectado a PCI
      - Manejador chip específico + manejador OHCI, EHCI, UHCI
    - Manej. clase HUB de dispo. USB (interacción con *root hub*)
  - Dispo. USB/UVC y cámara: manejador se apoyará en:
    - Manejador de funcionalidad común de todos los disp. USB
    - Manejador específico de clase VIDEO de dispositivo USB
    - Manejador general para todas las cámaras (V4L2)

Ejemplos de jerarquías de manejadores (LDD)





Ciclo de vida de manejador dispositivos PnP

- Especifica qué dispositivos maneja (MODULE_DEVICE_TABLE)
  - Durante compilación módulos recolecta esta info. (depmod)
- Carga del módulo cuando se detecta el dispositivo
  - **Manejador de bus** descubre/configura dispositivo
    - En arranque (PnP) o al conectar el dispositivo (*hot-plugging*)
    - Genera evento de descubrimiento hacia modo usuario
  - Proceso de usuario (udevd) lo recibe
    - Consulta información recolectada → módulo manejador
    - Si todavía no cargado (primer dispositivo), lo carga
    - Llama a la función para añadir un dispositivo (probe)
- Descarga del módulo (además de al parar el SO)
  - **Manejador de bus** descubre desconexión de dispositivo
    - Genera evento de desconexión hacia modo usuario
  - Proceso udevd llama a función para eliminarlo (remove)
    - Descarga módulo si último dispositivo

Aspectos específicos manejadores dispo PnP

- Función inicial: no añade dispositivo
  - Registra funciones para añadir/borrar (probe/remove) dispositivo
    - pci_register_driver, usb_register, …
  - Serán invocadas cuando se descubran los dispositivos
- Función que añade dispositivo:
  - Determina su configuración leyendo mediante dir. geográfico
    - Direcciones MMIO, puertos PIO, líneas de interrupción,…
- Ejemplo manejadores de dispositivos PnP en PCI y USB:
  - *https://elixir.bootlin.com/linux/latest/source/drivers/usb/host/ehci-pci.c*
  - *https://elixir.bootlin.com/linux/latest/source/drivers/media/usb/uvc/uvc_driver.c*
- Información recopilada durante la compilación:
  - *cat /lib/modules/`uname –r`/modules.alias*

Ejemplo práctico: buscando manejador de dispo.

- lspci en Ubuntu con Virtualbox encuentra disco AHCI SATA

00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)

$ lspci –n → 00:0d.0 0106: 8086:2829 (rev 02) → vendedor 8086; ID dispositivo 2829

- Buscamos el ID dispositivo entre la información recopilada

$ grep 2829 /lib/modules/`uname –r`/modules.alias

alias pci:v00008086d00002829sv*sd*bc*sc*i* ahci

- Comprobamos que el manejador declara que maneja ese modelo

https://elixir.bootlin.com/linux/v4.19/source/drivers/ata/ahci.c#L241

```
static const struct pci_device_id ahci_pci_tbl[] = {
 { PCI_VDEVICE(INTEL, 0x2652), board_ahci }, /* ICH6 */ …..
 { PCI_VDEVICE(INTEL, 0x2829), board_ahci }, /* ICH8M */ ….
```

Device Trees

- Sist. dedicados no incluyen mecanismos "descubrimiento" del HW
- ¿Cómo sabe el SO las características del sistema?
  - Procesadores, memoria, dispositivos (MMIO/PIO, interrup.,…)
- Código SO tiene fichero con definiciones por cada plataforma
  - En arranque recibe algunos parámetros adicionales
- Alternativa más flexible: *Device Trees (DT)*
  - Fichero de texto con descripción del HW de la plataforma
  - En arranque, además de SO, se carga versión binaria del *DT*
  - SO recibe dirección de *DT* y lo lee para conocer el sistema

    https://elinux.org/Device_Tree_Usage
- *DT overlay*:
  - Permite construir el *DT* de forma incremental
  - Pueden añadirse en tiempo de ejecución

Device Trees y manejadores de plataforma

- Manejador de plataforma
  - Manejador no realmente P&P pero con modo operación P&P
    - con funciones probe/remove;
    - se carga al descubrir 1er dispositivo
- Útil para gestionar *DT*
- Manejador indica qué dispositivos maneja (MODULE_DEVICE_TABLE)
  - SO lee *DT* y carga manejadores de plataforma requeridos
  - Campo .compatible de DT = .compatible de manejador
- Ejemplo:

  https://elixir.bootlin.com/linux/v4.19/source/arch/powerpc/platforms/pasemi/gpio_mdio.c#L290

```
static const struct of_device_id gpio_mdio_match[] ={     {
      .compatible      = "gpio-mdio",   },    {}, };
MODULE_DEVICE_TABLE(of, gpio_mdio_match);
```

API del SO usado por los manejadores

- Sincronización
- Acceso a registros de los dispositivos
- Gestión de bloqueos
- Soporte a las necesidades de memoria del manejador
- Control de tiempo
- Soporte GPIO
- Interrupciones software
- Creación de procesos/hilos de núcleo

Sincronización

- Tipos de problemas de sincronización:
  - Producidos por tratamiento interrupciones
  - Debidos a ejecución concurrente de procesos
    - Ejecución entremezclada de procesos en un procesador
    - Ejecución paralela real de procesos en un multiprocesador
- Es necesario crear secciones críticas (SC) dentro del manejador
- SO ofrece cuatro mecanismos para crear SC
  - Inhibir las interrupciones (local_irq_disable)
  - Inhibir la expulsión de procesos (preempt_disable)
  - *Spinlocks*: espera activa basada en operaciones de tipo *test&set*
    - Convencionales (spin_lock_init) o lectores/escritores (rwlock_init)
    - Linux también ofrece *seqlocks* y *RCU-locks*
  - Semáforos/*mutex*: espera bloqueante
    - Convencionales (sema_init) o lectores/escritores (init_rwsem)

Uso de los mecanismos de sincronización

- Sincronización entre llamada/rutina de int. y otra rutina de int.:
  - Ambas usan *spinlock*
  - Rutina interrumpida además inhibe localmente int (spin_lock_irq)
    - No válido semáforos: rutina de interrupción no puede bloquearse
- Sincronización entre llamadas concurrentes
  - Si SC muy corta y sin bloqueos:
    - *spinlock* + expulsión de procesos inhibida
    - En Linux spin_lock incluye preempt_disable
  - En caso contrario: semáforos/mutex proporcionados por SO
    - semáforo internamente usa *spinlock* + expulsión inhibida
- Cuestión de diseño: granularidad de la sincronización
  - Mayor la zona protegida por un elemento de sincronización
    - Menor paralelismo pero también menor sobrecarga

Acceso a registros del dispositivo

- Funciones de acceso al dispositivo del manejador lo requieren
- Acceso PIO
  - SO debe proveer macros portables
  - Evita uso ensamblador en manejador
    - inb, inw, inl, outb, outw, outl, …
- Acceso MMIO: uso de dirección lógica obtenida a partir de ioremap
  - En principio, podría usarse el puntero directamente
  - Pero en Linux desaconsejado: Problemas en algunas UCP
    - Y sobretodo mejor que se identifiquen esos accesos en el código
    - ioread8, ioread16, ioread32, iowrite8, iowrite16, iowrite32…
- SO proporciona barreras de memoria:
  - Para compilador: barrier; Para HW (y compilador): mb

Gestión de bloqueos en Linux

- Funciones de acceso al dispositivo del manejador pueden requerirlo
  - Recordatorio: Interrupciones solo pueden desbloquear
- Ofrece diversas funciones para bloquear/desbloquear procesos

- void **wait_event**(wait_queue_head_t q, int condition);
- int **wait_event_interruptible**(wait_queue_head_t q, int condition);
- int **wait_event_timeout**(wait_queue_head_t q, int condition, int time);
- int **wait_event_interruptible_timeout**(wait_queue_head_t q, int condition, int time);
- void **wake_up**(struct wait_queue **q);
- void **wake_up_interruptible**(struct wait_queue **q);
- void **wake_up_nr**(struct wait_queue **q, int nr);
- void **wake_up_interruptible_nr**(struct wait_queue **q, int nr);
- void **wake_up_all**(struct wait_queue **q);
- void **wake_up_interruptible_all**(struct wait_queue **q);
- void **wake_up_interruptible_sync**(struct wait_queue **q);

Soporte necesidades de memoria del manejador

- SO debe ofrecer diversas funciones para reservar memoria:
  - Reserva tipo "malloc" (kmalloc)
  - Reserva de páginas contiguas (alloc_pages)
  - Reserva espacio físicamente no contiguo pero sí lógica (vmalloc)
  - Soporte para crear cachés de objetos (kmem_cache_create)
    - Para manejador que reserva y libera mismo tipo de objeto
- Petición de memoria dentro de rutina de interrupción
  - Se deben usar funciones de reserva que no puedan bloquear
    - kmalloc con *flag* GFP_ATOMIC
- Acceso mapa de usuario (copy_from_user | copy_to_user)
  - No se pueden usar desde contexto asíncrono
  - Puede causar fallo de página pero eso es transparente a manej.

Soporte GPIO

- API "tradicional" basado en nº de cada pin
  - int gpio_request(unsigned gpio, const char *label)
  - int gpio_free(unsigned gpio)
  - int gpio_direction_input(unsigned gpio)
  - int gpio_direction_output(unsigned gpio)
  - int gpio_get_value(unsigned gpio)
  - int gpio_set_value(unsigned gpio, int value)
  - unsigned gpio_to_irq(unsigned gpio)
  - unsigned irq_to_gpio(unsigned irq)
  - Manejador debe implementar esas funciones
  - Otras partes del SO pueden usarlas
  - Programas usuario acceden a través de *sysfs* o */dev/gpiochipN*
- API "moderno" usa descriptores (p.e. "reset" en vez de pin 8)
  - int gpiod_direction_input(struct gpio_desc *desc)

Control del tiempo

- SO ofrece diversas funciones relacionadas con el tiempo:
  - Temporizadores basados en int. de reloj
    - Manejador requiere realizar una actividad periódica
    - Asocia una función suya con temporizador (add_timer)
      - Función ejecutará en contexto asíncrono (en una interrupción SW)
  - Funciones de espera por un plazo de tiempo
    - Espera bloqueante:
      - solo por tiempo (msleep, schedule_timeout)
      - Por un evento y por tiempo (wait_event_timeout)
    - Espera activa:
      - solo para esperas brevísimas (ndelay, udelay)
      - SO usa un bucle precalculado o basado en TSC

Interrupciones de dispositivo e int. software

- No todas las operaciones asociadas a interrupción son urgentes
- Importante minimizar duración de rutinas de interrupción
  - Mientras algunas interrupciones están inhibidas
- Ej. interrupción teclado:
  - urgente leer código de tecla; no urgente averiguar car. pulsado
- Rutina interrupción realiza operaciones urgentes
  - No urgentes ejecutan en contexto con interrupciones habilitadas
- Mecanismo de int. software: int. mínima prioridad pedida por SW
  - Rutina int. realiza operaciones urgentes y activa int. software
  - Tratamiento de interrupción SW $\rightarrow$ ops. no urgentes
    - En Linux *softirq (tasklet)* ; En Windows DPC

Uso de procesos/hilos de núcleo

- Manejador solo se activa cuando se invocan sus funciones
- En ocasiones puede requerir estar activo aunque no sea invocado
  - Puede crear proceso de núcleo que ejecuta en su propio contexto
- Proceso/hilo de núcleo: es un proceso más en el sistema pero
  - Ejecuta solo código del SO
  - No tiene mapa de memoria de usuario asociado
  - Puede realizar operaciones de bloqueo
  - Pero no acceder a direcciones de memoria de usuario
- Para evitar proliferación de procesos de núcleo
  - Colas predefinidas de trabajos servidas por procesos de núcleo
    - En vez de crear un nuevo proceso de núcleo, se encola trabajo
    - Linux workqueues

Definición de contexto atómico

- Como recapitulación sobre los contextos de ejecución
- Contexto atómico si se cumple **alguna** de estas condiciones:
  - Rutina de interrupción de un dispositivo
  - Rutina de interrupción software
  - Prohibidas las interrupciones de los dispositivos
  - Inhibidas las interrupciones software.
  - Deshabilitada la expulsión de procesos
  - En posesión de un *spinlock*
- solo se puede hacer un bloqueo
  - Si en contexto **no atómico**
- solo se puede acceder a mapa de usuario
  - Si en contexto **no atómico y no se trata de proceso de núcleo**

Desarrollo de manejadores en micronúcleos

- Se eliminan “peculiaridades” en su desarrollo
  - Biblioteca del lenguaje completa
  - Uso de llamadas al sistema
    - Además de los servicios proporcionados por el micronúcleo
  - Depuración convencional
  - Error en manejador solo afecta al acceso a ese dispositivo
    - Puede activarse nueva versión del manejador sobre la marcha
  - Uso de memoria convencional
    - Memoria virtual y pila sin restricciones
- Mayor productividad y menor propensión a errores
- Menor eficiencia
  - Más paso de mensajes y cambios de proceso

Estructura del manejador en micronúcleos

- Programa servidor convencional (¡tiene su main!)
  - Bucle que espera mensajes
    - Ops. implementadas por manejador (lect., escr., ...) son mensajes
    - Interrupciones también como mensajes
    - Micronúcleo ofrece servicios para que proceso reserve IRQ
      - Cuando se produce int., micronúcleo envía mensaje a ese proceso
  - Al recibir mensaje, comprueba su tipo y lo procesa
  - El servidor puede ser concurrente
    - Sincronización igual que cualquier programa de usuario
  - Manejador realiza directamente accesos PIO/MMIO
    - Micronúcleo ofrece servicios para habilitar acceso a los mismos

Bibliografía

- *Linux Device Drivers*, Jonathan Corbet, Alessandro Rubini, y Greg Kroah-Hartman. O'Reilly Media, 3ª edición, 2005
- *Building Embedded Linux Systems*, Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, y Philippe Gerum, O'Reilly Media, 2ª edición, 2008
- *Understanding the Linux Kernel,* Daniel P. Bovet y Marco Cesati. O'Reilly Media, 3ª edición, 2005
- *Programming Embedded Systems*, Michael Barr y Anthony Massa. O'Reilly Media, 2006
- *Designing Embedded Hardware*, John Catsoulis, O'Reilly Media, 2005
- *Sistemas Operativos: Una visión aplicada*. J. Carretero, P. de Miguel, F. García y F. Pérez. McGraw-Hill, 2ª edición, 2007
- *Gestión de procesos*. F. Pérez Costoya. http://laurel.datsi.fi.upm.es/~ssoo/DSO4/gestion_de_procesos.pdf