S. empotrados y ubicuos

Programación de dispositivos

Fernando Pérez Costoya
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

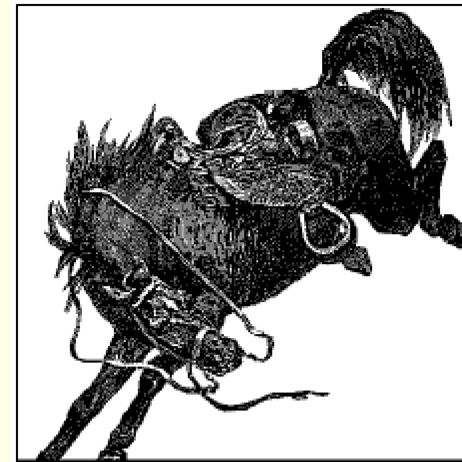
- Computador incluye dispositivos de E/S para:
  - Almacenamiento de información
  - Interacción con mundo exterior (usuarios | componentes físicos)
  - Comunicación con otros equipos
- Software de E/S muy complejo y heterogéneo
- Precisamente por eso surgieron los SS.OO.
  - Ofrecen interfaz uniforme 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**

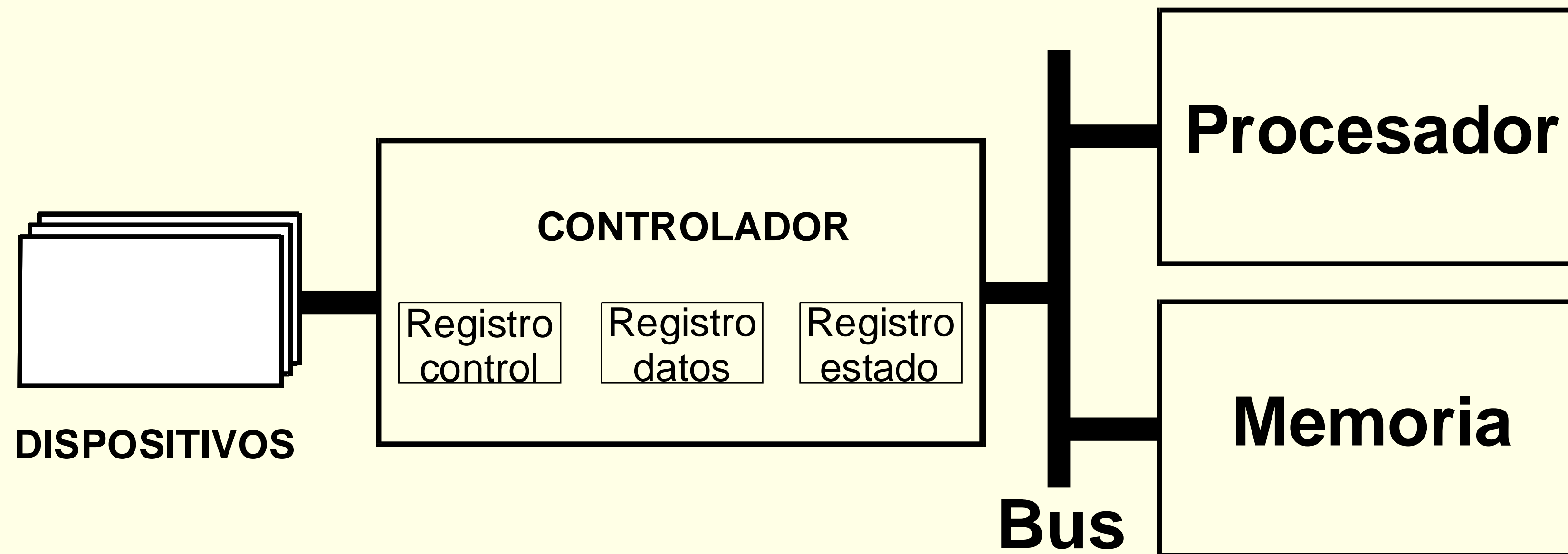
Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman



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 básico de dispositivo



- Se asume conocimientos básicos de arquitectura E/S del procesador
  - Por ejemplo, conceptos como:
    - E/S programada
    - E/S dirigida por interrupciones
    - E/S por DMA (reg. con dirección y tamaño de transferencia)

Memory-mapped I/O vs. Port I/O

- MMIO: Mismo espacio de dir. e instrucciones para mem. y E/S
- PIO: Distintos espacios de dir. e instrucciones para mem. y E/S
  - Bus incluye señal para discriminarlos (MEMREQ vs IOREQ)
  - Instrucciones de E/S específicas (IN/OUT) vs LOAD/STORE
- Ventajas de MMIO en la programación de dispositivos
  - No requiere ensamblador
- PIO no habitual excepto familia x86
  - Aunque también usa MMI
  - Linux: ficheros /proc/ioports y /proc/iomem

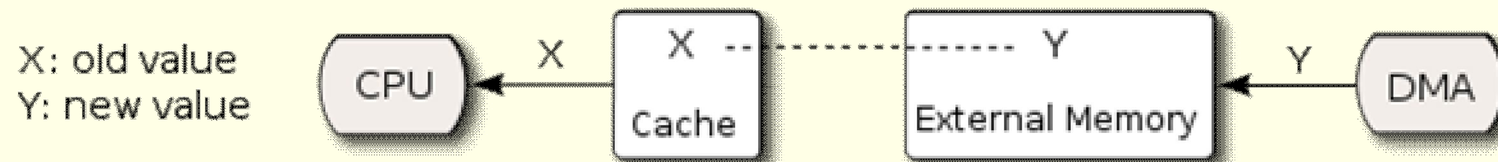
Interrupciones

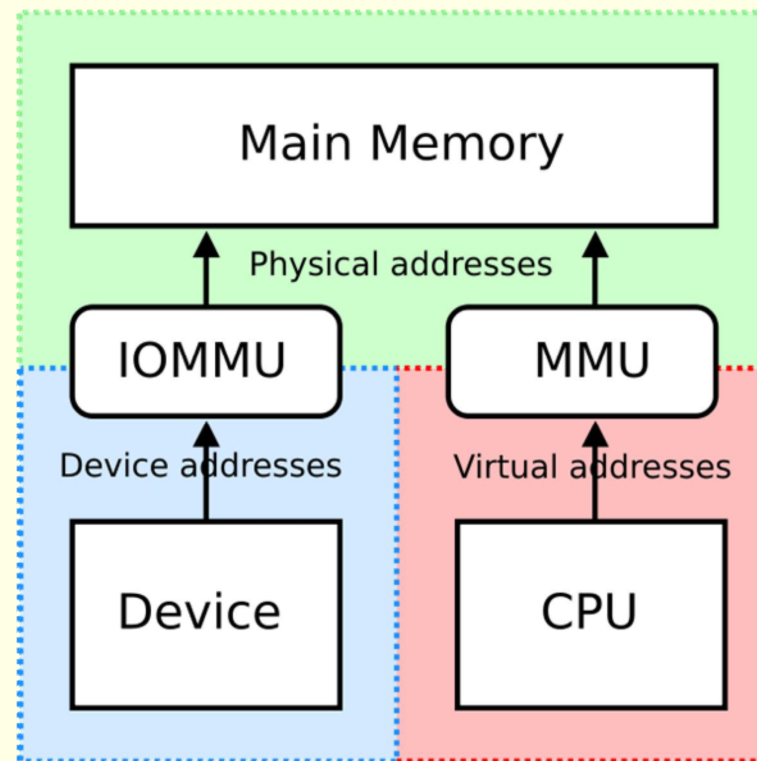
- Distintas alternativas de diseño hardware visibles desde software
  - Por flanco (*edge-triggered*) o por nivel (*level-triggered*)
  - Basadas en líneas de interrupción vs *message-signaled* (MSI)
    - MSI (*in band*): interrupción→escritura de valor en cierta dirección
  - Interrupciones compartidas: Múltiples dispositivos/línea
    - Dificultad para compartir interrupciones
      - Por flanco: se pueden mezclar o perder pulsos
      - Por nivel: si por error HW/SW dispo. no retira señal, int. bloqueadas
    - Comprobar estado de todos los dispositivos de la línea
  - Interrupciones en multiprocesador
    - Modalidad de distribución de interrupciones
      - Fija; *Broadcast*; *multicast*;
      - Turno rotatorio; por prioridad; a UCP más reciente; ...
  - Linux: fichero /proc/interrupts

DMA

- Distintas alternativas de diseño visibles desde el software
  - Controlador con o sin *scatter-gather DMA (vectored I/O)*
    - Controlador con múltiples pares (reg. dirección; reg. longitud)
  - Controlador con o sin coherencia entre caché y memoria
    - *Non-coherent DMA*: transferencias DMA no afectan a la caché
  - 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.
    - Facilita virtualización de la E/S
  - Limitaciones en rango acceso a memoria desde dispositivos
    - ISA sólo primeros 16MB
    - PAE y dispositivos con DMA de 32 bits: no más allá de 4GB
  - /proc/dma: canales DMA de bus ISA

Non-coherent DMA | IO-MMU (wikipedia)

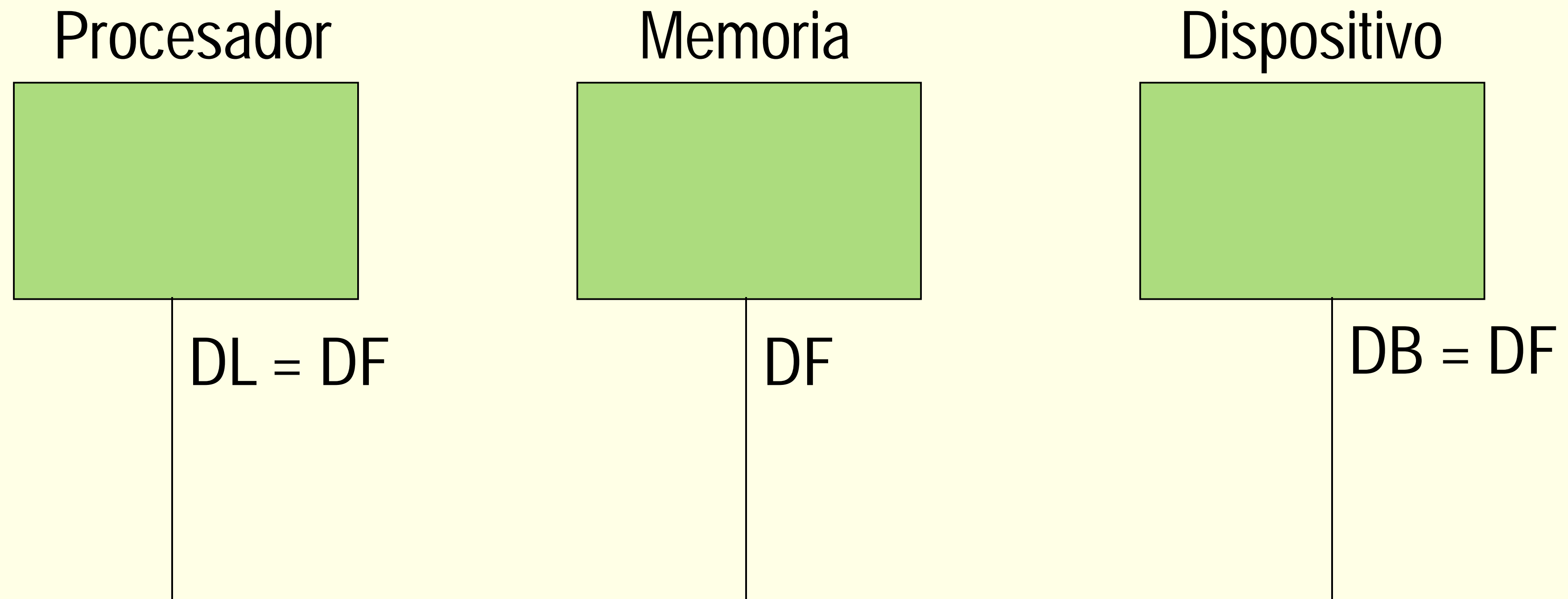




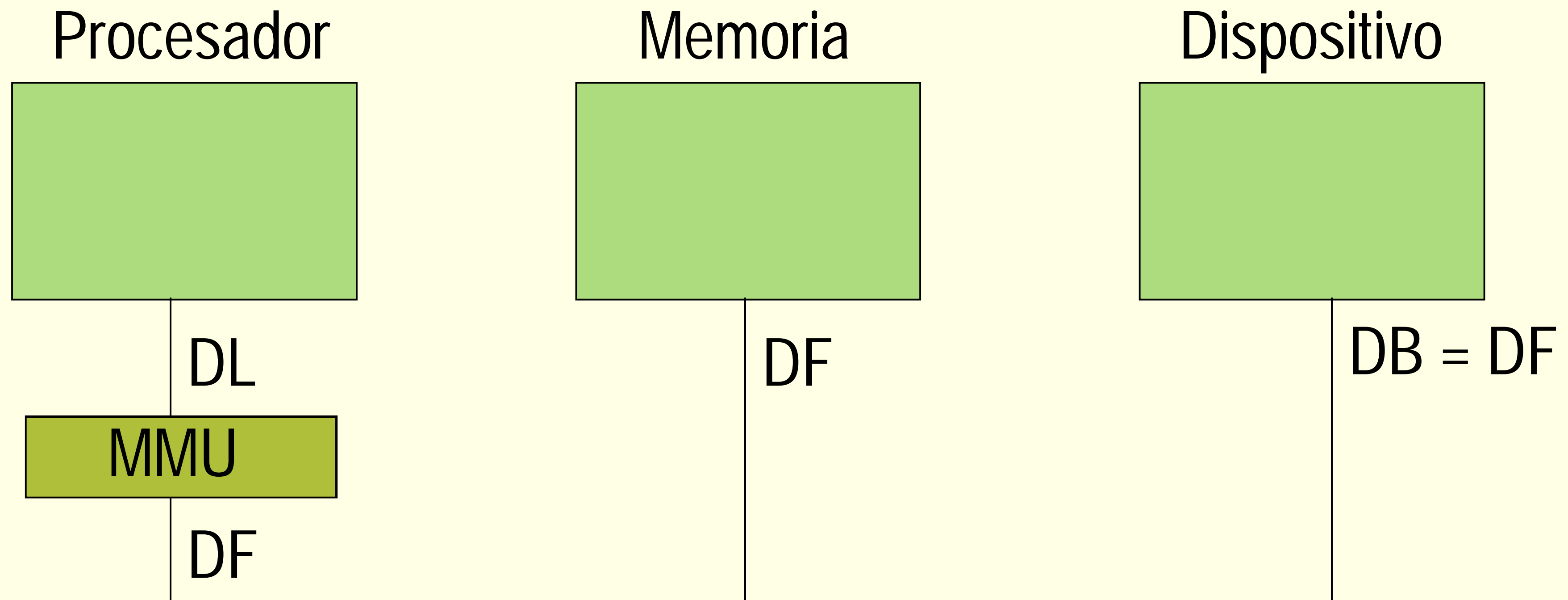
Tipos de direcciones

- 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
- ***http://www.makelinux.net/ldd3/chp-15-sect-1***

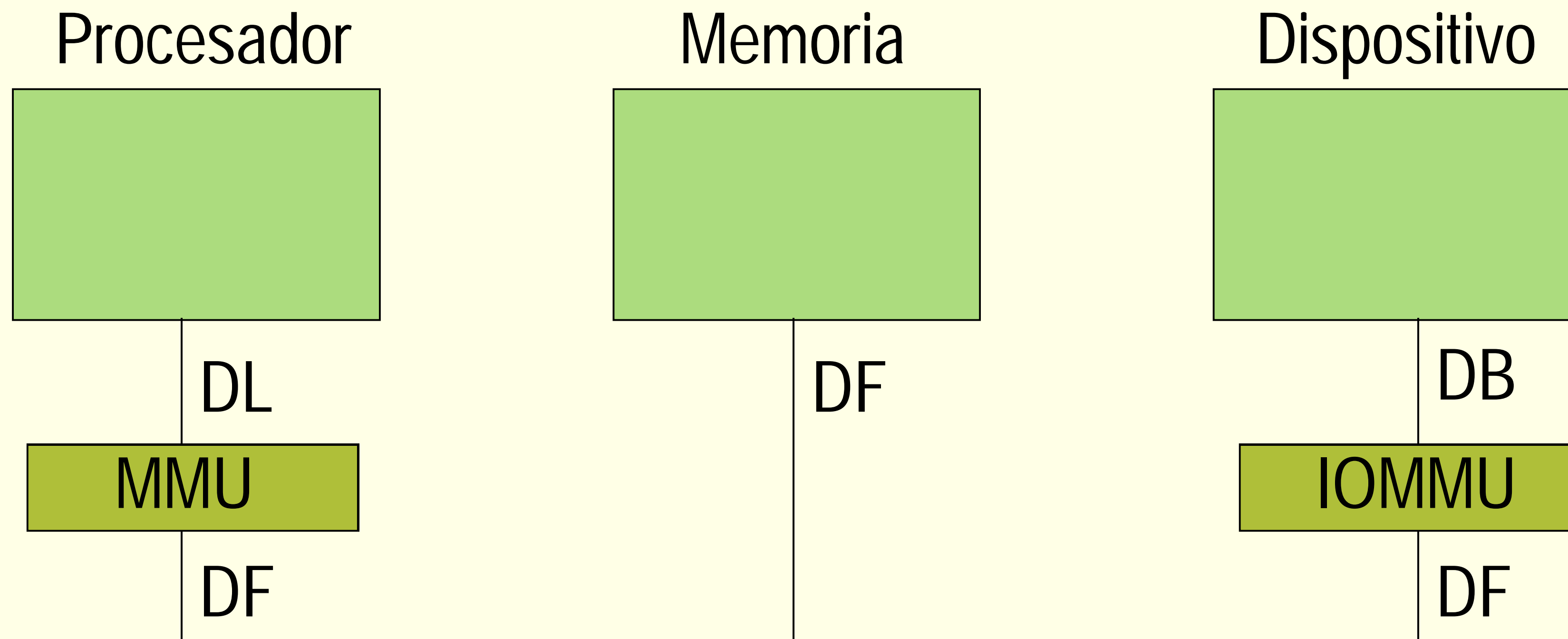
Sistema sin MMU ni IO-MMU



Sistema con MMU pero sin IO-MMU



Sistema con MMU e IO-MMU



Ejemplo 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 operación de escritura

Ejemplo en sistema sin MMU ni IO-MMU

```
uint32_t *dir;                  // para referenciar el registro 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
dir = X;                        // dirección del primer registro del dispositivo
*dir++ = &v;                    // dirección del buffer a escribir por DMA
*dir++ = sizeof(v);             // tamaño de los datos a escribir por DMA
*dir=1;                         // inicia escritura por DMA
```

Ejemplo en sistema con MMU pero sin IO-MMU

```
uint32_t *dir;                  // para referenciar el registro 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
dir = 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
*dir++ = dirf;                  // dirección del buffer a escribir por DMA
*dir++ = sizeof(v);             // tamaño de los datos a escribir por DMA
*dir=1;                         // inicia escritura por DMA
```

Ejemplo en sistema con MMU e IO-MMU

```
uint32_t *dir;                    // para referenciar el registro 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
dir = mmu_map(X,12);              // crea en MMU dir. lógicas asociadas X,X+1,...
// crea en IOMMU direccjones de bus asociadas a buffer a escribir
dirb = iommu_map(virt_to_phys(&v),sizeof(v));
*dir++ = dirb;                    // dirección del buffer a escribir por DMA
*dir++ = sizeof(v);               // tamaño de los datos a escribir por DMA
*dir=1;                           // inicia escritura por DMA
```

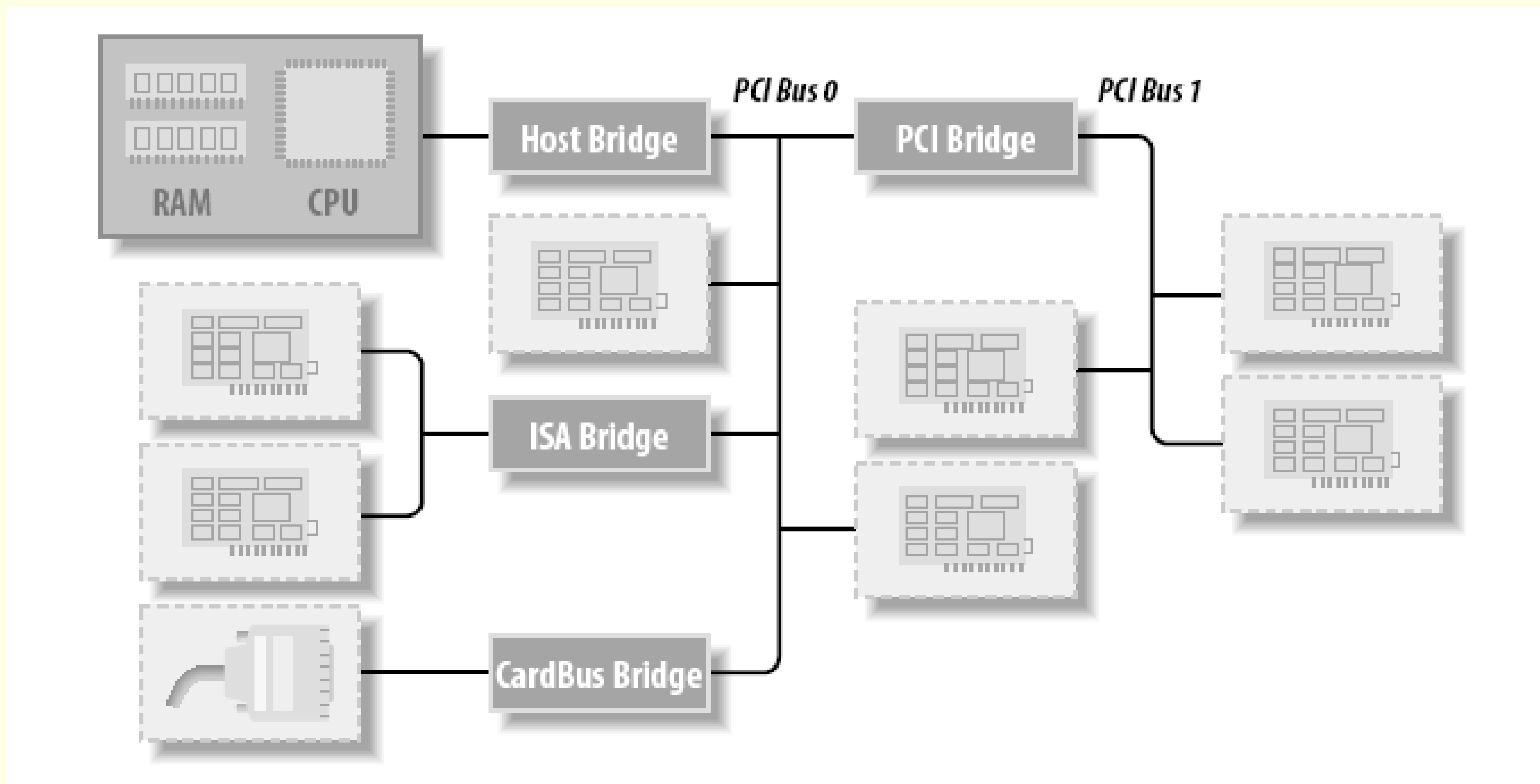
Configuración de dispositivos

- Cada dispositivo usa diversos recursos:
  - Rango dir E/S (puertos y/o memoria) + línea(s) de interrup.
- ¿Cómo asignarlos?
  - Estáticamente
  - Mediante *jumpers*
  - Configurables por software
    - Uso inicial de direccionamiento geográfico (por posición en bus)
    - Posibilita técnicas de *plug & play*
      - Configuración automática de dispositivos en arranque
    - Puede posibilitar técnicas de *hot-plugging*
      - Conexión de dispositivo con el sistema arrancado
    - Inclusión de info. adicional del dispositivo: vendedor, ID, clase, ...
      - Permite averiguar, y cargar, manejador en tiempo de ejecución

Jerarquía de buses de E/S: buses internos

- Variedad de dispositivos de muy diversas características
  - Conveniencia de jerarquía de buses
- Buses de E/S internos (PCI, ISA, ...)
  - Dispositivos directamente accesibles mediante PIO/MMIO
  - Jerarquía por limitaciones, rendimiento, compatibilidad, ...
    - Puentes (*Bridges*) permiten su interconexión
  - Proceso de enumeración de dispositivos (si el bus lo permite)
    - "Descubrimiento" mediante direccionamiento geográfico
    - Configuración de dirs. E/S e IRQs
    - Puede obtenerse info. de dispositivo (vendedor, ID, clase, ...)
      - Incluso en algunos niveles de consumo de energía disponibles

Jerarquía de buses internos



Linux Device Drivers, 3ª Edición

Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

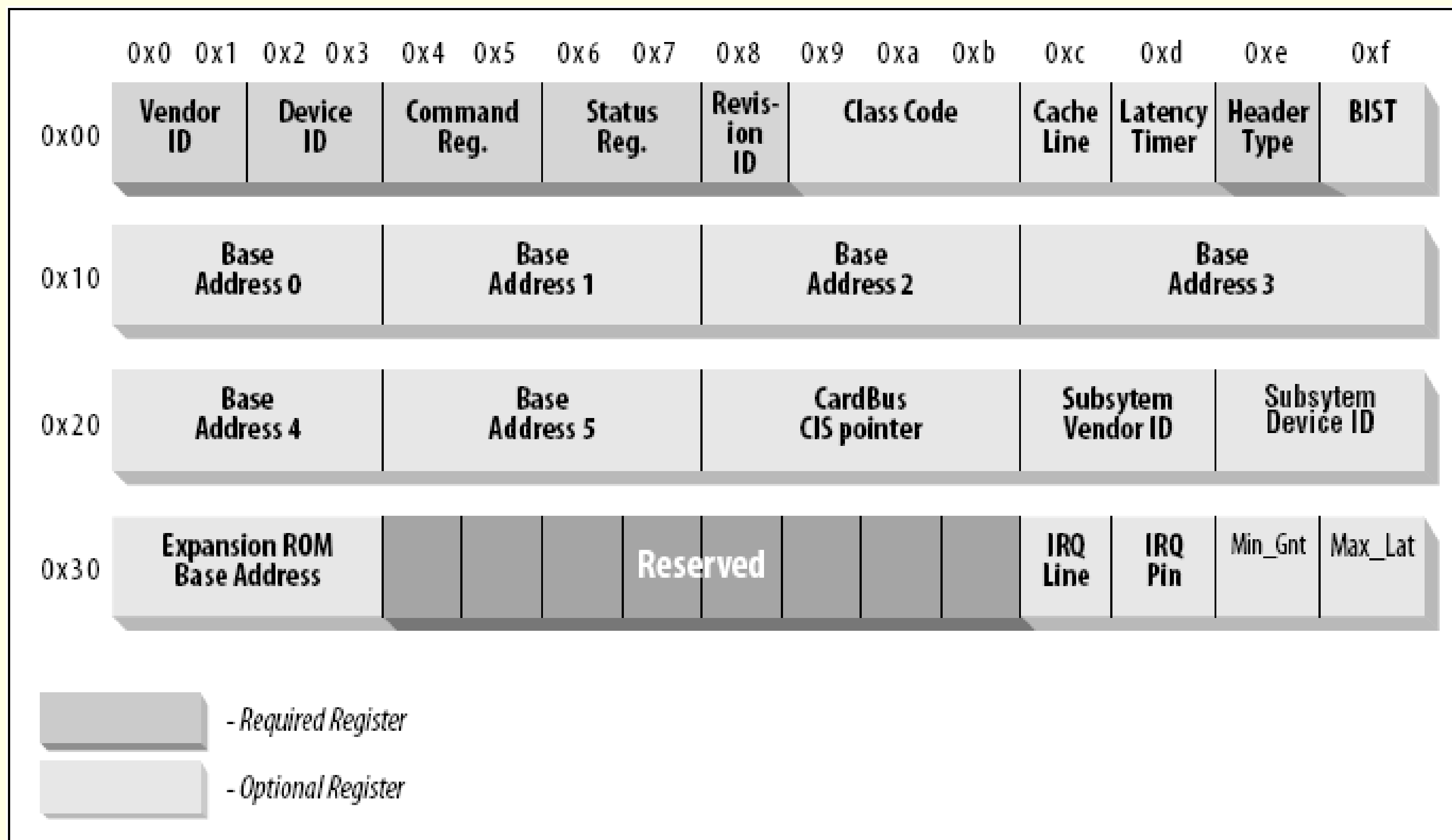
Introducción a la programación del bus PCI

- 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: "dispositivo" con múltiples "funciones"
  - Realmente, cada función es un dispositivo
- Bus jerárquico 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*)
    - Puede haber varios *HBs*: terminología Linux, múltiples dominios
- Cada *slot* tiene 4 *pines* de interrupción (A-D)
  - Cada función puede usar uno (PCI usa interrup. compartidas)
  - Conectados de forma entrelazada (*pin* A *slot* 0→*pin* B *slot* 1,..)
  - Ajeno a PCI:conexión *pines* a líneas controlador interrupciones
    - Puede ser fija o programable

Introducción a la programación del 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)
  - ¿Cómo HW realiza acceso geográfico? Solución habitual:
    - Si dispo. accedido en bus 0, *HB* genera acceso directo al mismo
      - Cada bit del bus direcciones a IDSEL de un *slot* (acceso tipo 0)
    - Si en otro, *HB* propaga dirección a *PPBs* y afectado repite proceso
      - Acceso de tipo 1

Info. de configuración de dispositivo en PCI

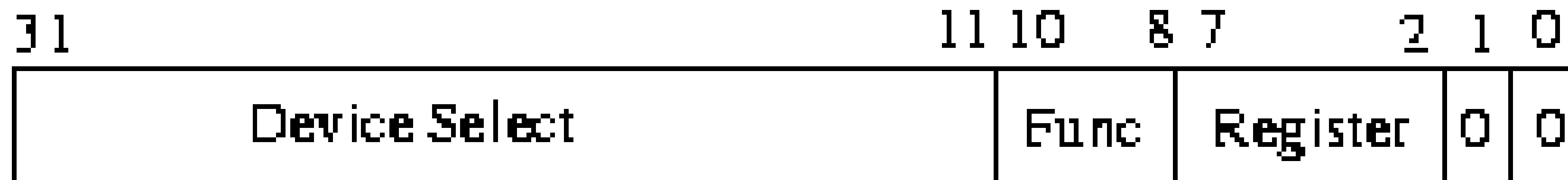


Linux Device Drivers, 3ª Edición

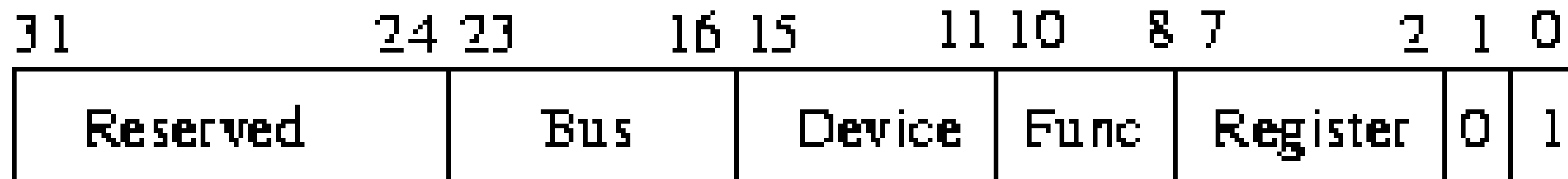
Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

Tipos de ciclos de configuración

Fuente: TLDP



Tipo 0

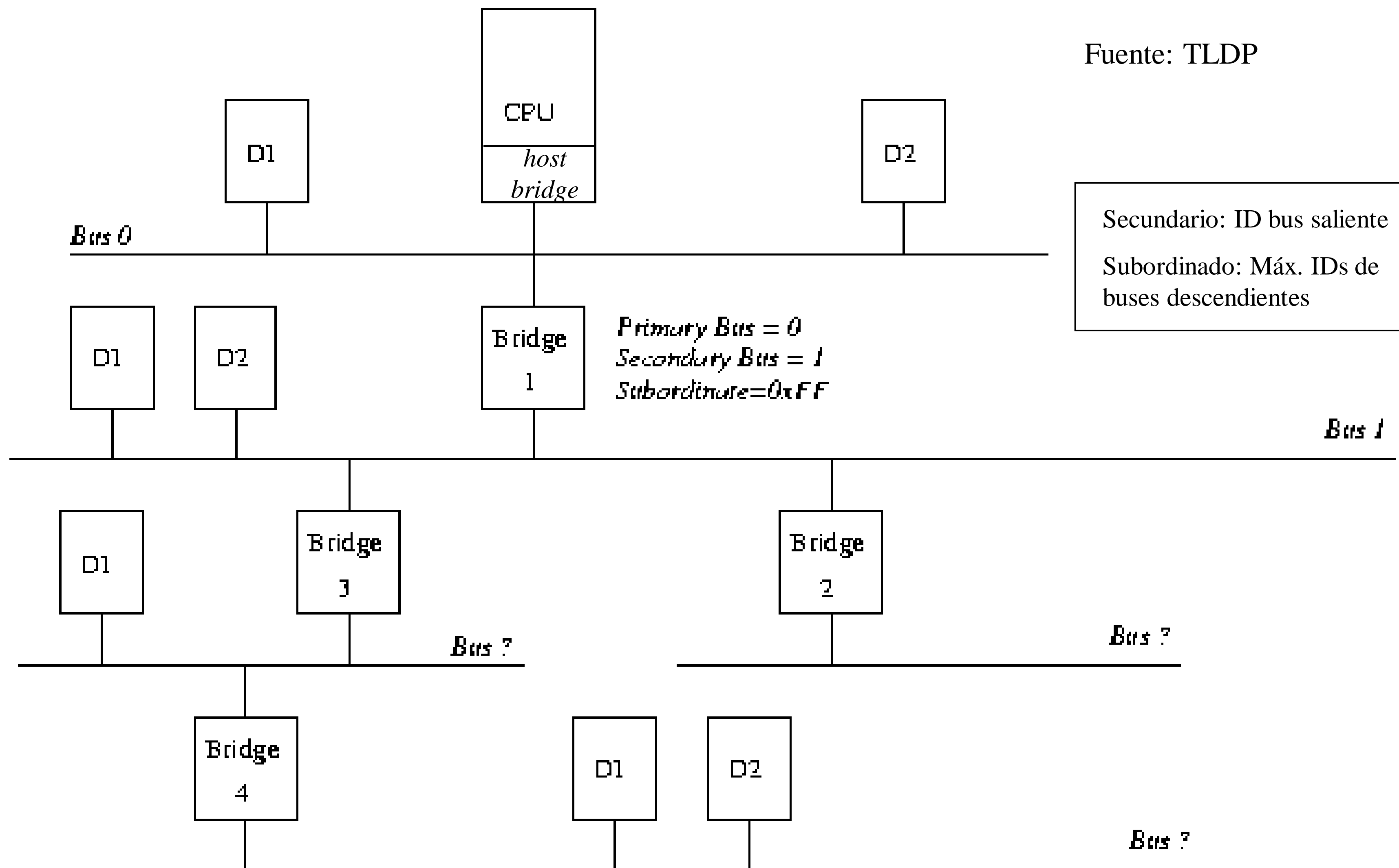


Tipo 1

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, prueba las otras 7 funciones (dispositivos)
    - Por cada función encontrada, si es 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)
    - Escribir en BAR dirección MMIO o PIO asignada
    - Escribiendo palabra con 1s en BAR y leyendo a continuación
      - Tamaño requerido por la región
- *http://wiki.osdev.org/PCI*

Enumeración de buses en PCI: Inicio

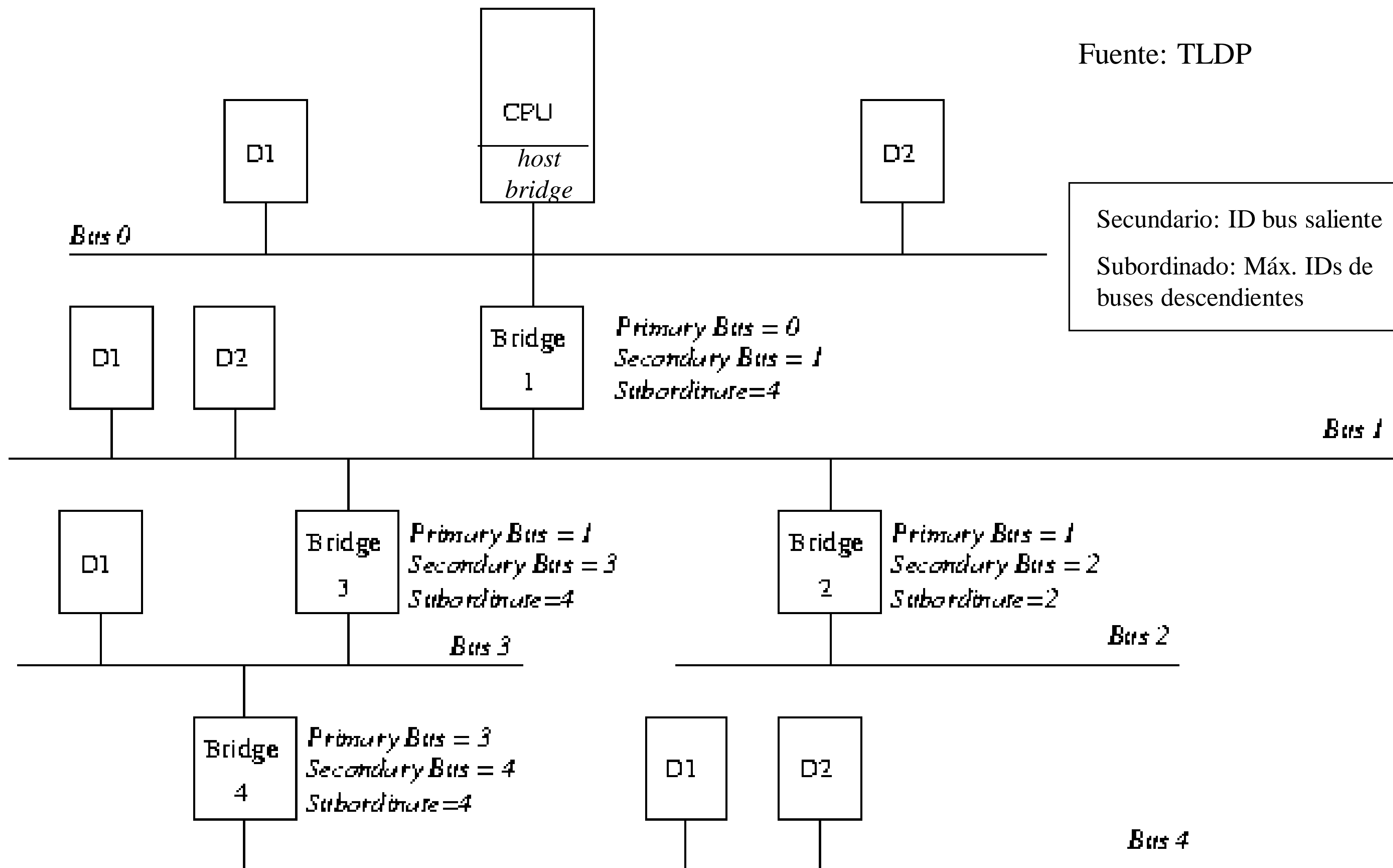


Fuente: TLDP

Secundario: ID bus saliente

Subordinado: Máx. IDs de buses descendientes

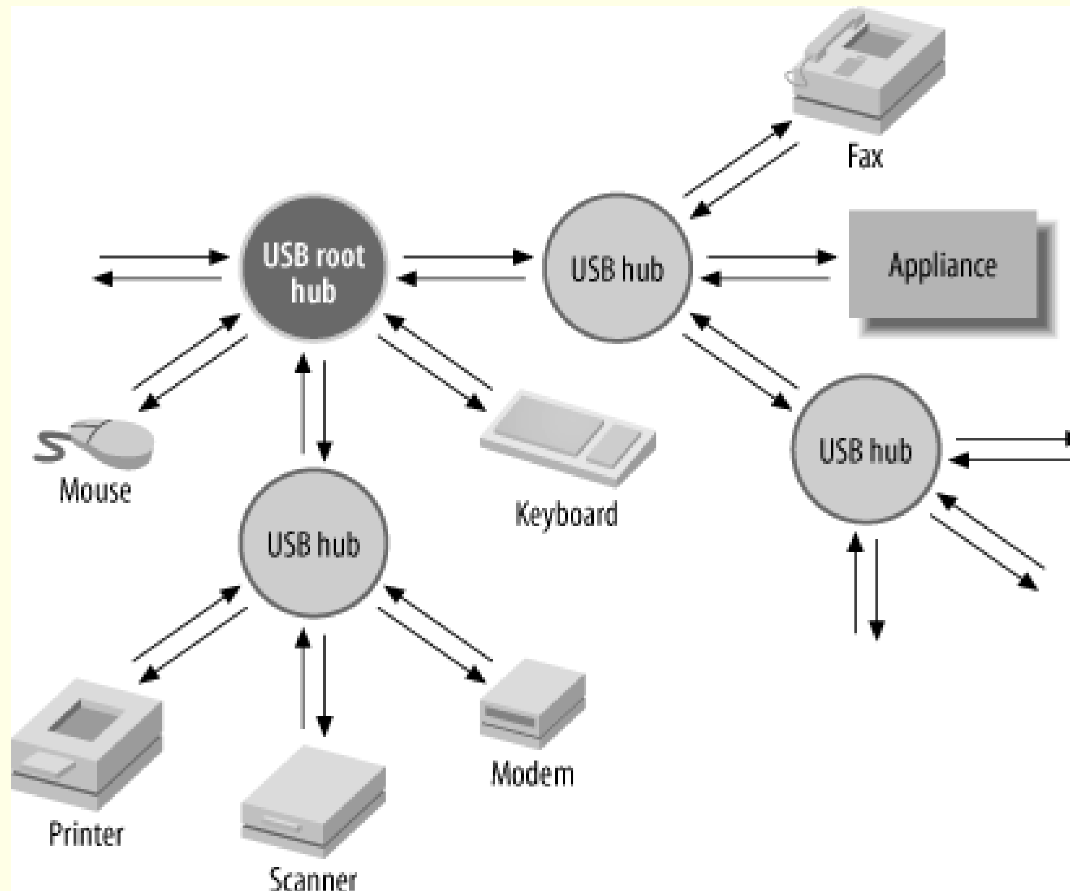
Enumeración de buses en PCI: Fin



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

- Conectados a internos mediante controlador (*host controller*)
  - Controlador descubierto/config. en enumeración de b. internos
- Dispositivos no directamente accesibles mediante PIO/MMIO
- SW interacciona (PIO/MMIO) con controlador de bus
  - Controlador interacciona con dispositivos conectados al bus
- Enumeración de bus externo (si lo permite, como USB)
  - Descubrimiento de dispositivos
  - Configuración de dispositivo (no de dirs. E/S ni de IRQs)
    - Puede asignar una dirección interna (de 7 bits en USB)
    - Puede obtenerse info. de dispositivo (vendedor, ID, clase, ...)
      - Incluso en algunos niveles de consumo de energía disponibles
  - Linux: mandato `lsusb -tv`

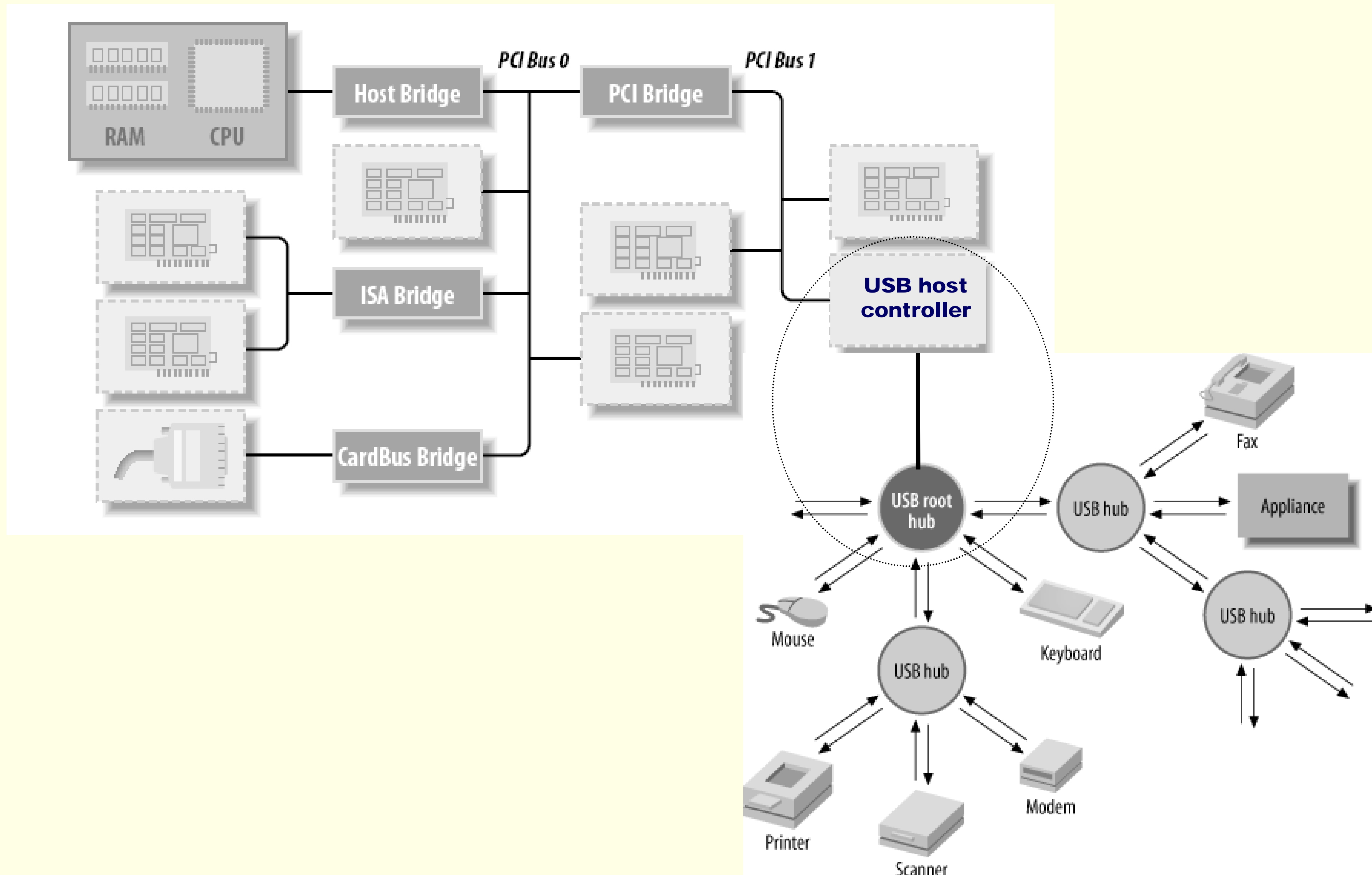
Bus externo USB



Designing Embedded Hardware

John Catsoulis

Jerarquía de buses de E/S



Control de consumo de energía de dispositivos

- Algunos dispositivos permiten distintos niveles consumo energía
- Estándar *Advanced Configuration and Power Interface* (ACPI)
  - Estados del sistema: S0 (*fully on*) a S5 (*fully off*)
  - Estados del dispositivo
    - D0: dispositivo completamente operativo (consumo máximo)
    - D3: dispositivo apagado
    - D1, D2: estados intermedios dependientes del dispositivo
- En "descubrimiento" de dispo. se averiguan niveles disponibles
- Cuando condiciones del sistema lo requieran →
  - Disminución alimentación de energía en el sistema
  - Aspectos operacionales (p.e. cambio de "misión")
  - Solicitud explícita de usuario
- → Se baja nivel de energía de dispositivos que lo permitan
  - De forma ordenada: 1º apagar dispos., después controlador bus

Contenido

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

MMIO vs. PIO

- PIO requiere instrucciones en ensamblador
  - x86: IN|INS|INSB|INSW|INSD; OUT|OUTS|OUTSB|OUTSW|OUTSD
  - Si UCP distingue modos de ejecución→ops. PIO privilegiadas
    - Aunque Pentium permite habilitarlas a modo no privilegiado
      - IOPL (todos los puertos); IOPERM (los puertos seleccionados)
- MMIO permite acceso convencional con puntero a dir. registro
  - *Programming Embedded Systems*. M. Barr, A. Massa

```
uint32_t volatile *pGpio0Set = (uint32_t volatile *)(0x40E00018);
*pGpio0Set = 1; /* Set GPIO pin 0 high. */
```

  - Cuidado con restricciones de alineamiento en acceso a memoria
    - Muchas UCPs sólo permiten accesos de Xbytes si dir. múltiplo X
  - Si procesador usa MMU, necesidad de crear *mapping*
    - Hacer que una DL corresponda a DF deseada (Linux ioremap)

Estructura del registro de controlador de E/S

- Información en registro 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 *bit fields*: ↓ problemas de portabilidad
    - Aspectos dependen del compilador (p.e. orden bits en memoria)
  - Uso de máscaras de bits: ↓ sin soporte de tipos
- Máscaras de bits (*Programming Embedded Systems*. M. Barr)
  - Comprobando valor :

```
#define TIMER_COMPLETE 0x08
if (*pTimerStatus & TIMER_COMPLETE) ...
```

  - Poniendo a 1 el bit4, a 0 el bit 2 y cambiando el bit 7:

```
*pTimerStatus |= 0x10; *pTimerStatus &= ~(0x04); *pTimerStatus ^= 0x80;
```

Ejemplo de *bit fields* del *kernel* de Linux

http://lxr.free-electrons.com/source/include/linux/tcp.h#L85

```
struct tcp_options_received {
        long    ts_recent_stamp;/* Time we stored ts_recent (for aging) */
        u32     ts_recent;      /* Time stamp to echo next              */
        u32     rcv_tsval;      /* Time stamp value                     */
        u32     rcv_tsecr;      /* Time stamp echo reply                */
        u16     saw_tstamp : 1, /* Saw TIMESTAMP on last packet         */
                tstamp_ok : 1,  /* TIMESTAMP seen on SYN packet         */
                dsack : 1,      /* D-SACK is scheduled                  */
                wscale_ok : 1,  /* Wscale seen on SYN packet            */
                sack_ok : 4,    /* SACK seen on SYN packet              */
                snd_wscale : 4, /* Window scaling received from sender  */
                rcv_wscale : 4; /* Window scaling to send to receiver   */
        u8      num_sacks;      /* Number of SACK blocks                */
        u16     user_mss;       /* mss requested by user in ioctl       */
        u16     mss_clamp;      /* Maximal mss, negotiated at connection setup */
};
static inline void tcp_clear_options(struct tcp_options_received *rx_opt)
{
        rx_opt->tstamp_ok = rx_opt->sack_ok = 0;
        rx_opt->wscale_ok = rx_opt->snd_wscale = 0;
}
```

Empaquetamiento de estructuras

- Ejemplo de *Linux Device Drivers*:

```
struct {
    u16 id;
    u64 lun;
    u16 reserved1;
    u32 reserved2;
} __attribute__ ((packed)) scsi;
```

- Compilador puede incluir relleno por restricciones de alineamiento

Endianness

- Problema habitual en comunicación sistemas heterogéneos
  - Pero también en programación de dispositivos
- Dispositivo utiliza un determinado *endian*
  - P.ej. información de dispositivo de PCI es *little-endian*
  - Si código para distintos procesadores debe adaptarse
- *http://lxr.free-electrons.com/source/sound/hda/hdac_controller.c*

```
bus->corb.buf[wp] = cpu_to_le32(val);

res_ex = le32_to_cpu(bus->rirb.buf[rp + 1]);
```

Operaciones con *efectos secundarios*

- Un registro de E/S no es una celda de memoria pasiva
  - Lectura de puerto puede no estar relacionada con escritura
- Operaciones sobre registro E/S puede tener efectos secundarios
  - Incluso op. lectura puede desencadenar una acción en dispo.
  - A veces: evitar lectura en actualización de parte del registro
- Solución: Uso de variable espejo del registro
  - *Programming Embedded Systems*. M. Barr, A. Massa
  - Asignación inicial a la variable y al registro

    ```
    timerRegValue = TIMER_INTERRUPT;    *pTimerReg = timerRegValue;
    ```
  - Cambio un bit sin leer el registro

    ```
    timerRegValue |= TIMER_ENABLE;      *pTimerReg = timerRegValue;
    ```

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 sólo 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

Volatile en C

```
char volatile *p = (char  volatile *)(0x40000000);
uint32_t volatile *timer = (uint32_t 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
  - No reordenar accesos a variables de este tipo
    - Pero no especifica orden accesos volátiles y no volátiles
- Definición en el estándar obliga a compilador
  - Pero no a procesador: éste 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 procesador):
  - 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, compilador traduce accesos a variables → a memoria
  - No movimiento de instrucciones entre lados de la barrera
  - GNU cc: `asm volatile("" ::: "memory");`
- Como volatile, barrera de compilador no es suficiente en el ejemplo
  - Procesador puede optimizar reordenando instrucciones
    - Solución: barrera de memoria
  - Aunque en ese ejemplo puede ser más eficiente que volatile
    - Basta con asegurar que START se ejecuta al final
    - Compilador podría optimizar sentencias previas

Barrera de memoria

- Barrera de memoria: orden parcial en accesos antes y después
  - Completa (Pentium MFENCE):
    - LDs|STs antes BM globalmente visibles antes que LDs|STs después
  - De lectura (Pentium LFENCE):
    - LDs antes barrera globalmente visibles antes que LDs después
  - De escritura (Pentium SFENCE):
    - STs antes barrera globalmente visibles antes que STs después
- Normalmente, si se necesita BM, también barrera de compilador
  - Linux rmb, wmb, mb incluyen ambos tipo de barreras
- NOTA: PIO sólo requiere que UCP no adelante instrucciones
  - *Flush* del *pipeline* (Motorola NOP; Pentium 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
```

Programación de interrupciones

- Instalación de manejador de interrupción en vector
- Operaciones realizadas por el manejador:
  - Salvar/restaurar contexto requerido del procesador
  - Bucle por cada dispositivo de la línea; Si dispo. ha interrumpido
    - Reconocer la interrupción, si requerido
    - Realizar labor específica
- Si rutina int. y programa comparten variable: debe ser volatile
  - Pero requiere además ser de actualización atómica (sig_atomic_t)
    - Escritura en variable sólo requiera una instrucción
- Puede ser necesario crear sección crítica (SC) con respecto a int.
  - En UP prohibir interrupción conflictiva
  - En MP además *spinlocks*
  - Minimizar tiempo SC → minimiza latencia de activación int.

Programación de DMA

- Supongamos operación de lec. o esc. con múltiples *buffers*:
  - Esc: ***N*** *buffers* → Dispo. | Lect: ***N*** *buffers* ← Dispo.
  - 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 cache de datos involucrados
  - Después de lectura: invalidación cache de datos involucrados
- Cont. DMA dispo. con limitaciones en rango acceso a memoria
  - Uso de doble (*bounce*) *buffer* (ineficiente)

Aspectos de configuración

- Dispositivos no configurable por software
  - SW no debe usar dir. E/S ni IRQ fijas sino parametrizables
  - A veces se conoce dir. E/S pero no IRQ → int. *probing*
    - Se programa dispositivo y a ver qué interrupción se genera
- Dispositivos configurable 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*, SO o la propia aplicación
    - Si *firmware*, después app.o SO averigua dir. E/S e IRQs dispo.
      - Leyendo regs. config. correspondientes mediante dir. geográfico
  - En arranque (o en *plug*) "enumera" buses externos
    - Realizado por controlador + SW (*firmware*, SO o la aplicación)

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

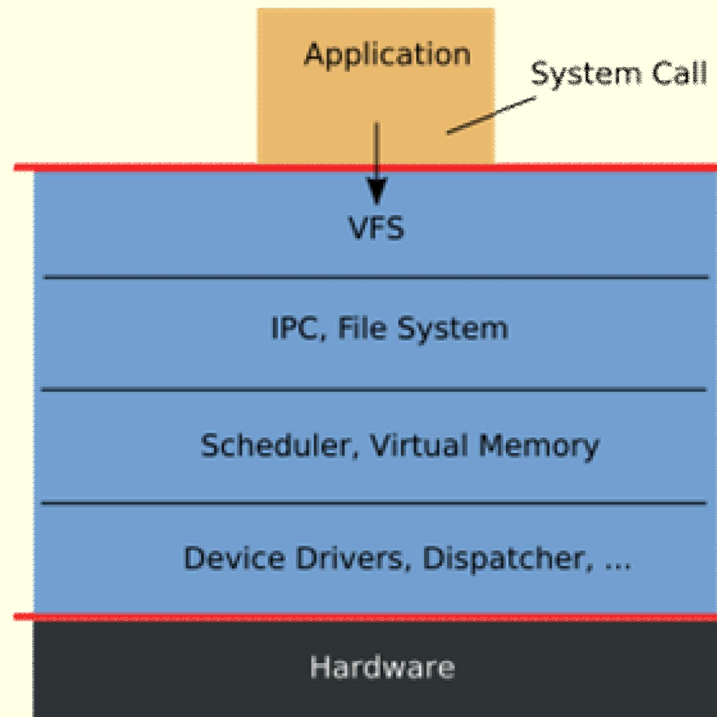
Programación de dispositivos en máquinas sin SO

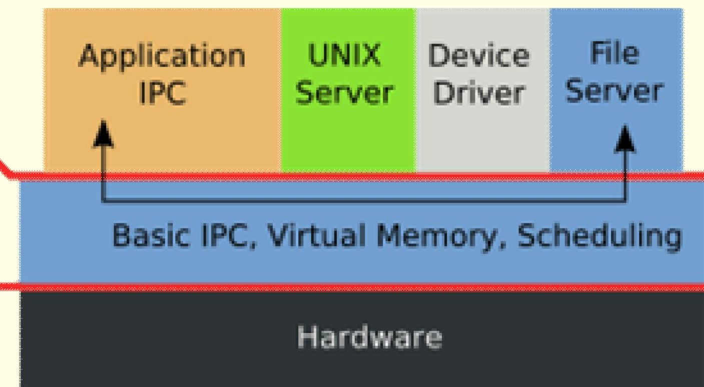
- Programador debe enfrentarse con toda la problemática identificada
  - Sólo con la ayuda del *runtime* del lenguaje
- Alternativas en el diseño de app que gestiona múltiples dispositivos
  - Ejecutivo cíclico con espera activa y *polling* de dispositivos
  - Ejecutivo cíclico con interrupciones
    - 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):
    - 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: es memoria no paginable
- 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 de usuario en monolíticos?

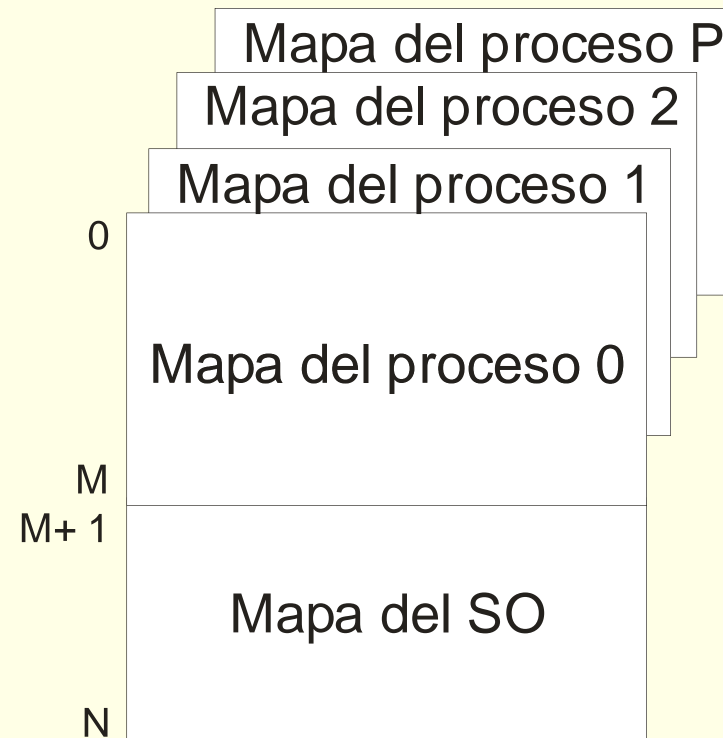
- Mejor si se puede manejar dispo. desde software en modo usuario
  - Todas las ventajas del micronúcleo
- ¿Cómo acceder a recursos privilegiados?
  - PIO: en x86 iopl, ioperm
  - MMIO: mmap de /dev/mem
- Pero...
  - No es posible manejo de interrupciones
  - Y no suficiente eficiente para dispositivos de altas prestaciones
    - Sobrecarga por cambios de modo y de contexto, código y datos del manejador expulsables de memoria (posible uso de mlock), …

Contexto de ejecución de funciones del manejador

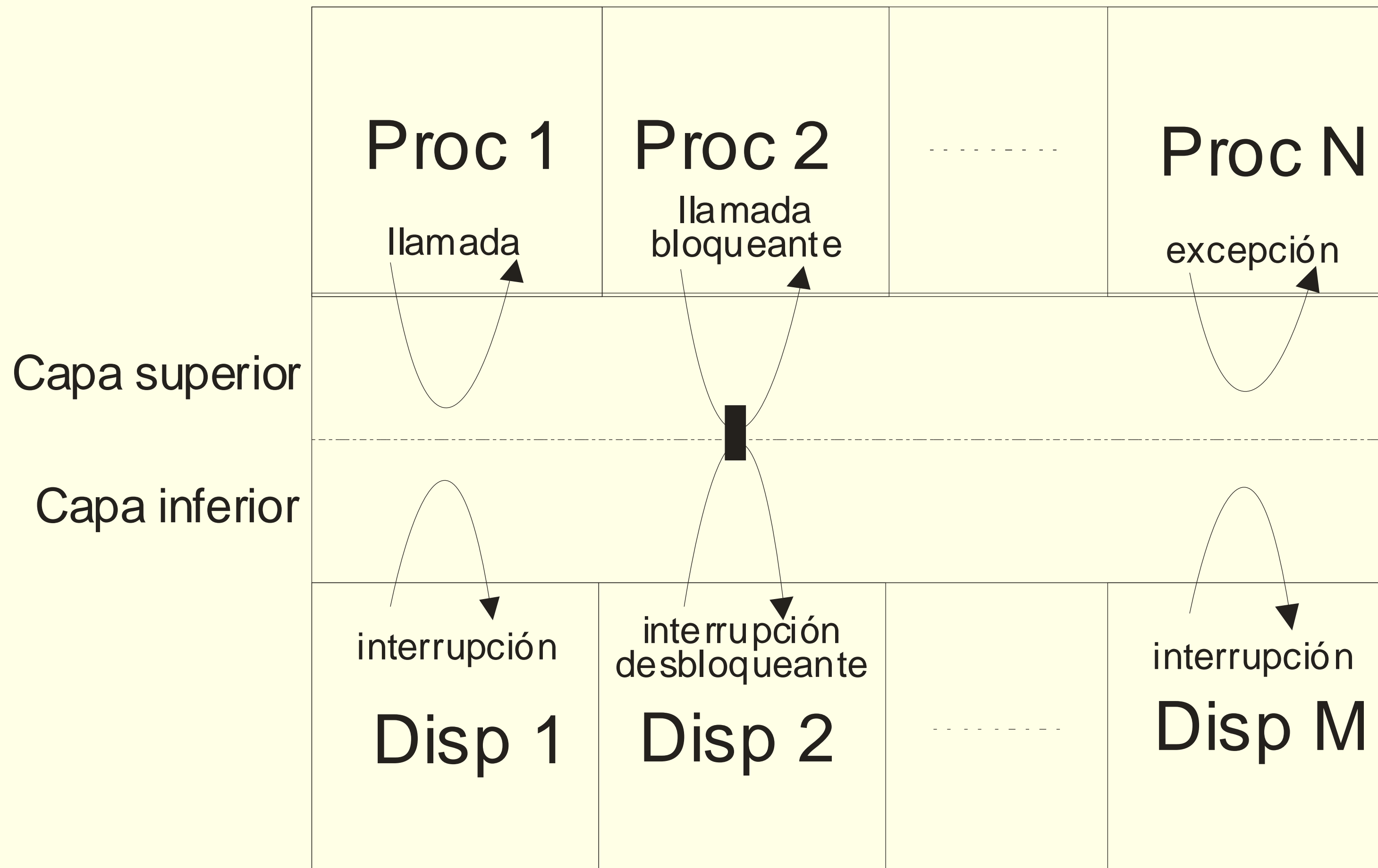
- Importante distinción entre funciones del manejador:
  - Su contexto de ejecución
- Funciones de acceso al dispo. (abrir, leer, escribir, cerrar, ...)
  - Ejecución en el contexto del proceso “solicitante”
  - Se puede acceder a mapa de memoria de proceso actual
  - Se puede realizar una operación 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

Contextos de ejecución

- SO programa dirigido por eventos:
  - Una vez iniciado el sistema, sólo 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



Organización del SO



Diseño de manejador de dispositivo de caracteres

- Pautas de diseño usando un dispositivo de E/S hipotético X:
  - Opera en modo carácter, dirigido por interrupciones y sin DMA
  - Operación de escritura:
    - dato → r. datos;
    - orden → r.control;
    - Interrupción → fin operación
  - Operación de lectura:
    - orden → r.control;
    - Interrupción → fin operación: dato en r. datos
- **No se tienen en cuenta aspectos de sincronización**
  - **ni entre llamadas concurrentes ni entre llamada e interrup.**

Objetivos del manejador hipotético

- Minimizar cambios de 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
    - App. lectora consume datos; dispositivo los produce
    - App. puede acceder a ficheros para generar/procesar datos y
      - puede haber fallos de página al acceder a *buffer* de usuario
      - Accesos a disco también concurrentes con operación del dispositivo
- Priorizar uso de dispositivo sobre ejecución de aplicaciones:
  - Dispo. sirve peticiones aunque en ejecución proc. alta prioridad

Versión lectura sin *buffering*: 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);
}
```

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

Versión lectura con *buffer* de 1 byte: ineficiente

```
char buf_ent;
tipo_cola_procesos cola_espera_entrada;
int lectura_X(char *dir, int tam) {
    while (tam--) {
            out(R_CONTROL_X, LECTURA);
            Bloquear(cola_espera_entrada); // demasiados cambios de contexto: 1/byte
            *(dir++) = buf_ent;   // puede causar un fallo de página
    }
}
void interrupcion_entrada_X() {
    char caracter;
    caracter = in(R_DATOS_LEC_X);
    buf_ent = caracter;
    Desbloquear(cola_espera_entrada);}
// Déficit: entre bytes, dispo. “parado” hasta que vuelva a ejecutar el proceso
```

Versión lectura con *buffer* de N bytes

```
tipo_buffer buf;
tipo_cola_procesos cola_espera_entrada;
int tam_datos
int a_leer;
int lectura(char *dir, int tam) {
    tam_datos = tam;
    while (tam_datos>0) {
        a_leer = (min(tam_datos, tam(&buf))); // tam(&buf): tamaño del buffer
        programar();
        Bloquear(&cola_espera_entrada); // nº cambios de contexto: tam/buf.tam
        copiar_de_buf_a_usuario(&buf, dir, a_leer); // puede causar un fallo de página
        tam_datos -= a_leer;
        dir+= a_leer;
    }
}
```

Versión lectura con *buffer* de N bytes

```
void interrupcion_entrada_X() {
    char c = in(R_DATOS);
    insertar(&buf,c); // buf.nelem++
    if (nelem(&buf)==a_leer) // nelem(&buf): nº bytes almacenados en buffer
        Desbloquear(&cola_espera_entrada);
    else
        programar();
}
void programar(){
    out(R_CONTROL, LECTURA);
}
```

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

// Déficit: entre llamadas, dispositivo "parado"

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: entre llamadas, dispositivo “parado”

Versión lectura anticipada

```
tipo_buffer buf;
tipo_cola_procesos cola_espera_entrada; int tam_datos, a_leer, dispo_activo=0;
int lectura(char *dir, int tam) {
    tam_datos = tam;
    while (tam_datos>0) {
        if (!vacio(&buf)) {
            a_leer = (min(tam_datos, nelem(&buf)));
            copiar_de_buf_a_usuario(&buf, dir, a_leer); // puede causar fallo de página
            tam_datos -= a_leer; dir+= a_leer;
        }
        if (!dispo_activo) {
            dispo_activo = 1; programar();
        }
        if (tam_datos>0)
            Bloquear(&cola_espera_entrada);
    }
}
```

Versión lectura anticipada

```
void interrupcion_entrada_X() {
    char c = in(R_DATOS);
    insertar(&buf,c); // buf.nelem++
    if (lleno(&buf))
        dispo_activo = 0;
    else
        programar();
    if (!vacia(cola_espera_entrada) && (nelem(&buf)>= min(tam_datos, tam(&buf))))
        Desbloquear(&cola_espera_entrada);
}
void programar(){
    out(R_CONTROL, LECTURA);
}
```

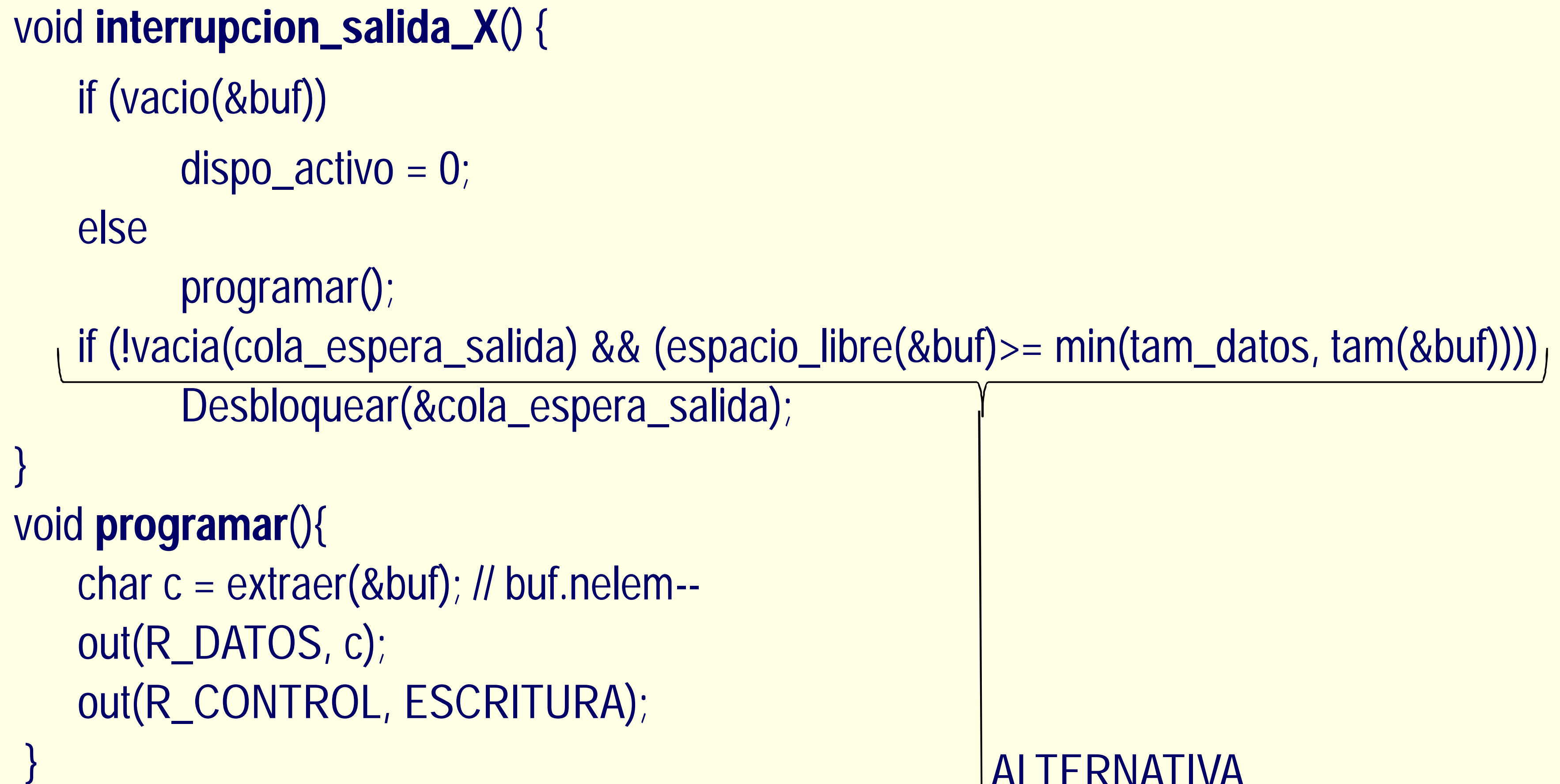
ALTERNATIVA

```
// despertar antes al proceso: en cuanto haya nº elementos superior a un cierto umbral
    if (!vacia(cola_espera_entrada) && (nelem(&buf)>= min(tam_datos, UMBRAL)))
// intenta evitar buffer lleno si hay lector y permite paralelismo entre copia y dispositivo
```

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);
}
```



```
// 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
  - Ofrecer a las aplicaciones acceso a los dispositivos
  - Tratar interrupciones de dispos. e interrupciones software
  - Cambiar nivel consumo de energía de dispositivo (si procede)
- 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
  - 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

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?
    - Función única “cajón de sastre” (en UNIX ioctl)
- struct file_operations
  - *http://lxr.free-electrons.com/source/include/linux/fs.h#L1679*
- Funcionalidad principal en las operaciones de lectura y escritura
  - Variedad de tipos de operaciones de lectura/escritura

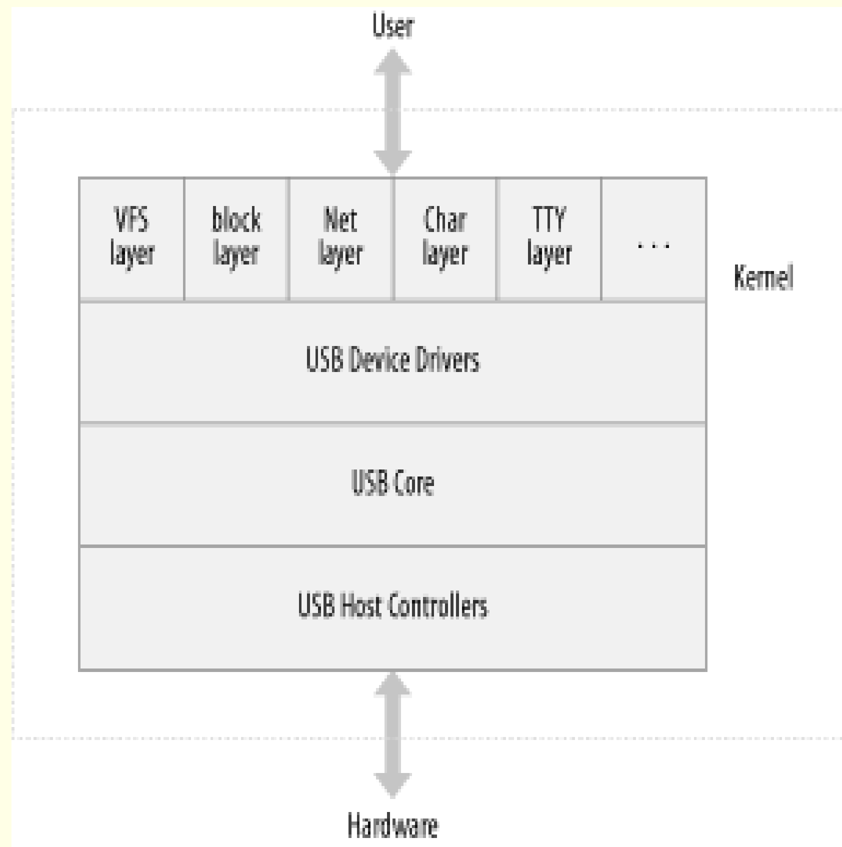
Operaciones de lectura y escritura

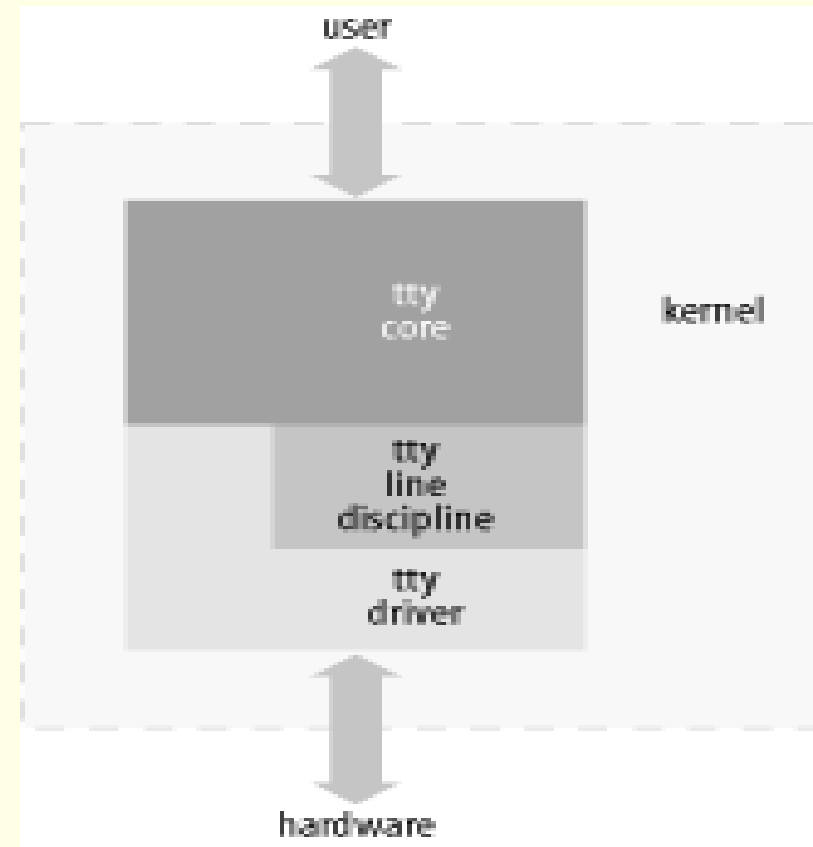
- Convencionales:
  - Programa especifica *buffer* para leer/escribir
  - SO devuelve control cuando datos leídos o escritos
  - Manejador puede usar *buffer* interno como almacén temporal
- No bloqueantes
  - Si operación no puede completarse inmediatamente → error
- Sin *buffers* intermedios
  - disp ↔ 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 controlador PCI
    - Manejador del controlador 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
  - **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
- Descarga del módulo (además de al parar el SO)
  - **Manejador de bus** descubre desconexión de dispositivo
    - Genera evento de desconexión hacia modo usuario
  - Proceso de usuario llama a la función para eliminarlo
    - Descarga módulo si último dispositivo

Aspectos específicos manejadores dispo PnP

- Función inicial: no añade dispositivo
  - Registra funciones para añadir/borrar dispositivo
    - pci_register_driver, usb_register, …
  - Serán invocadas cuando se descubran los dispositivos
  - Ejemplo info. de dispositivos PnP en PCI y USB
  - *http://lxr.free-electrons.com/source/drivers/usb/host/ehci-pci.c*
  - *http://lxr.free-electrons.com/source/drivers/media/usb/uvc/uvc_driver.c*
  - *cat /lib/modules/`uname –r`/modules.pcimap*
  - *cat /lib/modules/`uname –r`/modules.usbmap*
- Función que añade dispositivo:
  - Determina su configuración leyendo mediante dir. geográfico
    - Direcciones MMIO, puertos PIO, líneas de interrupción,…

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
- Soporte de DMA
- Control de tiempo
- 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 sólo 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 de DMA

- Mantenimiento de la coherencia (dma_alloc_coherent)
- Manejo de direcciones de bus requeridas por IOMMU
  - virt_to_bus, bus_to_virt
- Gestión de *scatter-gather* (struct scatterlist)
- Uso transparente de *bounce buffers*

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:
      - Sólo por tiempo (schedule_timeout)
      - Por un evento y por tiempo (wait_event_timeout)
    - Espera activa:
      - Sólo para esperas brevísimas (nanosegundos) (ndelay)
      - 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 sólo 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 sólo 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*
- Sólo se puede hacer un bloqueo
  - Si en contexto **no atómico**
- Sólo 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 sólo afecta al acceso a ese dispositivo
    - Puede activarse nueva versión del manejador sobre la marcha
  - Uso de memoria convencional
    - Memoria paginable 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