S. empotrados y ubicuos

Programación de dispositivos
(2ª sesión)

Fernando Pérez Costoya
fperez@fi.upm.es

Contenido

- Introducción
- Repaso de aspectos básicos del sistema de E/S
- 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 sólo 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
    - Sólo para proceso *root* o con la *capability* CAP_SYS_RAWIO

MMIO

- 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. */
    ```
  - Normalmente, acceso sólo en modo privilegiado
    - Las direcciones de E/S sólo aparecen en el mapa del SO
  - Para acceso modo usuario: dir. E/S dentro del mapa del proceso
    - En Linux mediante *mmap* de /dev/mem
    - Sólo para proceso *root*
  - Cuidado con restricciones de alineamiento en acceso a memoria
    - Muchas UCPs sólo permiten accesos de X bytes 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#L91

```
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

- Compilador puede incluir relleno entre campos
  - por restricciones de alineamiento
  - Puede no encajar con estructura del registro del dispositivo
  - Algunos compiladores permiten eliminarlos
- Ejemplo con compilador de GNU (*Linux Device Drivers*):

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

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.
  - En ese caso, 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
    - 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 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 a esa variable
  - 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 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
  - 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 sólo requiere que UCP no adelante 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
```

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 dispositivo ha interrumpido, realizar la labor específica
- ¿Anidamiento en el tratamiento de instrucciones?
  - No permitir tratamiento de una int. mientras se trata otra
    - Rutina de int. no rehabilita recepción de int. durante su ejecución
    - SW + sencillo pero con peor t. respuesta y sin soporte de prioridad
  - No permitir tratamiento int mientras se trata otra del mismo tipo
    - Rutina int. rehabilita interrup. pero enmascara las del mismo tipo
    - Solución más habitual: no requiere que rutina int. sea reentrante
  - Permitir tratamiento de una int. mientras se trata cualquier otra
    - Rutina interrupción rehabilita interrupciones

Sincronización en tratamiento 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 sólo requiera una instrucción
      volatile sig_atomic_t v;
  - No sólo 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 sólo operaciones críticas
  - Mejor tiempo respuesta y menos problemas de sincronización

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)

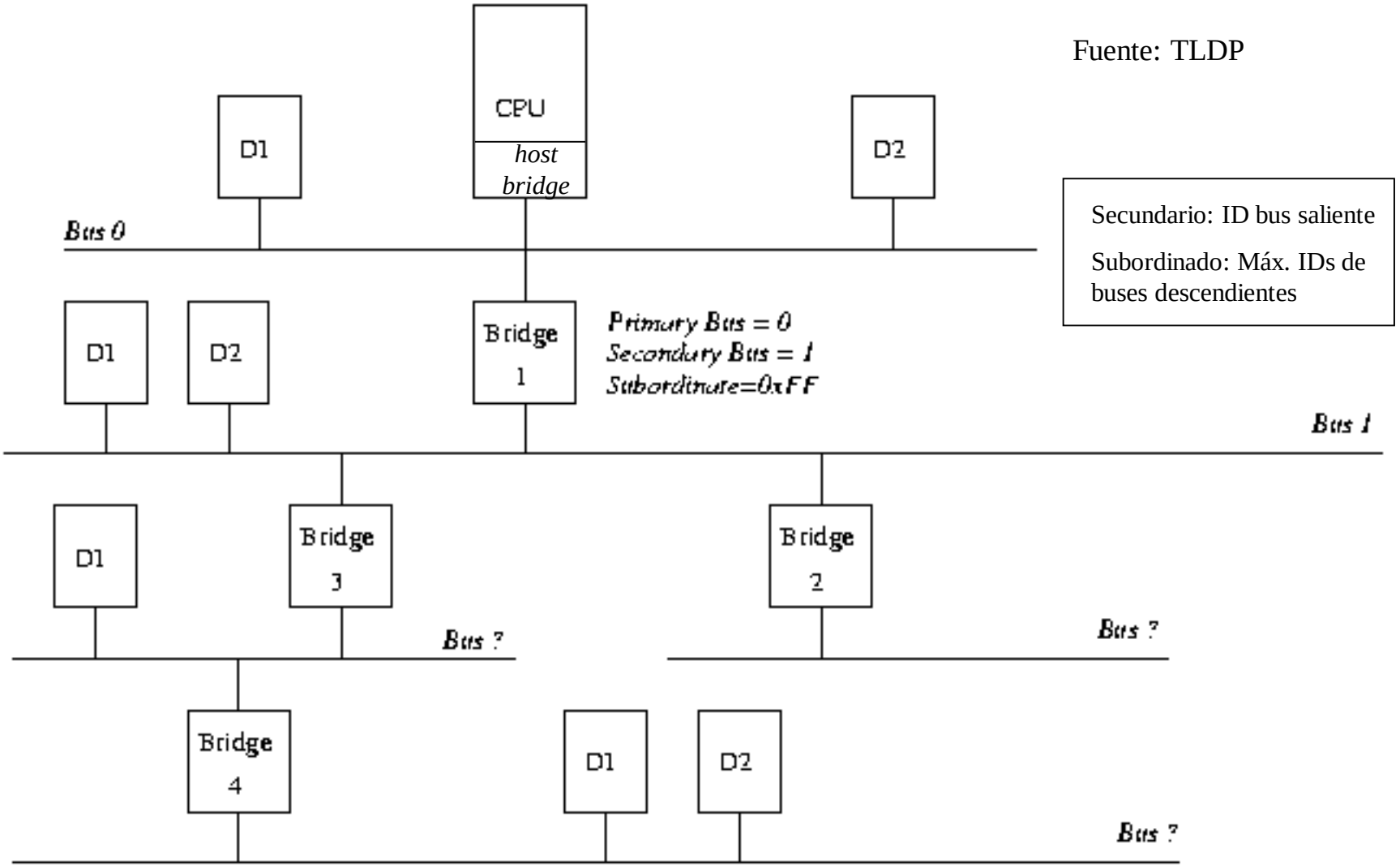
Aspectos de configuración

- 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)

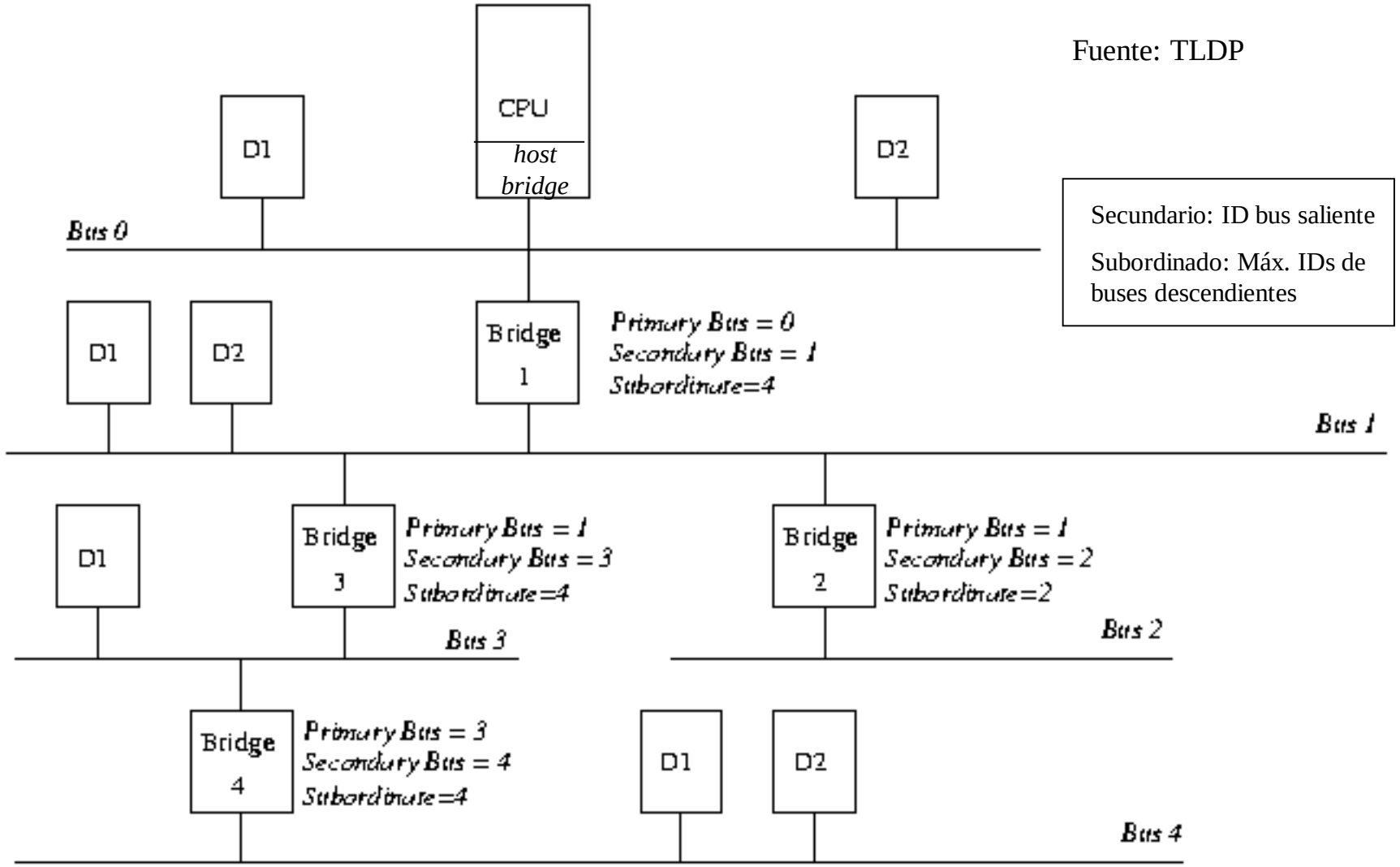
Introducción a la programación del bus PCI

- 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



Enumeración de buses en PCI: Fin



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 volatile *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
wmb();                                  // barrera compilador + b. memoria de escritura
*dir=1;                                 // inicia escritura por DMA
```

Ejemplo en sistema con MMU pero sin IO-MMU

```
uint32_t volatile *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
wmb();                        // barrera compilador + b. memoria de escritura
*dir=1;                       // inicia escritura por DMA
```

Ejemplo en sistema con MMU e IO-MMU

```
uint32_t volatile *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 direcciones 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
wmb();                                  // barrera compilador + b. memoria de escritura
*dir=1;                                 // inicia escritura por DMA
```