Ingeniería en Informática

Sistemas Operativos

Bases para el diseño de un driver USB en un sistema operativo didáctico

Equipo	Nicanor Casas
	Graciela De Luca
	Waldo Valiente
Docente	Gerardo Puyo
	Martín Cortina
	Sergio Martín

Día Cursada: *Martes*

	Alumnos	
Apellido	**Nombre**	**DNI**
Pietropaolo	Pablo	32.010.630
Sandler	Pablo	31.655.622
Sposato	Leandro	28.864.758
Tiraferri	Sergio	24.591.098

Acreditación:

Instancia	Fecha	Calificación
ENTREGA	23/11/2010	
FINAL	30/11/2010	

****

****

Observaciones:

****

****

BASES PARA EL DISEÑO DE UN DRIVER USB EN UN SISTEMA OPERATIVO DIDÁCTICO. 4

****

Bases para el diseño de un driver USB en un sistema operativo didáctico.

Introducción y alcances

Este informe tiene por objetivo establecer las bases teóricas y los requisitos necesarios para el diseño de un driver USB para el sistema operativo SODIUM.
Para concretar nuestro trabajo el grupo dividimos la investigación en tres grandes ramas.

1) Investigar el funcionamiento del puerto USB a bajo nivel, describiendo componentes, estructuras de datos, estructuras de control y funciones asociadas.

2) Evaluar las diferentes formas en las que los sistemas operativos de código abierto hacen su implementación, especialmente Linux.

3) Hacer un análisis de la arquitectura SODIUM para poder determinar cuales son los cambios necesarios para el desarrollo del driver.

Para ayudarnos en las tareas de planificación y trabajo en grupo utilizamos diferentes herramientas de tales como diagramas de Gantt, repositorios de información (SVN), sistema de asignación de tickets, etc.

El objetivo final del presente trabajo será presentar a la fecha de finalización un informe con el desarrollo de los tres ítems establecidos al comienzo de la investigación.

¿Qué es un driver?

Un driver es un componente de software que permite que un sistema operativo se comunique con un dispositivo físico. De esta manera el sistema operativo se independiza de la implementación física, accediendo al dispositivo mediante funciones ofrecidas por el driver.
Normalmente los drivers son desarrollados por los fabricantes de dispositivos, quienes proporcionan uno para cada uno de los sistemas operativos más comunes. Sin embargo, no proveen un driver para todos los sistemas operativos existentes, por esta razón, ésta tarea deberá ser llevada a cabo por el mismo equipo que mantiene o desarrolla el sistema operativo.

Requisitos mínimos de un driver USB

Un controlador de dispositivos USB debe cumplir al menos las características mencionadas a continuación

- Reconocer y soportar múltiples dispositivos.
- Proveer una interfase (las funciones mencionadas anteriormente).
- Transferir streams de datos.
- Permitir conexión y desconexión en caliente.

El puerto USB

La primera especificación USB, la 1.0, fue introducida en 1996 por un consorcio de empresas informáticas (Compaq, DEC, IBM, Intel, Microsoft, NEC y Nortel). Entre sus principales objetivos al momento de su desarrollo se encontraba la de interconectar dispositivos externos a la PC, tales como impresoras, mouses, escáneres y teclados entre otros, eliminando la gran variedad de conectores y protocolos existentes hasta la fecha.

En septiembre de 1998 se lanza al mercado la versión 1.1 y comienza a implementarse masivamente tanto en equipamiento de hardware (motherboards y periféricos) como en software (drivers y aplicaciones).

Actualmente el estándar USB se utiliza en prácticamente toda la industria.
El análisis del uso y la posible evolución de esta tecnología, como así también sus características físicas están fuera del alcance de nuestra investigación.

Arquitectura USB

Los componentes principales de un sistema USB funcional son tres:

- Un Host central.

- Hubs o concentradores.

- Periféricos USB.

El host USB

El host USB es un dispositivo que se encuentra dentro de la computadora y tiene la responsabilidad de administrar los dispositivos conectados y asignar direcciones lógicas a cada uno de ellos a medida que se van agregando. También es el encargado de administrar las comunicaciones entre los periféricos y la CPU. Un host USB puede tener muchos Host Controllers y cada Host Controller puede proveer de uno o varios puertos.

Los Hubs USB

Los hubs USB, también llamados concentradores, son los dispositivos que permiten interconectar a varios periféricos al único controlador central, similar a lo que ocurre en redes de datos. Gracias a esto, hasta 127 dispositivos USB pueden ser conectados al host.
Siempre existe un controlador central (Central HUB) que se encuentra incluido en el host USB y es el primero en la cadena de hubs.

Los periféricos USB

Los periféricos USB son los últimos dispositivos de la cadena y pueden estar compuestos por uno o más dispositivos lógicos. Cada dispositivo lógico tiene asociada una dirección distinta (otorgada por el host) y de esta manera puede brindar una funcionalidad diferente.

La comunicación USB

Las comunicaciones en USB se llevan a cabo mediante canales lógicos denominados pipes (tuberías) que se trazan desde el controlador USB hacia los endpoints que son las interfaces que se encuentran en los dispositivos. El número máximo de endpoints por dispositivo es de 32. Existen dos tipos de canales lógicos y cuatro tipos de transferencias.

Funciones USB

Cuando se piensa en un dispositivo usb, se piensa en un periférico, pero un dispositivo usb podría ser un Transmisor/Receptor utilizado en el Host, Un Hub usb, una Controladora, o un periférico.

El standard hace referencias a las funciones, que son tomadas como dispositivos usb que proveen la capacidad o funcionalidad, como Impresoras, scanners, Modems, etc.
La mayoría de estas funciones manejan los protocolos de transmisión de bajo nivel y poseen una serie de buffers, típicamente de 8 bits.

Cada buffer pertenece a un endpoint – EP0 IN, EP0 OUT, etc.

Por ejemplo, el host envía un request de descriptor de dispositivo a un dispositivo, el hardware va a leer la configuración del paquete y va a determinar (observando el campo ADDR) si el paquete en cuestión es para él, y entonces, va a copiar los datos del paquete al buffer del endpoint indicado en el campo de endpoint del paquete de Setup. Luego, él va a enviar un paquete de handshake para confirmar la recepción del byte y generar una interrupción interna con un micro-controlador asociado al endpoint significando que el ha recibido un paquete.

El Software recién ahora recibe una interrupción, y debe leer los contenidos del buffer del endpoint y hacer el parsing del request del descriptor de dispositivo.

Endpoints

Los Endpoints pueden describirse como fuentes de datos que interactúan con el host al final del canal de comunicaciones. En la capa de software, el driver del dispositivo debería enviar paquetes a sus dispositivos.

Como los datos fluyen desde el host y van a terminar en el buffer del EP1 OUT, el firmware va a leer estos datos. Si se quiere devolver datos, la función no puede simplemente escribir en el bus, debido a que el bus está controlado por el host
Entonces escribe los datos en el EP1 IN, el cual yace en el buffer hasta que el host envía un paquete de control del tipo IN a ese endpoint solicitando los datos.

Los endpoints también pueden ser vistos como una interfase entre el hardware del dispositivo y el firmware que ejecuta en él.

Todos los dispositivos deben soportar el Endpoint Cero. Este es el que recibe todos los mensajes de control y estado mientras se enumera el dispositivo.

Pipes (Tuberias)

Un Pipe es una conexión lógica entre el host y los endpoints, tienen un set de parámetros asociados, como por ejemplo, el ancho de banda se le asignado a dicho puerto, que tipo de transferencia usa (Contról, Bulk, isócronas o de interrupción), la dirección del flujo de los datos y el tamaño máximo de los buffer.

USB define dos tipos de Pipes:

Stream (Flujo)

No tienen un formato definido, esto quiere decir que es posible enviar cualquier tipo de dato por el pipe. Los datos fluyen secuencialmente y hasta una dirección predefinida, ya sea de Entrada o salida. Estos pipes pueden ser controlados por el host o por el dispositivo.

Message Pipes

Estos tienen un formato definido. Son controlados en inicializados por el host. Los datos se transfieren en ambas direcciones y pueden enviarse solo transferencias de control.
Mientras el dispositivo envía y recibe datos a una serie de endpoints, el software cliente transfiere esos datos a través de pipes.

Tipos de Transferencias en el Protocolo USB

La norma USB define cuatro tipos distintos.

Transferencias de control

Son esenciales para configurar el dispositivo usb mediante las funciones de enumeración. Son típicamente ráfagas de paquetes iniciadas por el host.
La longitud de los paquetes de transferencias de control en dispositivos de baja velocidad debe ser de 8 bytes, en dispositivos de alta velocidad permite paquetes de tamaño de 8,16,32 y 64 bytes, y en dispositivos de alta velocidad debe ser de 64 bytes.

Una transferencia de control puede tener hasta tres etapas.

Etapa de Configuración:

Es cuando el request es enviado. Esto consiste en tres paquetes. El token de configuración es enviado primero conteniendo la dirección y el número de endpoint.
El paquete de datos es enviado a continuación y siempre es un Data0 e incluye un paquete de setup con detalles del tipo de request.
El último paquete es de handshake, utilizado para confirmar la recepción exitosa o para indicar un error.
Si la función recibe satisfactoriamente los datos de configuración responde con un ACK, de otra manera ignora los datos y no envía ninguna paquete de handshake.

Etapa de datos:

Consiste en una o más transferencias. El paquete de configuración indica la cantidad de datos a transmitirse en esta etapa. Si se excede el tamaño máximo del paquete, los datos serán enviados en varias transferencias.
Las transferencias de datos tienen dos diferentes escenarios dependiendo de la dirección de las transferencias.

Etapa de estados:

Reporta el estado general del request, esto varía dependiendo de la dirección en que haya sido enviada la transferencia.

Transferencias de Interrupción

Si un dispositivo USB requiere atención del host, este debe esperar hasta que el host lo cense y este pueda reportar que necesita urgente atención.

Características.

- Latencia fija.
- Usan pipes de tipo Stream (Unidireccionales)
- Detección de errores y reintento siguiente
- Transferencias Isócronas (isochronous Transfers)

Transferencias Isócronas

Este tipo de transferencias ocurre continua y periódicamente, típicamente contienen información sensible en el tiempo como audio o video. El tiempo de transmisión podría no estar sincronizado todo el tiempo, sin embargo, si un paquete se pierde es difícil que un usuario se de cuenta.

Características de las transferencias Isócronas.

- Garantizan el acceso a el ancho de banda USB
- Utilizan un Stream unidireccional
- Detección de errores CRC, pero no reenvía los paquetes si estos son erroneos.
- Solo se utiliza en modos de velocidad High y Full.

El máximo tamaño de datos especificado en el standard puede ser: 1023 bytes para un dispositivo Full Speed, y de 1024 para uno de Hifh Speed.
Como el máximo tamaño de datos va a afectar a los requerimientos de ancho de banda del bus, es recomendable utilizar un tamaño de datos conservativo.

Transferencias en Masa (Bulk)

Las transferencias Bulk son usadas para grandes cantidades de datos, por ejemplo, cuando se realiza una impresión y los datos son enviados hacia la impresora.
Este tipo de transferencias provee de corrección de errores, tiene un campo de CRC16 en el bloque de datos. En el caso de recibir un bloque erróneo se retransfiere.
Las transferencias en masa utilizan un ancho de banda libre o no designado en el bus. Si el bus esta ocupado con transferencias sincrónicas y o interrupciones, entonces los datos en masa se moverán lentamente sobre el bus. Como se puede apreciar, este tipo de transferencia no garantiza ningún nivel de latencia.

Características:

- Se usan para transferir grandes cantidades de datos.
- Proveen de detección de errores CRC16 con garantizando que se va a recibir toda la información.
- No se garantiza un ancho de banda ni una minima latencia.
- Utilizan pipes del tipo Stream unidireccionales.
- Solo se utiliza en modos de velocidad High y Full.

Subsistema USB

Con el objetivo de operar los dispositivos USB de forma eficiente y transparente, es necesario contar con un módulo especial para controlar los mismos: a este módulo se lo conoce como subsistema USB. El mismo está compuesto por diversos componentes, los cuales colaboran para operar un dispositivo durante su ciclo de vida en el host.

Consideramos una estructura basada en cuatro partes distintas. Por supuesto, la misma no es obligatoria, pero dada la flexibilidad que ofrece, se hace muy recomendable.

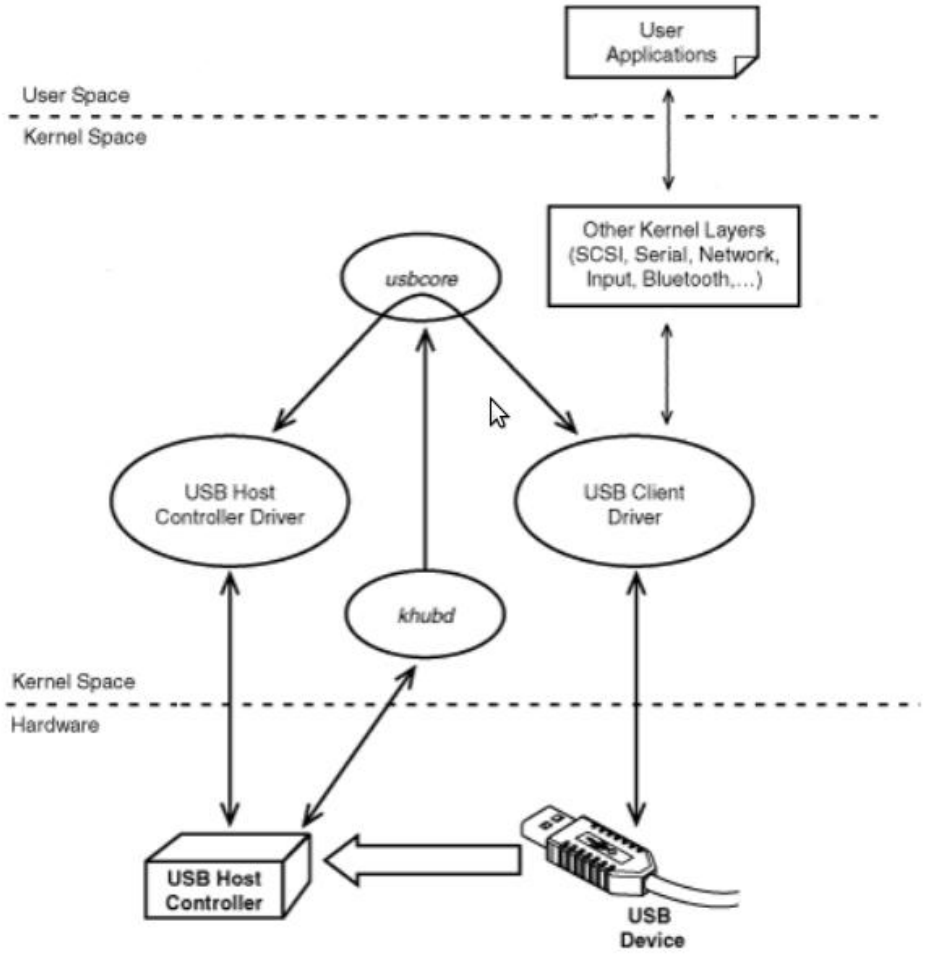
Las partes constitutivas son las siguientes:

HCDs (host controller drivers) que manejan diferentes controladores de host.

El núcleo USB es una base de código compuesto de rutinas y estructuras a disposición de los HCD y drivers del cliente. Ofrece indirección a estas dos partes, es decir, desacoplando los drivers del cliente de los drivers del controlador.
Un driver de hub, para el hub central (y hubs físicos) y un proceso de ayuda, khubd, que vigila todos los puertos conectados al hub. La detección de cambios de estado en los puertos y la configuración “en caliente” de dispositivos consume mucho tiempo, por lo que se logran mayores resultados usando un proceso independiente. El proceso khubd está dormido de forma predeterminada y despierta cada vez que detecta un cambio de estado en algún puerto USB.

Los controladores de dispositivo, para dispositivos USB cliente particulares
Aunque no son parte del subsistema USB, sí son necesarios para el funcionamiento de extremo a extremo, la asistencia de otros subsistemas kernel. Para apoyar a los dispositivos USB de almacenamiento masivo, por ejemplo, el subsistema USB trabaja en conjunto con los controladores SCSI; para conducir los teclados USB Bluetooth, los actores son cuatro: el subsistema USB, la capa de Bluetooth, el subsistema de la entrada, y la capa de tty.

Todo lo comentado anteriormente puede verse en el siguiente gráfico:



Núcleo USB

Es una API específica para trabajar con dispositivos USB y los controladores de host. Su propósito es abstraer las partes dependientes del hardware o el dispositivo, mediante la definición de un conjunto de estructuras de datos y funciones.

El núcleo USB contiene rutinas comunes a todos los controladores de dispositivos USB y los controladores de host.

Intentaremos ahora dar una visión general de su estructura, explicando los puntos de entrada y el uso de funciones de la API.

Drivers del controlador de host USB

Los controladores de host USB se encuentran estandarizados, y cumplen con alguna de las cuatro especificaciones que existen en el mercado. Esto debemos tenerlo en cuenta para el desarrollo adecuado de los drivers de los controladores.

Las cuatro especificaciones son las siguientes:

- Universal Host Controller Interface (UHCI): La especificación UHCI fue creada por Intel
- Open Host Controller Interface (OHCI): especificación tiene su origen en las empresas Compaq y Microsoft. Un controlador compatible con OHCI tiene más inteligencia incorporada en el hardware que uno UHCI, por lo que un driver para un host contrlooer OHCI es relativamente más simple
- Enhanced Host Controller Interface (EHCI): este es el controlador de host que admite dispositivos USB 2.0. Suelen venir acompañados de controladores OHCI o UHCI para manejar los dispositovs más lentos

Enumeración

Antes de que las aplicaciones puedan comunicarse con un dispositivo, el host necesita saber sobre el dispositivo y asignarle un controlador adecuado. La enumeración es el intercambio de información que cumple estas tareas. El proceso incluye asignar una dirección al dispositivo, leer los descriptores del dispositivo, asignar y cargar el controlador del dispositivo, y seleccionar una configuración específica según los requerimientos de energía, endpoints, etc.

Luego de concluido este proceso, el dispositivo se encuentra listo para transferir datos usando cualquiera de los endpoints de su configuración.
Durante la enumeración, un dispositivo atraviesa seis estados: encendido, default, direccionado, configurado, conectado y suspendido. En cada estado, el dispositivo tiene distintos comportamientos y capacidades.

A continuación se expone la secuencia típica de pasos que logran la enumeración de los dispositivos cuando se conecta un Pendrive USB en un equipo.

1. El Hub raíz informa de un variación de tensión en el puerto, debido a la conexión del dispositivo. El controlador del hub detecta este cambio de estado y despierta al proceso khubd. El dispositivo cambia al estado encendido.
2. El proceso khubd descifra la identidad del puerto USB que ha sufrido un cambio de estado. En este caso, es el puerto donde se enchufa el pendrive. A continuación, realiza un reinicio del dispositivo. El mismo cambia al estado default.
3. A continuación, khubd elige una dirección de dispositivo entre 1 y 127 , y lo asigna al bulk endpoint del pendrive, utilizando utilizando un URB de control conectado al endpoint 0. El dispositivo pasa al estado direccionado.
4. El proceso khubd utiliza el mismo URB de control, para obtener el descriptor del dispositivo de la unidad. A continuación, solicita descriptores de

configuración del dispositivo y selecciona el adecuado. En el caso del pendrive, un sólo descriptor se ofrece.
5. El proceso khubd pide al núcleo USB que enlace un controlador de cliente indicado para el dispositivo que se ha insertado.
6. Luego, khubd invoca al método probe() del controlador de cliente asociado al dispositivo. A partir de este momento, el controlador de almacenamiento masivo es responsable de la operación normal del dispositivo. El mismo pasa al estado configurado.

Como se habrá visto, hay dos estados que no fueron tenidos en cuenta en los pasos anteriores de la enumeración. Esto se debe a que mismos ocurren en circunstancias especiales:

- conectado: si el hub no provee energía a la línea del bus, el dispositivo cambia al estado conectado. La ausencia de energía puede deberse a alguna condición de error o a una solicitud explícita para remover el dispositivo. En este estado el host y el dispositivo no pueden comunicarse.

- suspendido: un dispositivo entra en este estado cuando no se detecta actividad en el bus por más de 3 milisegundos.

Detalles especiales de la enumeración:

- No debe asumirse que los requerimientos o eventos van a ocurrir en un orden específico. La especificación USB no dice nada sobre el orden de las operaciones.
- Estar listo para abandonar la transferencia de control, o terminarla antes. Por ejemplo, si se recibe un nuevo paquete de setup, un dispositivo debe abandonar cualquier transferencia en curso y comenzar una nueva.
- No intentar enviar más datos que los que el host require.
- Estar listo para cambiar al estado suspendido.
- Probar con diferentes tipos de controladores de dispositivo.

Conexión y desconexión de dispositivos

La estructura del controlador usb especifica dos funciones que el núcleo USB llama en los momentos requeridos.
En primer lugar, la función probe es llamada cuando un dispositivo es insertado en el host, y el núcleo USB supone cuál es el controlador que lo debe manejar; la función probe debe realizar controles sobre la información que se le pasa sobre el dispositivo y decidir si es el controlador es realmente apropiado para ese dispositivo. Además, el controlador debe inicializar las estructuras locales que requiera para administrar el dispositivo y guardar toda la información que necesita sobre el dispositivo; por ejemplo, el tamaño del buffer y la dirección del endpoint.
La función disconect se llama cuando el controlador debe dejar de controlar el dispositivo y puede hacer limpieza de variables y estructuras creadas para tal fin. Cuando un dispositivo es removido, la dirección queda disponible para un nuevo dispositivo.

El driver USB en Linux

En Linux existe un sistema llamado “Núcleo USB” o “USB Core” que elimina toda la dependencia de los dispositivos físicos mediante estructuras de datos y funciones estándar.
Este núcleo se comunica tanto con dispositivos USB como con el host mediante dos API’s. Una API superior, encargada de la comunicación con los dispositivos y otra inferior encargada de la comunicación con el host.

Estructuras de datos del Framework USB

La siguiente figura muestra la estructura necesaria para registrar un device USB.

```
struct usb_driver {
const char *name;

void * (*probe)(struct usb_device *, unsigned int,
const struct usb_device_id *id_table);
void (*disconnect)(struct usb_device *, void *);

struct list_head driver_list;

struct file_operations *fops;
int minor;
struct semaphore serialize;
int (*ioctl) (struct usb_device *dev, unsigned int code,
void *buf);
const struct usb_device_id *id_table;
};
```

Estructuras de descriptores de datos

El subsistema USB en Linux tiene una estructura jerárquica de descriptores que extienden o incluyen los descriptores estándares del USB dentro de un subsistema específico. Esta estructura contiene punteros hacia las configuraciones e interfaces necesarias.

```
struct usb_device{
  struct usb_config_descriptor *actconfig;
  struct usb_device_descriptor descriptor;
  struct usb_config_descriptor *config;
}
```

Estructuras de transferencia de datos (URB's)

Para transferir los datos el subsistema USB de Linux utiliza una única estructura llamada USB Request Block (URB) que contiene todos los parámetros necesarios para realizar una transferencia vía USB de cualquier tipo.
Todas las transferencias son enviadas de forma asincrónica y el request se completa con una función de callback.

Estructura de un URB

```
typedef struct
{
  unsigned int offset;       // offset to the transfer_buffer
  unsigned int length;       // expected length
  unsigned int actual_length;// actual length after processing
  unsigned int status;       // status after processing
} iso_packet_descriptor_t, *piso_packet_descriptor_t;

struct urb;
typedef void (*usb_complete_t)(struct urb *);

typedef struct urb
{
     spinlock_t lock;
     void *hcpriv;              // private data for host controller (don't care)
     struct list_head urb_list; // list pointer to all active urbs (don't care)
>CO  struct urb* next;          // pointer to next URB
>CM  struct usb_device *dev;    // pointer to associated USB device
>CM  unsigned int pipe;         // pipe information
<C   int status;                // returned status
TCO  unsigned int transfer_flags;//USB_DISABLE_SPD|USB_ISO_ASAP|USB_URB_EARLY_COMPL
>CM  void *transfer_buffer;     // associated data buffer
>CM  int transfer_buffer_length; // data buffer length
<C   int actual_length;         // actual data buffer length
     int bandwidth;             // allocated bandwidth
<X-- unsigned char *setup_packet;// setup packet (control only)
T-XX int start_frame;           // start frame (iso/irq only)
>--X int number_of_packets;     // number of packets in this request (iso only)
>-X- int interval;              // polling interval (irq only)
<--X int error_count;           // number of errors in this transfer (iso only)
>XXX int timeout;               // timeout in jiffies

>CO  void *context;             // context for completion routine
>CO  usb_complete_t complete;   // pointer to completion routine

>--X iso_packet_descriptor_t  iso_frame_desc[0]; // optional iso descriptors
} urb_t, *purb_t;
```

Los elementos marcados con una "C" son comunes a todos los tipos de transferencias.
Los elementos marcados con un signo ">" son parámetros de entrada.
Los elementos marcados con una "M" son obligatorios y los marcados con una "O" opcionales.

Los principales parámetros son los siguientes

- ***dev:** Puntero al USB device.
- **pipe:** Es un entero que contiene el número de endpoint. Existen diferentes funciones que al llamarlas devuelven un número de pipe para asignarle a éste parámetro.
- ***transfer_buffer:** Es un puntero hacia el buffer que va almacenando los datos enviados desde o hacia el dispositivo.
- ***transfer_buffer_length:** Tamaño del buffer en bytes.

Funciones para controlar las URB

En el USB Core existen cuatro funciones encargadas de manejar los URB.

- **usb_alloc_urb:** Cuando se necesita una estructura URB se llama a esta función. Devuelve un puntero a una URB vacía o NULL en caso de error.
- **usb_free_urb:** Libera la memoria reservada por la función anterior. Recibe como parámetro el puntero a la urb a liberar.
- **usb_submit_urb:** Envía un request de transferencia asincrónico al USB Core. Recibe como parámetro un puntero a la URB creada e inicializada con la primer función. Devuelve 0 en caso de no haber errores o de lo contrario el código de el error encontrado.
- **usb_unlink_urb:** Esta función cancela un request programado antes de que se complete. También recibe como parámetro un puntero a la URB con la que s envió el request de transferencia con la función anterior. Puede ser utilizada de forma sincrónica o asincrónica dependiendo de la bandera transfer_flag.

Anatomía del subsistema SCSI de Linux

Una introducción a la arquitectura en capas SCSI

M. Tim Jones, Ingeniero Consultor, Emulex Corp.

Introducción

El Small Computer System Interface (SCSI) es un conjunto de normas que definen la interfaz y protocolos para la comunicación con un gran número de dispositivos (en su mayoría relacionadas con el almacenamiento). El subsistema SCSI permite la comunicación con estos dispositivos. El SO mediante una arquitectura de capas une a los controladores de alto nivel, tales como controladores de disco, usb massive storage o DVD/CD-ROM, con una interfaz física.

SCSI implementa una arquitectura de comunicaciones del estilo cliente / servidor. Un iniciador envía una solicitud de comando a un dispositivo de destino. El destino procesa la solicitud y devuelve una respuesta al iniciador. Un iniciador puede representar un dispositivo SCSI en un ordenador central, y un destino SCSI puede ser una unidad de disco, CD-ROM, unidad de cinta, etc.

Comandos SCSI

Mientras que los protocolos sobre los que se transporta SCSI han cambiado a lo largo de los años, el conjunto de comandos SCSI conserva muchos de sus elementos originales. Un comando SCSI se define dentro de un bloque descriptor del comando (CDB). El CDB contiene un código de operación que define la operación especial a llevar a cabo y una serie de parámetros específicos de la operación.

Los comandos SCSI soportan la lectura y escritura de datos (cuatro variantes de cada uno) y también un número de comandos que no son de datos como los de prueba de unidad lista (es el dispositivo preparado), investigación (recuperar la información básica sobre el dispositivo de destino), lectura de la capacidad (recuperar la capacidad de almacenamiento del dispositivo de destino), y muchos otros. Los comandos particulares dependerán del tipo de dispositivo. Un iniciador identifica el tipo de dispositivo a través del comando de consulta. En la tabla se muestran los comandos SCSI más comunes.

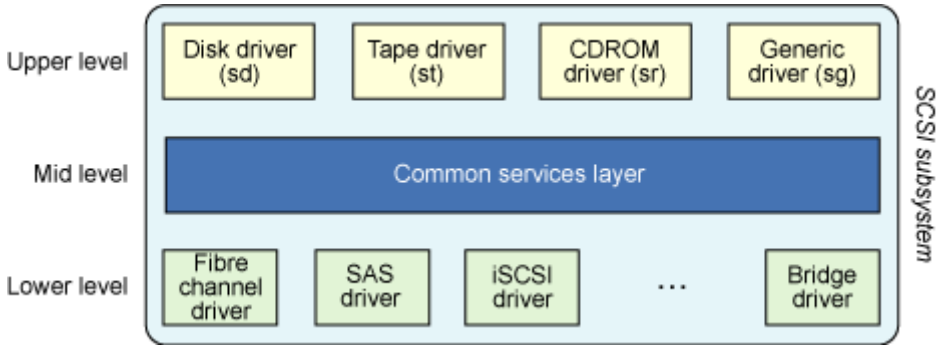
Comandos	Propósito
Test unit ready	Pregunta si el dispositivo está listo para las transferencias
Inquiry	Solicitud de información básica sobre el dispositivo
Request sense	Solicitud de información de error de un comando anterior
Read capacity	Solicitud de información sobre la capacidad de almacenamiento
Read	Leer datos desde el dispositivo
Write	Escribir datos desde el dispositivo
Mode sense	Solicitud de modo paginado (parámetros del dispositivo)
Mode select	Configurar los parámetros del dispositivo en modo paginado

El subsistema SCSI existe como una arquitectura de capas con tres capas distintas. En la parte superior es lo que se llama el nivel superior, que representa el más alto de la interfaz del kernel para SCSI y los controladores de los principales tipos de dispositivos.

El siguiente es el nivel medio, también llamado capa común o "unificador". En esta capa son los servicios comunes para los niveles superior e inferior de la pila SCSI.

Por último, está el nivel inferior, que representa los controladores reales de las interfases físicas que son aplicables a SCSI.

Arquitectura del subsistema SCSI



Se puede encontrar el código fuente para el subsistema SCSI (SCSI de nivel superior, nivel medio, y un gran número de drivers) en por ejemplo: /linux/drivers/scsi.
Las estructuras de datos SCSI pueden estar en el directorio de origen SCSI y también en. /linux/include/scsi.

Las Capas del Subsistema SCSI

Capa SCSI de nivel superior

El nivel superior del subsistema SCSI representa la interfaz de más alto nivel desde el kernel (el nivel de dispositivo). Se compone de un conjunto de drivers, tales como los dispositivos de bloque (disco SCSI y SCSI de CD-ROM) y los dispositivos de caracteres (cinta SCSI y SCSI genéricos). El nivel superior acepta peticiones desde arriba (el VFS por ejemplo) y las convierte en peticiones SCSI. La capa superior también completa los comandos SCSI y notifica a la capa de arriba un status.

El controlador de disco SCSI está implementado en /linux/drivers/scsi/sd.c.

El controlador de disco SCSI se inicializa con una llamada a register_blkdev (como un controlador de bloque) y proporciona un conjunto de funciones que representan todos los dispositivos SCSI con scsi_register_driver. Dos de interés aquí son sd_probe y sd_init_command.

Cada vez que un nuevo dispositivo SCSI está conectado a un sistema, la función sd_probe se llama desde la capa media SCSI. La función sd_probe determina si el dispositivo será gestionado por el controlador de disco SCSI y, si es así, crea una estructura scsi_disk nuevos para que la represente. El sd_init_command es la función que toma una solicitud de la capa de sistema de archivos y convierte a la solicitud en un comando SCSI para leer y para escribir (`sd_rw_intr` es llamado a completar estas peticiones de I/O).

Capa SCSI de nivel medio

El nivel medio de SCSI es una capa de servicios comunes, tanto para el nivel superior SCSI e inferior (implementado parcialmente en /linux/drivers/scsi/scsi.c). Ofrece una serie de funciones que son utilizadas por los controladores de nivel inferior y de nivel superior y, por tanto, sirve de unión entre estas dos capas distintas. Esta capa es importante, ya que abstrae la implementación de los drivers de menor nivel (LLD), implementado parcialmente en /linux/drivers/scsi/hosts.c.

El registro del driver de bajo nivel y el control de errores son proporcionados por esta capa. El nivel medio también ofrece comandos SCSI de encolado entre los niveles superior e inferior. Un aspecto clave del nivel medio de SCSI es la conversión de las solicitudes de comando de la capa superior en las solicitudes SCSI. También gestiona la recuperación de errores de dispositivos SCSI específicos. El handler de error y timeout SCSI es implementado en /linux/drivers/scsi/scsi_error.c.

La capa intermedia fundamentalmente actúa como un intermediario entre los niveles superior e inferior del subsistema SCSI. Acepta las solicitudes de transacciones SCSI y las colas para su procesamiento (como se muestra en. /linux/drivers/scsi/scsi_lib.c). Cuando estos comandos se han completado, se recibe la respuesta SCSI del LLD y realiza la notificación al nivel superior de la terminación de la solicitud.

Capa SCSI de bajo nivel

El nivel inferior es una colección de controladores llamados los controladores SCSI de bajo nivel. Estos son los drivers específicos de la interfaz de los dispositivos físicos tales como adaptadores de bus host. La LLD proporciona una abstracción de la capa intermedia común para el dispositivo específico de HBA. Cada LLD proporciona la interfaz para el hardware subyacente particular, sino que utiliza el conjunto estándar de interfaces para la capa intermedia.
El nivel inferior contiene la mayor cantidad de código ya que representa las variaciones de los tipos de adaptadores SCSI. Por ejemplo, el protocolo Fibre Channel incluye LLD para los adaptadores de Emulex y QLogic. Los LLDs para adaptadores SAS incluyen los de Adaptec y LSI.

El driver Massive Storage y sus interacciones con el subsistema SCSI

Los dispositivos USB de almacenamiento masivo adhieren al protocolo Small Computer System Interface (SCSI) para comunicarse con los sistemas host. Los bloques de acceso a los dispositivos de almacenamiento USB es por lo tanto, encaminado a través del subsistema SCSI del kernel.

El driver massive storage se registra como un adaptador SCSI virtual. El adaptador virtual se comunica a través de flujos de entrada y flujos de salida utilizando comandos SCSI URBS. Un disco USB aparece a las capas superiores como un dispositivo SCSI conectado a este adaptador virtual.

Para comprender mejor las interacciones entre las capas USB y SCSI, vamos a implementar una modificación en el controlador USB massive storage. El nodo usbfs /proc/bus/usb/devices, contiene las propiedades y detalles de todos los dispositivos USB conectados al sistema. La linea "T" en el archivo */proc/bus/usb/devices* output, por ejemplo, contiene el número de buses, la profundidad desde el Hub raíz, la velocidad de funcionamiento, y así sucesivamente. La linea "P:" contiene el ID de proveedor, Id. de producto y número de revisión del dispositivo. Toda la información disponible en */proc/bus/usb/devices* es gestionado por el subsistema USB, pero **hay una pieza faltante que está bajo la jurisdicción del subsistema SCSI.**

El nombre del nodo /dev asociado con el dispositivo USB de almacenamiento (sd [az] [1-9] para los discos y sr [0-9] para CD-ROM) no está disponible en /proc/bus/usb/devices. Para superar esta limitación, vamos a añadir una linea "N:" que muestra el nombre asociado en /dev .

_____

Para entender el código 11.6, primero vamos a trazar el flujo de código, continuando desde donde lo dejamos en la sección de "enumeración".

En esa sección, se inserta un pendrive USB y se sigue el tren de la ejecución hasta que la invocación de storage_probe (), el médodo probe() del controlador de massive storage. Pero avanzando:

1. storage_probe () registra un adaptador SCSI virtual llamando al `_add_host()`, se le suministra una estructura de datos privada llamada us_data como argumento. scsi_add_host () devuelve una estructura scsi_host para este adaptador virtual, con espacio al final para us_data.
2. Se inicia un hilo del núcleo (kernel thread) llamado **usb-storage** para manejar todos los comandos SCSI en cola para el adaptador virtual.
3. Se programa un hilo del núcleo llamado *usb-stor-scan* que se pide a la capa SCSI de nivel medio para explorar el bus en búsqueda de dispositivos conectados.
4. El hilo de exploración iniciado en el Paso 3 descubre la presencia del Pendrive insertado y enlaza el driver de disco de la capa de nivel superior SCSI (sd.c) al dispositivo. Esto da lugar a la invocación del método de sondeo del driver de disco SCSI, sd_probe ().
5. El controlador sd asigna un /dev /sd * nodo a el disco. A partir de ahora, las aplicaciones utilizan esta interfaz para acceder al disco USB. El subsistema de disco SCSI encola los comandos al adaptador virtual, los cuales el hilo del núcleo usbstorage maneja con los URBS apropiados.

Ahora que sabes lo básico, vamos a inspeccionar el listado 11.6, mirando a las adiciones de estructuras de datos en primer lugar. El listado agrega un campo `snam` a la estructura scsi_host para mantener la asociación SCSI / nombre de /dev que esta interesado en él. También agrega un campo privado en la estructura usb_interface para asociar cada interfaz USB con sus us_data.

Como puede ver, el subsistema SCSI ha asignado al Pendrive la denominación sda, sr0 para el CD-ROM, y sdb para el disco duro. Las aplicaciones en el espacio de usuario por operar en estos nodos para comunicarse con los dispositivos respectivos. Es de destacar que con la llegada de udev, sin embargo, usted tiene la opción de crear abstracciones de alto nivel para identificar a cada dispositivo sin necesidad de depender de la identidad de los nombres / dev asignados por el subsistema SCSI.

Conclusiones

Como vimos a lo largo de la investigación, el desarrollo de un driver USB es una tarea compleja que requiere un buen diseño en capas y soporte de muchos servicios del kernel. Algunos de ellos ya se encuentran implementados o lo estarán próximamente. Sin embargo, muchos de ellos todavía se requieren.
En primer lugar, se hace necesario un mecanismo para compartir datos entre procesos y sincronizar los mismos. Esto permite una comunicación más rápida y fiable a la hora de configurar los dispositivos e inicializarlos.
Al mismo tiempo, se necesita contar con la posibilidad de asignar memoria de forma dinámica, para poder crear las estructuras de datos pertinentes durante la ejecución.
El factor más crítico para la concreción de un driver para unidades de almacenamiento masivo USB es la existencia de una capa de datos del kernel que provea servicios de alto nivel para acceder a los dispositivos de almacenamiento por parte de los procesos de usuario. Estamos hablando del subsistema SCSI, que interactúa en forma constante con el subsistema USB para el acceso a datos de los dispositivos. Dicho de otra forma, el subsistema USB es incapaz de funcionar por sí solo, y requiere de las otras partes del kernel para trabajar adecuadamente.

Histórico de cambios / testeos realizados

17-08-2010

- Prueba con otro sistema operativo (Kolibri) en Lenovo
    - Usando imagen de CD no lo detecta
    - Usando imagen de floppy mensaje: “Starting System – Kernel MNT
- Prueba con otro sistema operativo (Kolibri) en Acer
    - Usando imagen de diskette bootea (la misma imagen que no bootea en Lenovo)

20-08-2010

- Prueba de SODIUM en Lenovo Thinkcentre. Con imagen de CD (ISO).

Las pruebas realizadas en las desktops Lenovo(IBM) Thinkcentre fueron exitosas, solo detectamos fallas al ejecutar el comando ls.

24-08-2010

- Prueba con otro sistema operativo (Kolibri) utilizando la aplicación “Lili” en FAT-32
    - Con imagen de CD (ISO) mensaje “Boot Error”
    - Usando imagen de floppy mensaje: “Starting System – Kernel MNT ”
- Prueba de SODIUM en Lenovo
    - Con SYSLinux mensaje: “Starting System – Kernel MNT ”

Modificamos parámetros de la BPB de SODIUM

31-08-2010

- Modificamos sodium.asm tratando de imprimir un mensaje antes de habilitar la compuerta 20 (suponemos el error en ese lugar).
- Imprimimos el mensaje de prueba en la línea 375 de sodium.asm
- Ahora suponemos que el error está después de habilitar la compuerta 20. Hasta ahora no se limpia la pantalla y se rebootea.
- Ahora imprimimos el mensaje en la línea 350. Se imprime correctamente.

07-10-2010

- Queremos saber si sodium.asm llama a main.c en la máquina Lenovo para saber si el error está en sodium.asm o en el main. Por eso imprimimos un mensaje antes del jump y otro en el main.

- Imprimimos un mensaje en la línea 390 de sodium.asm. La máquina lenovo muestra mensajes raros. ‘ S ’
- Imprimimos un mensaje en la línea 432 al comienzo de la función “jContinuar”. Otra vez obtenemos mensajes raros. ‘ S ’
- Ponemos un break en la línea 483, antes del salto a 32 bits. La pantalla muestra un color verde.
- Cambiando el flag en la línea 489 cambió el color (Lila) y se muestra “Iniciando SODIUM”.

14-10-2010

- Estábamos haciendo referencia a segmentos que tal vez no estuvieran definidos y por eso se reinicia la máquina en la línea 524 de sodium.asm, justo antes del salto a main.c
- Imprimimos un recuadro justo antes del jump al main.c en sodium.asm. Tarda un tiempo y se imprime el recuadro.
- Buscamos la dirección del main. Obtenemos 0x1C93C

Tabla de Pruebas

Prueba 1	**Hardware**	**HP – AMD Athlon dualcore M300 Ghz**
	SO	Windows 7
	Bios	F.03
	Problema	No reconoce el dispositivo usb para booteo.
	Resultado	Ninguno

Prueba 2	**Hardware**	**Acer 5536 – AMD Turion 64 x2**
	SO	Kubuntu 10.4
	Bios	Phoenix
	Problema	Ninguno
	Resultado	éxito

Prueba 3	**Hardware**	**Acer 5100 – AMD Turion 64 x2**
	SO	Ubuntu 8.10
	Bios	Phoenix V2.73
	Problema	Ninguno
	Resultado	éxito

Prueba 4	**Hardware**	**Acer 3000 – AMD Sempron**
	SO	Windows XP
	Bios	Phoenix
	Problema	Ninguno
	Resultado	éxito

Prueba 5	**Hardware**	**Commodore – KE 8326-MB**
	SO	Windows 7
	Bios	Phoenix Secure Core version 1.0B-1646-0018
	Problema	Ninguno
	Resultado	éxito

Prueba 6	**Hardware**	**Asus - P4S800-MX SE - CPU Intel 800 Mhz**
	SO	Windows 7
	Bios	AMI BIOS SM 2.3
	Problema	Ninguno
	Resultado	éxito

Prueba 7	**Hardware**	**Compaq presario v3000**
	SO	Windows XP
	Bios	Phoenix F.21
	Problema	Ninguno
	Resultado	éxito

Prueba 8	**Hardware**	**Lenovo T61**
	SO	Windows 7
	Bios	210(7L0BWW)
	Problema	Ninguno
	Resultado	éxito

Prueba 9	**Hardware**	**Lenovo (IBM) intel Core 2 Duo**
	SO	Windows 2003 Server
	Bios	Phoenix (tipo 2043A, fecha 11/8/10, rev. 20ICT45A5P)
	Problema	
	Resultado	Éxito

Prueba 10	**Hardware**	**CLON AMD 64X2 – 5200+**
	SO	Windows XP
	Bios	Award Bios V6,00PG
	Problema	Ninguno
	Resultado	éxito

Prueba 11	**Hardware**	**Lenovo T400**
	SO	Windows XP
	Bios	Ver 1.00PARTTBLX
	Problema	Ninguno
	Resultado	Éxito

Prueba 12	**Hardware**	**Lenovo T41**
	SO	Windows XP
	Bios	210(7L0BWW)
	Problema	Ninguno
	Resultado	Éxito

Prueba 13	**Hardware**	**Lenovo T42**
	SO	Windows XP
	Bios	210(7L0BWW)
	Problema	Ninguo
	Resultado	Éxito

Prueba 14	**Hardware**	**Lenovo T43**
	SO	Windows XP
	Bios	210(7L0BWW)
	Problema	Niguno
	Resultado	Éxito
Prueba 15	**Hardware**	**Commodore – KE 8326-MB**
	SO	Windows 7
	Bios	Phoenix Secure Core version 1.0B-1646-0018
	Problema	Ninguno
	Resultado	éxito
Prueba 16	**Hardware**	**Toshiba Satéllite L455 Core Duo**
	SO	Windows 7
	Bios	Phoenix Sp2903 A
	Problema	Ninguno
	Resultado	éxito

Prueba 17	**Hardware**	**Lenovo (IBM) intel Core 2 Duo**
	SO	Windows 2003 Server
	Bios	Phoenix (tipo 2043A, fecha 11/8/10, rev. 20ICT45A5P)
	Problema	
	Resultado	Éxito
Prueba 18	**Hardware**	**Lenovo (IBM) intel Core 2 Duo**
	SO	Windows 2003 Server
	Bios	Phoenix (tipo 2043A, fecha 11/8/10, rev. 20ICT45A5P)
	Problema	
	Resultado	Éxito

Bibliografía

1.1 Essential Linux Device Drivers [VEN08]

1.1.1 Venkateswaran, Sreekrishnan
1.1.2 Essential Linux Device Drivers
1.1.3 Prentice Hall
1.1.4 2008

1.2 Linux Device Drivers, 3rd Edition [COR05]

1.2.1 Corbet, Jonathan
1.2.2 Greg Kroah-Hartman, Alessandro Rubini
1.2.3 Linux Device Drivers, 3rd Edition
1.2.4 O'Reilly
1.2.5 2005

==========

Sitios Web primer Q

1.3 Estudio de un Sistema Operativo [004]

1.3.1 Sergio Sáez, Juan Carlos Pérez,Vicente Lorente
1.3.2 Estudio de un Sistema Operativo
1.3.3 Universidad Politécnica de Valencia.
1.3.4 http://futura.disca.upv.es/~eso/

1.4 Operative System Developers [005]

1.4.1 OSDev.org
1.4.2 Estudio de un Sistema Operativo
1.4.3 http://wiki.osdev.org/Main_Page

1.5 Boot and run Linux from a USB flash memory stick [006]

1.5.1 Pen Drive Linux
1.5.2 Boot and run Linux from a USB flash memory stick
1.5.3 http://www.pendrivelinux.com/

1.6 Boot and run Linux from a USB flash memory stick [007]

1.6.1 Pen Drive Linux
1.6.2 Boot and run Linux from a USB flash memory stick
1.6.3 http://www.pendrivelinux.com/

1.7 Hello World Boot Loader [008]

1.7.1 Daniel Faulkner
1.7.2 Boot loader tutorial
1.7.3 http://www.osdever.net/tutorials/view/hello-world-boot-loader

1.8 Technische Universität München

1.8.1 http://www.lrr.in.tum.de/Par/arch/usb/usbdoc/node14.html

1.9 The Linux USB Subsystem

1.9.1 http://www.linux-usb.org/USB-guide/book1.html