Trabajo Práctico 2

Programación de Sistemas Operativos

1er Cuatrimestre 2011

1. Introducción

1.1. Objetivo

En este segundo trabajo práctico la propuesta es la implementación de controladores de hardware o *drivers*, con una interfaz común.

1.2. Entrega

Fecha límite de entrega: **Jueves 2 de Junio - 16hs**

La entrega del trabajo práctico debe realizarse en forma digital e impresa. Se debe enviar un correo electrónico a `pso-doc@dc.uba.ar` cuyo asunto (*subject*) sea "Entrega TP2 - Apellido1, Apellido2, Apellido3", conteniendo:

- Nombre, apellido y LU de los 3 (tres) integrantes del grupo
- Código fuente, en un archivo comprimido.
- Informe en formato digital.

Paralelamente, se debe entregar **impreso** el informe del trabajo práctico en una carpeta. No se debe imprimir el código fuente. Tampoco es necesario entregar el código en un medio de almacenamiento físico junto con el informe. El informe del trabajo debe contener la documentación de lo realizado, explicando *cómo* fue realizado.

1.3. Alcance

El alcance de esta primer entrega del trabajo llegará hasta cubrir los siguientes temas o unidades del kernel:

- Driver de teclado
- Driver de un *char device*: puerto serie
- Driver de un *block device*: Disco
- File System

1.4. Organización general

Para esta etapa, se agregan al trabajo realizado anteriormente nuevos módulos que exportan la funcionalidad pedida.

2. Char devices

Uno de las interfaces más comunes para acceder a distintos recursos que ofrece el sistema operativo es lo que llamamos un *char device*. Este tipo de dispositivo ofrece un mecanismo de acceso secuencial a un dispositivo. Esto se implementa genéricamente a través del siguiente `struct`:

```
typedef struct str_chardev chardev;

typedef sint_32(chardev_read_t)(chardev* this, void* buf, uint_32 size);
typedef sint_32(chardev_write_t)(chardev* this, const void* buf, uint_32 size);
typedef sint_32(chardev_seek_t)(chardev* this, uint_32 pos);
typedef uint_32(chardev_flush_t)(chardev* this);

struct str_chardev {
  uint_32 clase;
  uint_32 refcount;
  chardev_flush_t* flush;
  chardev_read_t* read;
  chardev_write_t* write;
  chardev_seek_t* seek;
} __attribute__((packed));
```

Las funciones `read` y `write` permiten leer y escribir del *char device* un buffer de tamaño fijo a continuación del punto *actual*.

La función `read` tiene una semántica bloqueante: Si no hay datos para leer, pero eventualmente puede llegar a haber datos para leer, entonces se bloquea a la espera de que llegue algún dato. Por otro lado, si hay datos disponibles para leer, los lee en el buffer, indicando la cantidad de bytes leídos como valor de retorno, los cuales podrían ser *menos* que el tamaño del buffer indicado. En caso de error se devuelve un número negativo.

Por su parte, la función `write` escribe en el dispositivo el buffer pasado por parámetro. Si no es posible escribir en el dispositivo, pero eventualmente se podrá, entonces la función se bloquea hasta que termine de escribir todo el buffer. En caso de que no sea posible escribir todo el buffer, entonces podría devolver un valor menor al tamaño del buffer pedido. En caso de error devuelve un valor negativo.

La función `seek` cambia la posición *actual* del dispositivo a la pasada por parámetro. Devuelve la posición actual del dispositivo o un valor negativo en caso de error.

La función `flush` cierra el dispositivo. Esta función es llamada cuando el contador de referencias `refcount` (contador de *file descriptors*) llega a 0, indicando que el archivo se debe destruir.

2.1. `device` - Devices

La tarea a nivel usuario no conoce la interfaz que provee el kernel sino un conjunto de *syscalls* que la utilizan. Para el usuario, un *char device* se identifica con un número entero, que llamamos *file descriptor*.

> **Ejercicio 1:** Implementar las syscalls `read`, `write`, `seek` y `close`, con la aridad propuesta, que dados un entero que denota un *file descriptor* y los respectivos parámetros llame a la función correspondiente del char device. Considere que la función podría no estar implementada (puntero en NULL) y que el descriptor proporcionado por el usuario podría ser inválido. Esta funciones devuelven un valor negativo en caso de error.

Ejercicio 2: Implementar la función `device_descriptor` que dado puntero a un *char device* asigne un número entero nuevo (o *file descriptor*) con el cual el usuario se referirá a él. Recuerde actualizar el contador de referencias.

2.2. `con` - Driver del consola

Se desea implementar un *driver* de consola que permita mostrar en pantalla y leer del teclado como si fuese un *char device*. De este modo, las tareas pueden utilzar la pantalla, un archivo, u otro dispositivo indistintamente.

El manejo de la consola será de la siguiente manera:

- Una consola ocupa toda la pantalla (80x25).
- Sólo una consola tiene el *foco*, que es la que se ve en pantalla y la que atrapa el teclado.
- Las teclas Alt+Shift+← y Alt+Shift+→ permiten alternar entre las consolas abiertas, de manera cíclica.
- El contenido de la pantalla se desplaza hacia arriba al aparecer una nueva línea, dando la ilusión de una consola en la pantalla.
- Al cambiar entre consolas, se cambia también el contenido de la pantalla.

Ejercicio 3: Implementar la función `con_init` que inicializa el módulo.

Ejercicio 4: Implementar la función `con_open` que abre una consola nueva y devuelve un puntero a un `chardev` con la nueva consola creada y el contador de referencias en 0.

Ejercicio 5: Implementar la rutina de atención de interrupción del teclado para que envíe la tecla correspondiente a la consola *actual* (la que tiene el foco). Tenga en cuenta que debe convertir los scan-codes del teclado en caracteres ASCII.

Ejercicio 6: Implementar las funciones de lectura de teclado (`read`) y de escritura en pantalla (`write`). Implementar la función `flush` que elimine la consola del anillo de consola.

Ejercicio 7: (opcional) Muestre un bonito ASCII-art en colores en caso de que no haya ninguna consola abierta.

2.3. `serial` - Driver del puerto serie

El puerto serie es un puerto de propósito general que permite a un sistema comunicarse con otros dispositivos externos, como un modem, un mouse o incluso otra computadora. El módulo `serial` implementa un *driver* de este puerto presente en la mayoría de las computadoras, que exporta una interfaz de *char device*.

Ejercicio 8: Implementar la función `serial_init` que inicialice el módulo.

Ejercicio 9: Implementar la función `serial_open` que dado el número de puerto devuelve un puntero a un `chardev` que representa el puerto serie en cuestión.

3. Block devices

Otra interfaz posible que puede exportar un driver es la de *block device* que permite acceder a un dispositivo cuya naturaleza no es de acceso secuencial sino aleatoreo. En algunos sistemas este acceso además está limitado solamente a un acceso de a *bloques*. En nuestro caso vamos a simplificar esto y acceder de a bytes, siendo responsabilidad de quien usa la interfaz leer o escribir regiones múltiplo del tamaño del bloque.

```
typedef struct str_blockdev blockdev;

typedef sint_32(blockdev_read_t)(blockdev* this, uint_32 pos, void* buf, uint_32 size);
typedef sint_32(blockdev_write_t)(blockdev* this, uint_32 pos, const void* buf, uint_32 size);
typedef uint_32(blockdev_flush_t)(blockdev* this);

struct str_blockdev {
  uint_32 clase;
  uint_32 refcount;
  blockdev_flush_t* flush;
  blockdev_read_t* read;
  blockdev_write_t* write;
  uint_32 size;
} __attribute__((packed));
```

Al igual que con los *char devices* se cuentan con las funciones de lectura, escritura y cierre del dispositivo. Sin embargo, en este caso las funciones de lectura y escritura reciben un valor más `pos`, que indica la posición a partir de la cual se debe leer o escribir. Además, el campo `size` indica el tamaño del dispositvo. Las lecturas o escrituras fuera de ese tamaño darán error.

Importante: Debe implementar **al menos uno** de los dos siguientes módulos: `fdd`, `hdd`.

3.1. `fdd` - Floppy Disk Driver

Este módulo implementa el acceso a un diskette. Nos vamos a limitar a diskettes de $3\frac{1}{2}$ de 1.44MB de capacidad. Para ello, debe ofrecer una interfaz de *block device* que permita acceder al dispositivo. Tenga en cuenta que sólo una tarea puede acceder al dispositivo al mismo tiempo al dispositivo físico.

Ejercicio 10: Implementar la función `fdd_init` que inicialice el módulo. Tenga en cuenta que esta función se llama de un punto de la inicialización del kernel donde todavía no cuenta con interrupciones.

Ejercicio 11: Implementar la función `fdd_open` que dado el número de floppy disk drive, iniciando en 0, devuelva un puntero a un *block device* que represente este dispositivo.

Ejercicio 12: Implementar la función de lectura del dispositivo por bloques de 512 bytes. Es decir, sólo es necesario soportar lecturas alineadas a 512 bytes, de un tamaño múltiplo de 512.

Ejercicio 13: *(opcional)* Implementar la función de escritura del dispositivo con las mismas restricciones de tamaño.

3.2. `hdd` - Hard Disk Driver

Este módulo implementa el acceso a un disco rígido ATA IDE. Para esto se debe ofrecer una interfaz de *block device* que permita acceder al dispositivo. Tenga en cuenta que sólo una tarea puede acceder al dispositivo al mismo tiempo al dispositivo físico.

Ejercicio 14: Implementar la función `hdd_init` que inicialice el módulo. Tenga en cuenta que esta función se llama de un punto de la inicialización del kernel donde todavía no cuenta con interrupciones.

Ejercicio 15: Implementar la función `hdd_open` que dado el número de disco rígido (iniciando en 0: primary master, primary slave, secondary master, secondary slave) devuelva un puntero a un *block device* que represente este dispositivo. Esta función debe devolver NULL si el disco no está presente.

Ejercicio 16: Implementar la función de lectura del dispositivo por bloques del tamaño de un sector físico del disco. Es decir, sólo es necesario soportar lecturas alineadas al tamaño de un sector físico.

Ejercicio 17: *(opcional)* Implementar la función de escritura del dispositivo con las mismas restricciones de tamaño.

4. File System

Uno de los recursos que ofrece un sistema operativo es un sistema de archivos. Un lugar para almacenar persistentemente archivos y accederlos. Sin embargo, esta interfaz permite ver como un único árbol de directorios varios sistemas de archivos. En este trabajo se implementará algún sistema de archivos con ciertas limitaciones: se ignoran las cuestiones de permisos de usuario y sólo se implementa lectura.

4.1. `fs` - Sistema de archivos

El sistema de archivo puede representar tanto archivos físicos de uno o más medios de almacenamiento, como dispositivos o archivos.

En este trabajo implemetaremos algunas simplificaciones. No hay una noción de *directorio actual*, todas las rutas son absolutas separadas por el caracter "`/`". El *montaje* u organización del filesystem se configurará de manera estática en el código. El único disco disponible se debe encontrar en "`/disk/`", mientras que los dispositivos se deben encontrar en "`/`".

Ejercicio 18: Implementar la función `fs_open` que permita abrir un archivo o dispositivo devolviendo el correspondiente puntero al *char device*.

Ejercicio 19: Implementar la syscall `open` que dado el nombre de archivo y el modo de apertura devuelva un *file descriptor* al archivo o dispositivo abierto. En caso de error debe devolver un número negativo.

4.2. Implementación del sistema de archivos

Un sistema de archivos ofrece escencialmente una función `open` que devuelve un *char device* que representa el archivo abierto. Luego, a este char device es posible leerlo, escribirlo, etc.

Ejercicio 20: Implementar **al menos uno** de los siguientes sistemas de archivos: FAT12, FAT16, FAT32 o ext2. Implementar la función *`mod`*`_open` del sistema correspondiente que dado un nombre de archivo con su ruta *absoluta* devuelva el *char device* correspondiente a ese archivo.

Ejercicio 21: Implementar la función de lectura (`read`) del *char device* que permita leer el archivo hasta el final.

Ejercicio 22: Implementar la función `seek` para los archivos.

Ejercicio 23: *(opcional/difícil)* Implementar la función `write` para los archivos. Tenga en cuenta que necesitará cierto mecanismo para que dos tareas no puedan corromper el sistema al escribir sobre el mismo archivo abierto. Tenga en cuenta además las opciones de apertura de los archivos en la función `open`.

- `fat12` - Sistema de archivos FAT12
  Dado un *block device* de un diskette implemente un sistema de archivos FAT12 que ofrezca las funciones pedidas.
- `fat16` o `fat32` - Sistema de archivos FAT16 o FAT32
  Dado un *block device* que represente una *partición* formateada con FAT16 o FAT32 implemente un sistema de archivos sobre él.
- `ext2` - Sistema de archivos ext2
  Dado un *block device* que represente una *partición* formateada con ext2 implemente un sistema de archivos, ignorando las cuestiones de permisos de usuarios.

Para el caso de los sistemas de archivo sobre un disco rígido tenga en cuenta que no debe utilizarse una tabla de particiones sino que se debe formatear el disco entero.

5. Tareas

Ejercicio 24: Implementar la syscall `run(const char* archivo)` que cree una nueva tarea cargando el contenido desde el archivo proporcionado. En caso de error se debe devolver un número negativo. En caso de éxito, se debe devolver el valor del identificador de proceso recientemente creado.

Ejercicio 25: Programar una tarea `init` que sea la única tarea que se incluya junto con el `kernel.bin` y que cargue las demás tareas desde los archivos `.pso` utilizando la syscall `run`.

Ejercicio 26: Programar una tarea `console` que cree una consola, lea del teclado comandos (uno por línea) y los ejecute cargando el correspondiente archivo `.pso` del disco, si existiera.