Trabajo Práctico 2 ∼ PSO

Liptak, Leandro - leandroliptak@gmail.com
Reboratti, Patricio - darthpolly@gmail.com
Silvani, Damián - dsilvani@gmail.com

June 16, 2011

Documentación

Char Devices

`device` - Devices

La mayor parte del diseño del módulo `device` se centró en cómo almacenar los descriptores de archivos y en como debían ser implementadas las llamadas al sistema. Respecto al primer punto, decidimos almacenar los FDs (*file descriptors*) en una matriz de PID por FD, acotada por el máximo process id y máximo file descriptor (ambos con valor actual 32). Dicha matriz contiene punteros a descriptores de dispositivos y se considera una entrada disponible aquella con valor 0 (puntero nulo). Entonces, la cantidad de memoria empleada por dicha estructura es 32 * 32 * 4 bytes = 4KB. Fue fundamental que recordásemos la necesidad de cerrar los descriptores de archivo que el usuario pudiese dejar abiertos al llamar a `loader_exit` pues de lo contrario, ciertos dispositivos como el puerto serie (de apertura exclusiva) jamás hubiesen podido ser reabiertos por ningún otro proceso.

Es importante mencionar la implementación de una función `copy2user` (acompanada de otras, por ejemplo, para brindar el nivel de privilegio de cierta dirección virtual) utilizada en todos los char devices. Aún así, sabemos que dicha función no es perfecta y no está libre de bugs de seguridad, pero es un primer intento (aprovechamos por ejemplo el hecho de que el kernel se encuentra en direcciones inferiores a cualquiera de usuario para evitar validar todas las direcciones en cierto rango).

`con` - Driver de consola

El driver de consola administra un anillo de multiples **consolas**. Cada una de estas representa un *char device* con algunos atributos extras que permiten

almacenar el estado de la consola en un momento dado, una cola para la lectura del teclado y punteros a otros dispositivos de consola, definiendo el orden de las consolas en el anillo. Se pueden tener abiertas 8 consolas como máximo (los *devices* se almacenan en un arreglo fijo en el módulo).

El **estado** de una consola comprende un buffer de 4000 bytes[1] y una estructura `vga_screen_state_t` que almacena la posición del cursor (fila y columna) y los atributos de color. Además del estado, cada consola tiene puntero a la *siguiente* y *anterior* consola en el anillo.

En la inicialización, el driver inicializa el arreglo de consolas y registra un *handler* de interrupción del teclado. Este handler lee 1 byte (el scancode) del puerto de control del teclado, y actualiza unas variables globales que contienen el último scancode leído, el último caracter leído y si las teclas de control `ALT`, `SHIFT` o `CTRL` están siendo presionadas. Una vez que esta información esta disponible, chequea si la combinación `ALT`+`SHIFT`+`Izquierda` o `ALT`+`SHIFT`+`Derecha` se presionó, y hace un *cambio de consola* a la anterior o a la siguiente, según corresponda. Si la consola que tiene foco está esperando al teclado, se la despierta para que pueda procesar el nuevo scancode o caracter.

En un principio habíamos definido una única cola de lectura de teclado global, para todas las consolas. Pero esto no iba a ser posible por que se podía dar la situación en la que varias tareas tienen una consola diferente abierta, leyendo del teclado, y sólo habría que despertar a la tarea que está esperando sobre la consola que tiene foco, en cuanto la interrupción llegue.

Sólo una cola de lectura es suficiente para una consola, pues, si bien una tarea puede tener varios file descriptors de consolas distintas, sólo puede bloquearse a leer de una sola de ellas al mismo tiempo.

Como las tareas no tienen un mecanismo para compartir file descriptors, sólo una tarea puede estar bloqueada en la cola de una consola a la vez, así que el estado de esta consola siempre se alternaría entre `FREE_QUEUE` y el pid de una tarea existente.

Cuando una tarea abre una nueva consola, se crea e inicializa un nuevo dispositivo, se la agrega al anillo justo después de la consola que tiene foco en ese momento, y finalmente se cambia el foco a la nueva. Cuando se cierra la consola, luego de eliminar el buffer dinamico y corregir los punteros de la lista enlazada, se le da foco a la anterior consola. Cuando no hay más consolas, se limpia la pantalla y se ejecuta una tarea de usuario llamada `screen_saver`[2].

En el momento del cambio de foco, se copia el contenido del buffer VGA en 0xB8000 al buffer de la consola, guardando el estado, y se restaura el contenido del buffer de la nueva consola al buffer VGA. Cuando se escribe al *file descriptor* de una consola, si ésta tiene foco, se escribe directamente al buffer VGA en 0xB8000, y sino al buffer de la consola. De esta manera, sólo escribimos una

[1] 80 filas x 25 columnas x 2 bytes por caracter = 4000 bytes

[2] Creemos que cada tanto es necesario ventilar nuestra creatividad de alguna manera :-)

sola vez, sea en el buffer de video real, o el de la consola, y no ambos al mismo tiempo.

`serial` - Driver del puerto serie

El controlador del puerto serie está conformado por los siguientes componentes:

- Un handler de interrupción que lee de la placa UART el byte recibido y lo escribe en un buffer local.
- Un buffer circular en el cual se almacenan los bytes recibidos, esperando a ser leidos por el usuario.
- Un puntero que indica la primera posición válida del buffer.
- Un puntero que indica la última posición válida del buffer.

El controlador configura al dispositivo para trabajar por interrupciones y declara los handlers para atenderlas. Es la manera que consideramos más eficiente, ya que trabajar con polling contra un dispositivo tan lento sería contraproducente. Al recibir la interrupción, la atiende rápidamente tomando el byte leido y agregándolo en el buffer. Cuando el usuario llama a la función `read` con el parámetro `size`, el driver copia al usuario una cantidad de bytes equivalente al menor valor entre `size` y la cantidad de bytes almacenados en el buffer. Luego de la copia, ajusta los punteros de modo tal que si queda información en el buffer, la lectura seguirá leyendo desde donde quedó y la escritura se efectuará a continuación del último byte válido.

El descriptor de dispositivo cuenta con una cola de espera de lectura y otra de escritura. El estado de estas colas es siempre vacío o bien con un proceso encolado. Esto se debe a que tomamos la decisión de que la apertura del puerto serie fuese exclusiva: dos procesos no pueden tener al mismo tiempo un descriptor a éste. Dicho esto, nos queda la duda si quizás una cola hubiese bastado, pues al ser bloqueantes las lecturas y escrituras, una para cada operación se torna innecesario.

Configuramos los parametros de conexión (baudios, paridad, tamaño del caracter, etc.) al momento de la inicialización de los dispositivos y permanecen configurados de esta forma a lo largo de la ejecución. Estos parametros son: 8 bits de tamaño del caracter, sin paridad, un bit de parada y 9600 baudios. Cabe mencionar que la dirección del puerto de E/S es almacenada en el descriptor de dispositivo en un intento de proveer mayor flexibilidad al controlador.

Durante el desarrollo afrontamos un problema que permanece sin solución: no logramos recibir una interrupción desde los puertos 3 y 4. Consultando con los docentes concluimos que es probable que bochs no tuviese implementada dicha funcionalidad.

Block Devices

`hdd` - Hard Disk Driver

Nos limitamos a implementar sólo la funcionalidad de lectura del disco rígido, para el canal primario del mismo (primary master, primary slave). Cualquier lectura fuera del tamaño de bloque del dispositivo da como resultado un error.

El controlador de dispositivo efectúa PIO (entrada/salida programada) utilizando LBA de 28 bits y con lectura sencilla (no múltiple, es decir, un comando de lectura efectúa la lectura de un sólo sector de disco). Durante el desarrollo nos encontramos con ciertas dudas acerca de los retardos necesarios antes del envío de cada comando al dispositivo (teóricamente de 400ms), pues si bien en Bochs el controlador funciona, parece no funcionar en otros entornos como VirtualBox con un chipset y controladores idénticos.

El descriptor de dispositivo alberga el puerto de E/S primario y el puerto auxiliar de control, así como el canal (maestro, esclavo) y una cola de espera por lecturas. En dicha cola podría haber cero o más procesos. Cabe destacar que no es necesario un puntero para la cantidad de bytes copiados así como tampoco se utiliza la función `copy2user` pues se asume que dicho controlador es utilizado por el kernel y no directamente por el usuario.

File System

`ext2` - Driver de *ext2*

Decidimos implementar una versión limitada del sistema de archivos *Second Extended Filesystem (ext2)* que únicamente lee directorios y archivos, ignorando los permisos.

En el driver del block device decidimos almacenar el superbloque y la tabla de descriptores de grupos de bloques, ya que para abrir cualquier archivo es necesario acceder a esa información y por el espacio que ocupa es ridículo pedírselo al disco cada vez.

Vale la pena aclarar que la inicialización del filesystem es `lazy`, es decir, realiza una inicialización incompleta y la completa cuando recibe la primera solicitud de apertura de archivo. Esto se debe a que la función de lectura del driver de disco es bloqueante y pretende encolar a la tarea que lo llama en caso de no tener la información disponible de inmediato. En el momento que llamamos a la función de inicialización del filesystem, aún no hay tareas cargadas en el sistema y de no hacer la inicialización `lazy` el driver de disco intentaría bloquear a la IDLE_TASK, resultando en un error.

Los pasos internos a seguir para abrir un archivo fueron los siguientes:

- Se recibe la ruta absoluta del archivo en el filesystem.

- Se parsea la ruta de a una barra (`/`) por vez, empezando por root.
- Cada paso recorrido resulta en un nombre de archivo que se busca en el último directorio abierto y en caso de estar presente se continúa con el proceso.
- Si se termina la cadena y se confirma la existencia del archivo buscado, se obtiene el inodo de dicho archivo y se lo almacena junto con otra información en un nuevo objeto del tipo `ext2_file` que es una extensión del tipo `chardev`.

El objeto `ext2_file` tiene una particularidad. Implementa un buffer del tamaño de una página de memoria que se va llenando desde el dispositivo de bloque a medida que se va leyendo el archivo. Pero distingue entre tipos de archivo `regular` y `directorio`. Cuando se abre para lectura un archivo del tipo `directorio`, dentro del proceso de creación del `ext2_file`, se lee el contenido del directorio y se lo escribe en el buffer. Asumimos para esto que nunca el contenido de un directorio superará el tamaño de página.

También implementa la función `seek` que en caso de posicionar el puntero en una sección no cargada dentro del buffer, se encarga de hacer el prefetch de información desde el block device. Es decir, si bien la función seek no lee del `chardev`, es posible que dispare una lectura al dispositivo de bloque.

Tareas

Syscall `run`

Durante la implementación de dicha llamada al sistema nos topamos con una limitación: `loader_load` esperaba el puntero a un archivo PSO que eventualmente podría tener un tamaño mayor al de una página. Esto se traduce en la necesidad de direcciones virtuales contiguas y debimos implementar funciones en el módulo de manejo de memoria para tal fin. Lo que finalmente hicimos fue buscar una entrada vacía en el directorio de páginas del proceso actual, lo cual nos brindaba un espacio de direccionamiento virtual contiguo de hasta 4MB. De aquí se deduce la limitación actual para utilizar dicha llamada al sistema: el ejecutable más grande no puede exceder los 4MB. Una vez ejecutada la carga del archivo se liberan los recursos previamente reservados.

Tarea `init`

Anteriormente, todas las tareas de usuario estaban integradas en el propio binario del kernel. Ahora que éste tiene accesos a dispositivos como el disco rígido, las tareas están siendo leídas de éste, y la única tarea integrada al kernel es `init`.

Su función es cargar y ejecutar las mismas tareas de *testing* que se cargaban en nuestro trabajo anterior, y finalmente ejecutar `console`. Si la carga o la ejecución de alguna de estas tareas falla, se alerta al usuario y se detiene la ejecución del sistema.

Tarea `console`

Para poder demostrar el uso del filesystem virtual y el acceso a los dispositivos que el kernel ahora soporta, implementamos un shell con funcionalidad limitada, *tab completion* y *reverse search*[3]. La tarea abre una consola e imprime por pantalla un *prompt* para que el usuario pueda ingresar comandos con el teclado.

Los comandos que implementamos dentro del shell son los siguientes:

- `:PATH` lee y ejecuta un archivo .PSO ubicado en `PATH`, e imprime en pantalla su PID.
- `@PATH` abre el archivo o dispositivo en `PATH`, y lee e imprime en pantalla (similar a `cat` en UNIX).
- `exit` cierra la consola y termina la tarea.

[3] (mentira)