Sistemas Operativos

Práctica 6: Entrada/Salida

Notas preliminares

- Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

Parte 1 – Métodos de acceso

Ejercicio 1 ★

Esperar la finalización de una operación de E/S utilizando *polling* desperdicia muchos ciclos de reloj (*busy waiting*). Sin embargo, si el dispositivo está listo para la operación, esta técnica puede ser mucho más eficiente que una estrategia basada en interrupciones.

Describir una estrategia híbrida, que combine *polling* e interrupciones para acceder a dispositivos de E/S. Mostrar tres escenarios, uno donde *polling* sea el más conveniente, otro donde interrupciones sea el método de acceso más favorable y un tercero donde convenga utilizar la estrategia híbrida.

Ejercicio 2 ★

Un sistema informático que incorpora DMA permite una implementación eficiente de la multiprogramación. Suponga que un proceso en promedio usa sólo el 23 % de su tiempo la CPU y el resto está en entrada y salida (E/S) y suponiendo que toda operación de E/S se realiza por DMA. Sabiendo que se puede demostrar que $U_{\text{CPU}}(n) = 1 - U_{\text{E/S}}(1)^n$ y que $U_{\text{DMA}}(n) = 1 - U_{\text{CPU}}(1)^n$, donde n es la cantidad de procesos, estime qué utilización del procesador y del canal de DMA se logra, si se tiene un grado de multiprogramación de 6 procesos.

Ejercicio 3 ★

a) ¿Se puede implementar *spooling* sin dispositivos virtuales?
b) ¿Qué métricas mejora o empeora el uso de *spooling*: latencia, *throughput*, tiempo de ejecución, liberación de recursos?
c) ¿Por qué no se suele hacer spooling de placas de red y sí de impresoras?

Ejercicio 4

Se tiene un sistema de impresión que utiliza spooling sobre una impresora virtual y DMA sobre la impresora real. Responder y justificar:

a) ¿De cuál de estas técnicas es importante que esté al tanto el usuario final?
b) ¿El *driver* de impresión tiene alguna noción de *spooling*?

Parte 2 – Interfaz de E/S

Para todos los ejercicios de esta sección que requieran escribir código deberá utilizarse la API descripta en la parte final de esta práctica.

Ejercicio 5 ★

¿Cuáles de las siguientes opciones describen el concepto de *driver*? Seleccione las correctas y **justifique**.

a) Es una pieza de *software*.
b) Es una pieza de *hardware*.
c) Es parte del SO.
d) Dado que el usuario puede cambiarlo, es una aplicación de usuario.
e) Es un gestor de interrupciones.
f) Tiene conocimiento del dispositivo que controla pero no del SO en el que corre.
g) Tiene conocimiento del SO en el que corre y del tipo de dispositivo que controla, pero no de las particularidades del modelo específico.

Ejercicio 6

Un cronómetro posee 2 registros de E/S:

- `CHRONO_CURRENT_TIME` que permite leer el tiempo medido,
- `CHRONO_CTRL` que permite ordenar al dispositivo que reinicie el contador.

El cronómetro reinicia su contador escribiendo la constante `CHRONO_RESET` en el registro de control.

Escribir un *driver* para manejar este cronómetro. Este *driver* debe devolver el tiempo actual cuando invoca la operación `read()`. Si el usuario invoca la operación `write()`, el cronómetro debe reiniciarse.

Ejercicio 7

Una tecla posee un único registro de E/S : `BTN_STATUS`. Solo el *bit* menos significativo y el segundo *bit* menos significativo son de interés:

- $\texttt{BTN\_STATUS}_0$: vale 0 si la tecla no fue pulsada, 1 si fue pulsada.
- $\texttt{BTN\_STATUS}_1$: escribir 0 en este *bit* para limpiar la memoria de la tecla.

Escribir un *driver* para manejar este dispositivo de E/S. El *driver* debe retornar la constante `BTN_PRESSED` cuando se presiona la tecla. Usar *busy waiting*.

Ejercicio 8 ★

Reescribir el *driver* del ejercicio anterior para que utilice interrupciones en lugar de *busy waiting*. Para ello, aprovechar que la tecla ha sido conectada a la línea de interrupción número 7.

Para indicar al dispositivo que debe efectuar una nueva interrupción al detectar una nueva pulsación de la tecla, debe guardar la constante `BTN_INT` en el registro de la tecla.

Ayuda: usar semáforos.

Ejercicio 9

Indicar las acciones que debe tomar el administrador de E/S:

a) cuando se efectúa un *open*.
b) cuando se efectúa un *write*.

Ejercicio 10

¿Cuál debería ser el nivel de acceso para las *syscalls* `IN` y `OUT`? ¿Por qué?

Ejercicio 11 ★

Se desea implementar el *driver* de una controladora de una vieja unidad de discos ópticos que requiere controlar manualmente el motor de la misma. Esta controladora posee 3 registros de lectura y 3 de escritura. Los registros de escritura son:

- `DOR_IO`: enciende (escribiendo 1) o apaga (escribiendo 0) el motor de la unidad.
- `ARM`: número de pista a seleccionar.
- `SEEK_SECTOR`: número de sector a seleccionar dentro de la pista.

Los registros de lectura son:

- `DOR_STATUS`: contiene el valor 0 si el motor está apagado (o en proceso de apagarse), 1 si está encendido. Un valor 1 en este registro no garantiza que la velocidad rotacional del motor sea la suficiente como para realizar exitosamente una operación en el disco.
- `ARM_STATUS`: contiene el valor 0 si el brazo se está moviendo, 1 si se ubica en la pista indicada en el registro `ARM`.
- `DATA_READY`: contiene el valor 1 cuando el dato ya fue enviado.

Además, se cuenta con las siguientes funciones auxiliares (ya implementadas):

- `int cantidad_sectores_por_pista()`: Devuelve la cantidad de sectores por cada pista del disco. El sector 0 es el primer sector de la pista.
- `void escribir_datos(void *src)`: Escribe los datos apuntados por `src` en el último sector seleccionado.
- `void sleep(int ms)`: Espera durante `ms` milisegundos.

Antes de escribir un sector, el *driver* debe asegurarse que el motor se encuentre encendido. Si no lo está, debe encenderlo, y para garantizar que la velocidad rotacional sea suficiente, debe esperar al menos 50 ms antes de realizar cualquier operación. A su vez, para conservar energía, una vez que finalice una operación en el disco, el motor debe ser apagado. El proceso de apagado demora como máximo 200 ms, tiempo antes del cual no es posible comenzar nuevas operaciones.

a) Implementar la función `write(int sector, void *data)` del *driver*, que escriba los datos apuntados por `data` en el sector en formato LBA indicado por `sector`. Para esta primera implementación, no usar interrupciones.

b) Modificar la función del inciso anterior utilizando interrupciones. La controladora del disco realiza una interrupción en el `IRQ 6` cada vez que los registros `ARM_STATUS` o `DATA_READY` toman el valor 1. Además, el sistema ofrece un *timer* que realiza una interrupción en el IRQ 7 una vez cada 50 ms. Para este inciso, no se puede utilizar la función `sleep`.

Parte 3 – API para escritura de drivers

Un SO provee la siguiente API para operar con un dispositivo de E/S. Todas las operaciones retornan la constante `IO_OK` si fueron exitosas o la constante `IO_ERROR` si ocurrió algún error.

`int open(int device_id)`	Abre el dispositivo.
`int close(int device_id)`	Cierra el dispositivo.
`int read(int device_id, int *data)`	Lee el dispositivo `device_id`.
`int write(int device_id, int *data)`	Escribe el valor en el dispositivo `device_id`.

Para ser cargado como un *driver* válido por el sistema operativo, el *driver* debe implementar los siguientes procedimientos:

Función	Invocación
`int driver_init()`	Durante la carga del SO.
`int driver_open()`	Al solicitarse un *open*.
`int driver_close()`	Al solicitarse un *close*.
`int driver_read(int *data)`	Al solicitarse un *read*.
`int driver_write(int *data)`	Al solicitarse un *write*.
`int driver_remove()`	Durante la descarga del SO.

Para la programación de un *driver*, se dispone de las siguientes *syscalls*:

`void OUT(int IO_address, int data)`	Escribe `data` en el registro de E/S.
`int IN(int IO_address)`	Devuelve el valor almacenado en el registro de E/S.
`int request_irq(int irq, void *handler)`	Permite asociar el procedimiento `handler` a la interrupción `IRQ`. Devuelve `IRQ_ERROR` si ya está asociada a otro *handler*.
`int free_irq(int irq)`	Libera la interrupción `IRQ` del procedimiento asociado.