Sistemas Operativos

Práctica 5: 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 – 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 1 ★

¿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 2

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 3

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 4 ★

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 5

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

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

Ejercicio 6

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

Ejercicio 7 ★

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 (es decir, que los sectores son numerados según un índice, empezando por cero) 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`.

Ejercicio 8 ★

Se desea escribir un *driver* para la famosa impresora *Headaches Persistent*. El manual del controlador nos dice que para comenzar una impresión, se debe:

- Ingresar en el registro de 32 bits `LOC_TEXT_POINTER` la dirección de memoria dónde empieza el buffer que contiene el *string* a imprimir.
- Ingresar en el registro de 32 bits `LOC_TEXT_SIZE` la cantidad de caracteres que se deben leer del buffer.
- Colocar la constante `START` en el registro `LOC_CTRL`.

En este momento, si la impresora detecta que no hay suficiente tinta para comenzar, escribirá *rápidamente* el valor `LOW_INK` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`. Caso contrario, la impresora comenzará la impresión, escribiendo el valor `PRINTING` en el registro `LOC_CTRL` y el valor `BUSY` en el registro `LOC_STATUS`. Al terminar, la impresora escribirá el valor `FINISHED` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`.

Un problema a tener en cuenta es que, por la mala calidad del *hardware*, éstas impresoras suelen detectar erróneamente bajo nivel de tinta. Sin embargo, el fabricante nos asegura en el manual que "alcanza con probar hasta 5 veces para saber con certeza si hay o no nivel bajo de tinta".

El controlador soporta además el uso de las interrupciones: `HP_LOW_INK_INT`, que se lanza cuando la impresora detecta que hay nivel bajo de tinta, y `HP_FINISHED_INT`, que se lanza al terminar una impresión.

Se pide implementar las funciones `int driver_init()`, `int driver_remove()` y `int driver_write()` del *driver*. Piense cuidadosamente si conviene utilizar *polling*, *interrupciones* o una mezcla de ambos. Justifique la elección. Además, debe asegurese de que el código no cause condiciones de carrera. Las impresiones deberán ser bloqueantes. No hace falta que implemente *spooling*.

Ejercicio 9 ★

Se requiere diseñar un sistema de seguridad compuesto por una cámara, un sensor de movimiento, y un software de control. El requisito principal es que cuando el sensor detecta movimiento, el sistema responda con el encendido de la cámara por una cantidad de tiempo T, si detecta movimiento antes de que termine el tiempo T, la política a seguir es la de esperar un tiempo T desde esta última detección.

1. Proponga un diseño para este sistema de seguridad, donde debe indicar cuántos y qué tipo de registros tendría cada dispositivo, e indicando también para qué se utilizarían. También debe indicar y justificar el tipo de interacción: interrupciones, *polling, DMA*, etc. con cada dispositivo. Puede usar uno o más tipos de interacción diferentes para cada dispositivo.

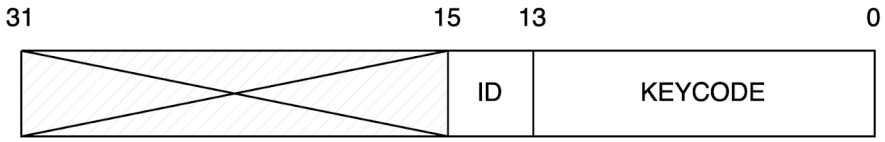
2. Una vez que tenga el diseño, debe escribir los dos *drivers* correspondientes (las operaciones que considere necesarias: `open()`, `write()`, `read()`, etc, para poder cumplir el objetivo planteado. Tenga en cuenta que el software de control correrá a nivel de usuario, y podrá tener interacción con los drivers. No es necesario que escriba las especificaciones del software, pero sí se debe indicar cómo interactuará con los *drivers*. Cada operación usada debe estar justificada.

3. Explique cómo se genera la interacción a nivel del código entre los *drivers* que propuso.

Ejercicio 10

Se pide programar el driver para un teclado gamer. Este teclado es muy novedoso, ya que es mucho más grande y tiene muchas más teclas que los teclados normales, y puede ser utilizado al mismo tiempo en hasta tres juegos distintos. Esto permite que hasta tres personas a la vez puedan jugar con el mismo teclado, pero cada quién con su propia aplicación, y sin interferirse entre ellos. Además, el teclado cuenta con tres visores inteligentes que permiten mostrar información sobre el juego que se esté jugando (por ejemplo, la vida del jugador, o mensajes importantes). También permite activar, desactivar y remapear teclas, y configurar macros, para cada usuario por separado. Además de todas estas impresionantes características, tiene luces de todos los colores posibles (también configurables).

El funcionamiento del controlador del dispositivo es el siguiente: cada vez que el usuario presiona una tecla el teclado levanta la interrupción `IRQ_KEYB` y coloca en el registro `KEYB_REG_DATA` un entero de 32 bits que contiene en sus 14 bits menos significativos el código `KEYCODE` correspondiente a la tecla presionada, y en los 2 bits siguientes un identificador `P`, que vale `0` cuando la tecla debe ser recibida por todas las aplicaciones conectadas, y en caso contrario un número entre `1` y `3` que indica a qué aplicación está destinada esa tecla. Considerar que, al ser una interrupción, esta debe ser atendida en un tiempo acotado, es decir, no debe bloquearse ni quedarse esperando. Se le deberá informar al dispositivo que la tecla pudo ser almacenada escribiendo correctamente escribiendo el valor `READ_OK` en el registro `KEYB_REG_CONTROL`. En caso contrario se le deberá escribir el valor `READ_FAILED`.



Por otro lado, el dispositivo cuenta con tres direcciones de memoria (una por cada aplicación posible) `INPUT_MEM_0`, `INPUT_MEM_1`, `INPUT_MEM_2`, siendo cada una un arreglo de 100 bytes desde los que el dispositivo leerá el input de las aplicaciones. Estas direcciones serán mapeadas a memoria por el driver durante su carga en el Kernel, y se mapearán en el arreglo `input_mem`.

Cuando una aplicación se conecte al teclado, se le deberá informar al mismo escribiéndole el valor `APP_UP` en el registro `KEYB_REG_STATUS`, y el id de la aplicación correspondiente en el registro `KEYB_REG_AUX`. Cuando una aplicación se desconecte se deberá hacer lo mismo pero escribiendo el valor `APP_DOWN`.

El driver deberá ir almacenando los caracteres ASCII correspondientes a las distintas pulsaciones del teclado en un buffer, a la espera de ser leídas por las aplicaciones que se encuentren conectadas. Para ello, se cuenta con una función `keycode2ascii(int keycode)` que traduce los códigos de pulsación en su correspondiente caracter ascii.

Los procesos de usuario que deseen leer el input del teclado deberán hacerlo mediante una operación de `read()` bloqueante. Esta operación sólo puede retornar cuando haya leído la cantidad de bytes solicitados. En caso de haber más de un proceso conectado al dispositivo, cada proceso deberá poder consumir cada caracter leído de forma independiente. Ejemplo: suponer que hay tres procesos conectados, y desde el dispositivo se presionaron las teclas correspondientes a ”sistemas”, en todos los casos con el identificador `P=0`. En tal caso si p0 hace un read de 2 bytes, obtendrá ”si”, si p1 luego hace un read de 5 bytes, obtendrá ”siste”, si luego p0 hace un read de 1 byte obtendrá ”s”, y si luego p2 hace un read de 8 bytes obtendrá ”sistemas”.

Además, los procesos conectados al teclado podrán utilizar la función `write()` para informarle al teclado el estado del jugador, comandar los colores del teclado, reconfigurar teclas, y varias otras funciones que provee el dispositivo.

Se cuenta con el siguiente código:

```
char input_mem[3][100];
char buffer_lectura[3][1000];
```

```
atomic_int buffer_start[3];
atomic_int buffer_end[3];
boolean procesos_activos[3];

void driver_load() {
    // Se corre al cargar el driver al kernel.
}

void driver_unload() {
    // Se corre al eliminar el driver del kernel.
}

int driver_open() {
    // Debe conectar un proceso, asignandole un ID y retornandolo,
    // o retornando -1 en caso de falla.
}

void driver_close(int id) {
    // Debe desconectar un proceso dado por parametro.
}

int driver_read(int id, char* buffer, int length) {
    // Debe leer los bytes solicitados por el proceso ''id''
}

int driver_write(char* input, int size, int proceso) {
    copy_from_user(input_mem[proceso], input, size);
    return size;
}
```

a) Implementar las funciones `driver_load`, `driver_unload`, `driver_open`, `driver_close` y `driver_read`. Implementar cualquier función o estructura adicional que considere necesaria (tener en cuenta que en el kernel no existe la *libc*).

Para resolver la función `driver_read` es posible implementar una cola circular del modo descripto a continuación. Las variables `buffer_start[i]` y `buffer_end[i]` indican el inicio y el final del *buffer*. Al hacer una lectura, la variable `start` crece. Al hacer una escritura, la variable `end` crece. Si la variable `end` llega al final del *buffer*, debe comenzar por el principio, siempre teniendo cuidado de no pisarse con `start`. Si vale que `end+1` es igual a `start`, entonces se considera que el *buffer* está lleno. Para facilitar esta implementación, se cuenta con las funciones, que son todas atómicas:

- `int get_buffer_length(int i)`: dado un número de *buffer* devuelve la cantidad de caracteres ocupados en el mismo.
- `boolean write_to_buffer(int i, char src)`: intenta escribir un caracter en el *buffer* correspondiente, y retorna un booleano indicando si la escritura se pudo realizar.
- `boolean write_to_all_buffers(char src)`: intenta escribir un caracter en todos los *buffers*, y retorna un booleano indicando si la escritura se pudo realizar.
- `void copy_from_buffer(int i, char* dst, int size)`: lee la cantidad de caracteres indicada desde el *buffer* correspondiente, y los copia en un segundo *buffer* pasado como parámetro. Tener en cuenta que esta función no realiza ningún tipo de chequeo sobre los *buffers*, sino que simplemente copia.

Parte 2 – 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* y funciones:

`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.

```
unsigned long copy_from_user(char *to, char *from, uint size)
unsigned long copy_to_user(char *to, char *from, uint size)
void *kmalloc(uint size)
void kfree(void *buf)
void sema_init(semaphore *sem, int value)
void sema_wait(semaphore *sem)
void sema_signal(semaphore *sem)
void mem_map(void *source, void *dest, int size)
void mem_unmap(void *source)
```