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[1] indicado por `sector`. Para esta primera implementación, no usar interrupciones.

[1] El direccionamiento de bloque lógicos (LBA) es un método comunmente utilizado para especificar la localización de los bloques de datos en sistemas de almacenamiento.

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(void* data)` 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.

Ejercicio 9 ★

Explique qué función hace un driver que usa el código que se muestra a continuación. Justifique su respuesta explicando el funcionamiento de las partes principales del código.

```
#define MODULE
#define __KERNEL__
struct file_operations memoria_fops = {
        read: memoria_read,
        write: memoria_write,
        open: memoria_open,
        release: memoria_release };
int memoria_major = 60;
char *memoria_buffer;
int init_module(void) {
        int result;
        result = register_chrdev(memoria_major, "memoria",&memoria_fops);
        if (result < 0) {
                printk("<1>memoria: no puedo obtener numero mayor %d\n",memoria_major);
                return result; }
        memoria_buffer = kmalloc(1, GFP_KERNEL);
        if (!memoria_buffer) {
                result = -ENOMEM;
                goto fallo; }
```

```
        memset(memoria_buffer, 0, 1);
        printk("<1>Insertando modulo\n");
        return 0;
        fallo:
                cleanup_module();
                return result;      }
void cleanup_module(void) {
        unregister_chrdev(memoria_major, "memoria");
        if (memoria_buffer) { kfree(memoria_buffer); }
        printk("<1>Quitando modulo\n");  }
int memoria_open(struct inode *inode, struct file *filp) {
        MOD_INC_USE_COUNT;
        return 0; }
int memoria_release(struct inode *inode, struct file *filp) {
        MOD_DEC_USE_COUNT;
        return 0; }
ssize_t memoria_read(struct file *filp, char *buf,size_t count, loff_t *offset) {
        copy_to_user(buf,memoria_buffer,1);
        if (*offset == 0) {
                *offset+=1;
                return 1; }
        else { return 0; } }
ssize_t memoria_write( struct file *filp, char *buf,size_t count, loff_t *offset) {
        char *tmp;
        tmp=buf+count-1;
        copy_from_user(memoria_buffer,tmp,1);
        return 1; }
```

Ayudas:

- La función `memset` de C `*memset(void *str, int c, size_t n)` copia el caracter `c` (un char sin signo) a los primeros `n` caracteres de `str`.
- `MOD_INC_USE_COUNT` / `MOD_DEC_USE_COUNT`: Macros usadas para incrementar o decrementar el número de usuarios que están usando el driver en un determinado momento. El SO se encarga de gestionarlas.

Ejercicio 10 ★

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 11 ★

Se requiere diseñar un driver para instalar en un robot transportador de paquetes para un servicio de correos. El mismo está compuesto por: una batería, una pequeña computadora que corre un sistema operativo Linux, un lector de código de barras, un brazo mecánico y un sistema de movimiento.

El modo de funcionamiento es el siguiente: el robot corre un programa de usuario que ejecuta la función `siguiente_paquete()` para obtener un dato del tipo `struct paquete(int x, int y, int codigo)` que representa la localización `(x,y)` de alguna de las estanterías del depósito, y el código de barras del paquete a transportar. Luego, deberá indicarle al sistema de movimiento que se desplace hasta dicha ubicación. El desplazamiento puede demorar una cantidad de minutos imposible de saber previamente, por lo que el sistema de desplazamiento deberá informar de algún modo cuando haya llegado a la ubicación correcta.

Cada posible ubicación representa una estantería que contiene 10 paquetes. El robot deberá buscar el paquete correcto mediante el siguiente procedimiento. Deberá tomar cada paquete uno por uno, utilizando para ello su brazo mecánico, y leer su código de barras. Si el código no coincide, se deberá volver a dejar el paquete en la estantería, y leer el siguiente paquete. Si el código coincide, se deberá llevar el paquete hacia la posición `(0,0)` del depósito, en donde se encuentra una cinta de entrega de paquetes. Finalmente, se deberá depositar el paquete en dicha cinta.

Cuando se encuentre frente a una estantería, el brazo mecánico puede extenderse y moverse de forma horizontal y vertical a cada una las 10 posiciones correspondientes de una estantería. Cuando se encuentre frente a la cinta, puede extenderse sobre la cinta. También puede contraerse para acercarse al lector de códigos de barras. En su punta cuenta con una mano que puede tomar y soltar cosas. Asumir que los movimientos del brazo son tan veloces que se pueden considerar instantáneos. El brazo deberá estar contraído para poder ser leído por el lector de códigos de barras.

Tener en cuenta en el diseño que la duración de la batería en el robot se ve seriamente afectada cuando el procesador es usado intensivamente. Asumir que los paquetes se reponen de forma automática en las estanterías (siempre hay 10 paquetes disponibles). Asumir que no existe una estantería en la posición `(0,0)`. Asumir que siempre se va a poder encontrar el paquete buscado.

1. Proponga un diseño, en 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 con cada dispositivo (*interrupciones, polling, DMA, etc.*). Puede usar uno o más tipos de interacción diferentes para cada dispositivo. Se debe considerar el uso de la batería para justificar las decisiones.

2. Una vez que tenga el diseño, debe escribir los tres *drivers* correspondientes a lector, brazo y sistema de movimiento respectivamente. Escribir con código C todas las operaciones que considere necesarias para poder cumplir el objetivo planteado: `init()`, `load()`, `open()`, `write()`, `read()`, etc. Tenga en cuenta que el software de control correrá a nivel de usuario, y podrá tener interacción con los drivers. Indicar con código C cómo interactuará el software de control 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 12 ★

La Fórmula 1 es un deporte motor donde los pilotos corren en monopolazas diseñadas con tecnología sumamente avanzada.

Cada piloto corre para un equipo, el cual durante las distintas etapas que componen a un *Grand Prix* (prácticas, clasificación y carrera) se encuentra en constante contacto con el conductor para recibir feedback y así poder mejorar la puesta a punto del monoplaza y solucionar eventuales problemas.

El monoplaza cuenta con un sistema muy complejo que incluye sensores para medir distintas variables relevantes, como la presión de neumáticos, el consumo de combustible y distintas temperaturas.

En particular, es muy común que el piloto pueda consultar durante la carrera las temperaturas de los frenos desde el comando del volante y en función de eso decidir cambiar el balance de frenado. Para esto existen 5 sensores `TEMP_SENSOR_0...TEMP_SENSOR_4` que registran la temperatura en distintos puntos del sistema de frenado.

Luego estos valores se promedian y ese resultado es lo que se observa en el display del volante. Lo más importante en este caso es que la información esté disponible bajo garantías de tiempo estrictas aunque implique una pérdida de precisión. Esto es así ya que el piloto hace esta consulta, por ejemplo, a 300 km/h para maximizar luego el trazado de una curva. Por lo tanto, las lecturas del piloto a través del driver del sistema de sensado deben intentar obtener los valores de cada sensor y retornar, en un tiempo cercano a los 50 ms, el promedio de las lecturas válidas (este tiempo podría ser mayor si ocurre una interrupción durante la lectura y si el sistema está en modo `ACCURACY_HIGH` como se detalla a conitnuación). Para el diseño de este driver se debe asumir que cada consulta a un registro de sensado está en el orden de 1 ms y que si el registro no está listo el valor obtenido será negativo.

Un proceso puede interactuar con este sistema utilizando mayor precisión, seteando el registro `ACCURACY` en `HIGH`. De esta forma, el equipo puede conectarse al sistema del monoplaza y realizar un seguimiento del estado desde los boxes. Es normal que los tiempos de decisión del equipo sean más holgados pero en general se requiere la mayor precisión posible, ya que desde allí suelen tomarse decisiones para cambiar el mapa del motor del auto que puede afectar severamente la performance durante la carrera. Por lo tanto es viable que las lecturas sean bloqueantes y que retornen cuando el promedio se calcula con todos los valores.

Cuando un proceso se registra en el modo de precisión alta, su comportamiento debería ser similar al caso normal (es decir, debe intentar verificar si puede obtener rápidamente el valor de cada temperatura) pero ahora puede aprovechar el hecho de que cada sensor levanta una interrupción `IRQ_PRECISION_0...IRQ_PRECISION_1` cuando registra una nueva lectura válida.

Se pide escribir parte del código del driver del sistema de sensores de la computadora del monoplaza. En particular, se deben implementar las siguientes funciones:

1. `int driver read(int id, int* buffer, int size)`: devuelve el promedio de los sensores según el modo asociado al proceso registrado con id.
2. `int driver open()`: devuelve un id al proceso en cuestión. Por defecto todos los procesos se registran con el modo de accuracy `LOW`.

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*:

`void OUT(int IO_address, int data)` `int IN(int IO_address)`	Escribe `data` en el registro de E/S. Devuelve el valor almacenado en el registro de E/S.

`int request_irq(int irq, void *handler)`
 	
`int free_irq(int irq)`
	Permite asociar el procedimiento `handler` a la interrupción `IRQ`. Devuelve `IRQ_ERROR` si ya está asociada a otro *handler*.
	Libera la interrupción `IRQ` del procedimiento asociado.