Ejercicio 2 - Sistemas de entrada/salida

Manuel Panichelli LU 072/18

Enunciado

Una tarjeta de red inalámbrica posee cinco registros de E/S: WL_STATUS, WL_BUFFER_ENTRADA, WL_BUFFER_SALIDA, WL_OK, WL_INT_ENABLED.

- WL_STATUS: (RO) Vale 1 si la tarjeta está ocupada, 0 si no lo está.
- WL_BUFFER_ENTRADA: Almacena el paquete a enviar.
- WL_BUFFER_SALIDA: Almacene el paquete a recibir.
- WL_OK: (RO) Vale 1 si el envío o recepción se logró con éxito, 0 de lo - contrario.
- WL_INT_ENABLED: Si se escribe 1 en este registro, el dispositivo generará una interrupción en la línea 42 al terminar el envío o recepción de un paquete. Si vale 0, el dispositivo no generará interrupciones.

Este dispositivo suele responder casi inmediatamente a las solicitudes, pero esporádicamente, en ciertas ocasiones, tarda en atender algunas solicitudes.

Escribir un driver para manejar este dispositivo de E/S, donde se implemente una estrategia híbrida (interrupciones y busy waiting) buscando optimizar el rendimiento del dispositivo. Tener en cuenta los aspectos de sincronización.

Resolucion

Preludio

Primero veamos una descripcion verbal del driver y de las cosas que asumimos.

- La responsabilidad del driver sera simplemente enviar y recibir paquetes, no sabe de quien son ni para donde van. Todo eso es delegado al usuario.
- Voy a suponer que es un char device y que un paquete puede ser encodeado en el int que se le escribe a driver write. (ya que *lo vemos en redes™*)
- Para los reads, voy a asumir que en el buffer de salida se pone un puntero a un dato en donde luego la controladora de la placa escribe el paquete que recibio. Supongo que la placa de red tiene internamente una cola con los paquetes recibidos, los cuales va desencolando con cada read.
- Y ademas, que como driver le digo que hacer a la placa poniendo cosas en el buffer de entrada y de salida, ya que no hay un registro de "operacion a realizar" o algo asi.
- Garantiza la exclusion mutua en el uso de la placa de red, no se deberian poder hacer reads o writes concurrentes, ya que por lo que parece en la interfaz del driver, solamente permite ver el estado de una operacion. Entonces se hara de una operacion a la vez. Para esto uso un mutex de kernel.

Por ultimo, sobre lo hibrido, ya que por lo general es rapido y a veces tarda mucho, haremos lo siguiente

- Comenzamos por suponer que va a responder inmediatamente, entonces despues de enviada la peticion, hacemos busy waiting por un tiempo predeterminado (una cantidad de intentos predeterminadas)

  Esto es un parametro que deberia ser ajustado dependiendo de cual sea el tiempo de respuesta cuando es casi inmediato.
- Si despues de eso no esta listo, entonces estamos en el caso de que tarda en atender. Ahi habilitamos interrupciones y esperamos a que termine (haciendo wait en un semaforo), luego de la cual lo deshabilitamos. Esto se hace para simplificar, pero como en realidad tarda en atender varias solicitudes, habria dejar interrupciones habilitadas y deshabilitarlas luego de varias solicitudes.

Implementacion

```
// Suponiendo que se cuenta con la interfaz de drivers de la guia
// Open y close no se implementan a proposito porque no hay nada que hacer.

#define INT_ENABLED   1
#define INT_DISABLED  0
#define IRQ_NET_CARD 42
```

```
#define STATUS_OCCUPIED 1
#define STATUS_FREE 0

#define RESULT_OK 0

int driver_init() {
    // Comenzamos sin interrupciones
    OUT(WL_INT_ENABLED, INT_DISABLED)

    // Asociamos el handler de interrupciones a la linea 42,
    // para que este listo para cuando lo necesitemos
    if request_irq(IRQ_NET_CARD, &handler) != 0 { return -1 }

    // Sync para interrupciones
    waiting = false
    card_ready = semaphore_create(0)

    // Mutex para garantizar EXCL
    mutex = mutex_create()
}

int driver_remove() {
    // Liberamos lo necesario
    if free_irq(IRQ_NET_CARD) != 0 { return -1 }
}

void handler() {
    // La int se genera cuando se termina el envio o recepcion de un paquete.
    // Si este proceso no estaba esperando, no hago signal porque podria
    // dejar inconsistente a la instancia del driver.
    if (waiting) {
        waiting = false
        card_ready.signal()
    }
}

int driver_write(int* data) {
    // Leemos los datos que nos dio el usuario de forma segura, ya que podria
    // ni siquiera tener acceso a esa porcion de memoria.
    int pkg
    if copy_from_user(&pkg, data, sizeof(int)) != 0 {
        // probablemente un error de privilegios
        return -1
    }

    // Garantizamos que solo un proceso puede hacer uso de la placa
    // de red a la vez.
    mutex.lock()

        // Guardamos el paquete en el buffer de la placa de red para que esta
        // lo envie.
        OUT(WL_BUFFER_ENTRADA, pkg)

        // Esperamos a que termine de procesar nuestra solicitud
        wait_done()

        // El resultado ya esta listo, lo leemos
        res = IN(WL_OK)

    mutex.unlock()

    // Si bien en la especificicacion dice que devuelve 1 cuando no salio
    // ok, voy a asumir que 0 es ok y todo el resto no.
    if res != RESULT_OK { return -1}

    // Todo OK
    return 0
}

int driver_read(int* data) {
    mutex.lock()

        // Decimos a la placa donde recibir el paquete
        int pkg
        OUT(WL_BUFFER_SALIDA, &pkg)
```

```
        // Esperamos a que termine con nuestra solicitud
        wait_done()

        // El resultado ya esta listo, lo leemos
        res = IN(WL_OK)

    mutex.unlock()

    // Vemos si algo fallo
    if res != RESULT_OK { return -1 }

    // Finalmente le copiamos al usuario el resultado
    if copy_to_user(data, pkg, sizeof(int)) != 0 { return -1 }

    return 0
}


// Espera a que la placa termine con una solicitud.
// Comienza haciendo busy waiting, y si tarda mucho,
// termina usando interrupciones (las cuales deshabilita
// al recibir la respuesta.)
void wait_done() {
    // Es la cantidad de intentos de busy waiting que hace el
    // driver antes de rendirse y pasar a interrupciones. Deberia ser
    // mayor a lo que tarda cuando responde "casi inmediatamente".
    // Le pongo 20 para que tenga algo pero no se cuanto deberia valer
    int MAX_ATTEMPTS = 20

    // Hacemos un numero finito de intentos de busy waiting, despues
    // del cual nos rendimos y hacemos fallback a interrupciones.
    for (i = 0; i < MAX_ATTEMPTS; i++) {
        if IN(WL_STATUS) == STATUS_FREE {
            // Respondio rapido, cortamos
            return
        }
    }

    // Fallback a interrupciones
    OUT(WL_INT_ENABLED, INT_ENABLED)
    waiting = true
    card_ready.wait()

    // Deshabilitamos interrupciones
    // Aca hay una oportunidad de optimizacion: esperar un par de
    // requests para deshabilitarla, ya que segun el enunciado
    // era lento por un par de requests. Pero bueno, asi es mas
    // simple. Espero que no sea problema :)
    OUT(WL_INT_ENABLED, INT_DISABLED)
}
```