Periféricos Interfaces y Buses

I. Arquitectura de E/S

II. Programación de E/S

Aspectos básicos de la programación de E/S. Arquitectura y programación de la E/S en el sistema operativo. Manejadores de dispositivos (drivers) y su programación (interrupciones).

III. Interfaces de E/S de datos

IV. Dispositivos de E/S de datos

V. Buses

VI. Controladores e interfaces de dispositivos de almacenamiento

VII. Sistemas de almacenamiento

II. Programación de E/S

Bloque II

- Operaciones de control en el driver
- Sincronización
- Temporización
- Acceso al hardware
- Interrupciones
- Threads del kernel

Operaciones de control en el driver

Las operaciones de control sobre el dispositivo se hacen a través del punto de entrada `ioctl`:

- establecer ciertos parámetros del dispositivo
- obtener información sobre el estado del mismo

La orden de control al dispositivo consta de dos parámetros:

- `cmd` es la orden
- `arg` es un argumento para ejecutar la orden

En Linux los comandos se pueden definir de acuerdo a unos criterios que codifican los *números mágicos*:

- cuatro grupos de bits: tipo, número, dirección y tamaño
- una serie de macros para construirlos y consultarlos

Operaciones de control en el driver (cont.)

En cualquier caso los comandos y los argumentos que éstos necesitan son un compromiso entre el driver y el programa de aplicación

La implementación del `ioctl` es normalmente una instrucción `switch` basada en el número del comando

El valor de retorno:

- si el número de comando no es válido algunas funciones devuelven -EINVAL (argumento inválido)
- POSIX establece que se devuelva -ENOTTY (comando inapropiado para dispositivo)

Acceso al argumento del `ioctl`

Como para las operaciones de lectura y escritura es necesario intercambiar infomación con la memoria de usuario

- se pueden utilizar `copy_from_user` y `copy_to_user`
- para intercambio de datos pequeños (uno, dos, cuatro y ocho bytes) se pueden utilizar otras macros definidas en `<asm/uaccess.h>`:

```
put_user(dato, puntero);

get_user(local, puntero);
```

  - escribe el *dato* al *puntero* del espacio de usuario y viceversa
  - el tamaño del dato depende del tipo del puntero
  - devuelve un cero si va bien o -EFAULT si falla; se llama internamente a `access_ok` que chequea si el puntero es válido

Ejemplo: iocontrol

Definición de las constantes de control

```
//  ioconst.h

// comandos de ioctl

#define CAMBIAR_MODO  10
#define LEER_MODO    100

// modos de operación para el argumento

#define IO_MODO1 1
#define IO_MODO2 2
```

Ejemplo: iocontrol (cont.)

```
//  iocontrol.h

int iocontrol_open(struct inode *inodep, struct file *filp);

int iocontrol_release(struct inode *inodep, struct file *filp);

ssize_t iocontrol_read (struct file *filp, char *buff,
                            size_t count, loff_t *offp);

ssize_t iocontrol_write (struct file *filp, const char *buff,
          size_t count, loff_t *offp);

int iocontrol_ioctl(struct inode *inodep, struct file *filp,
       unsigned int cmd, unsigned long arg);
```


Iocontrol: includes y variables globales

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include "iocontrol.h"
#include "ioconst.h"

MODULE_LICENSE("GPL");

struct iodatos {
  struct cdev *cdev; // Character device structure
  dev_t dev;         // informacion con el numero mayor y menor
  int modo;          // registro del modo de operación
};

static struct iodatos datos;
```

Iocontrol: inicialización de puntos de entrada y modo

```
static struct file_operations iocontrol_fops = {
  THIS_MODULE, // owner
  ...,         // todo nulos
};

static int modulo_instalacion(void) {

  ...

  // ponemos los puntos de entrada y el modo
  iocontrol_fops.open=iocontrol_open;
  iocontrol_fops.release=iocontrol_release;
  iocontrol_fops.write=iocontrol_write;
  iocontrol_fops.read=iocontrol_read;
  iocontrol_fops.ioctl=iocontrol_ioctl;
  datos.modo = IO_MODO1;

  ...
}
```


Iocontrol: operaciones de lectura

Solamente informan del estado que implementan

```
ssize_t iocontrol_read (struct file *filp, char *buff,
          size_t count, loff_t *offp)
{
  printk(KERN_INFO "iocontrol> (read) estoy en MODO 1\n");
  return 0;
}

ssize_t iocontrol_read2 (struct file *filp, char *buff,
          size_t count, loff_t *offp)
{
  printk(KERN_INFO "iocontrol> (read) estoy en MODO 2\n");
  return 0;
}
```

Iocontrol: operaciones de escritura

Solamente informan del estado que implementan

```
ssize_t iocontrol_write (struct file *filp, const char *buff,
          size_t count, loff_t *offp)
{
  printk(KERN_INFO "iocontrol> (write) estoy en MODO 1\n");
  return count;
}

ssize_t iocontrol_write2 (struct file *filp, const char *buff,
          size_t count, loff_t *offp)
{
  printk(KERN_INFO "iocontrol> (write) estoy en MODO 2\n");
  return count;
}
```


Iocontrol: ioctl

```
int iocontrol_ioctl (struct inode *inodep, struct file *filp,
                     unsigned int cmd, unsigned long arg)
{
  int modo;

  switch (cmd) {
    case CAMBIAR_MODO:
      // capturamos el modo que llega en arg
      if (get_user(modo,(int *)arg)) {
          printk(KERN_ERR "iocontrol> (ioctl) puntero erroneo\n");
          return -EFAULT;
      }
      printk(KERN_INFO "iocontrol> (ioctl) el modo es =%d\n",modo);
```

Iocontrol: ioctl (cont.)

```
      // cambiamos de modo
      switch (modo) {
        case IO_MODO1:
          filp->f_op->read=iocontrol_read;
          filp->f_op->write=iocontrol_write;
          datos.modo=modo;
          break;
        case IO_MODO2:
          filp->f_op->read=iocontrol_read2;
          filp->f_op->write=iocontrol_write2;
          datos.modo=modo;
          break;

        default:
        printk(KERN_ERR "iocontrol> (ioctl) modo %d no valido\n",modo);
          return -EINVAL;
      }
      break; // fin de CAMBIAR_MODO
```


Iocontrol: ioctl (cont.)

```
    case LEER_MODO:
      if (put_user(datos.modo,(int *)arg)) {
          printk(KERN_ERR "iocontrol> (ioctl) puntero erroneo\n");
          return -EFAULT;
      }
      break;

    default:
     printk(KERN_ERR "iocontrol> (ioctl) comando %d no valido\n",cmd);
      return -EINVAL;
  }
  printk(KERN_INFO "iocontrol> (ioctl) nuevo modo =%d\n",datos.modo);
  return 0;
}
```

Prueba de iocontrol

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include "ioconst.h"

int main() {

  int fd;
  int modo;
  char *mistring="Hola yo soy un mensaje";
  char strleido[100];
  ssize_t leidos, escritos;
```


Prueba de iocontrol (cont.)

```
  //Primer uso dejado con el modo cambiado
  fd=open("/dev/ioctl0", O_RDWR);
  if (fd==-1) { perror("Error al abrir el fichero");
    exit(1);}
  printf("Fichero abierto\n");
  escritos=write(fd,mistring,strlen(mistring)+1);
  printf("Escrito en modo 1\n");

  //cambio a modo 2
  modo=IO_MODO2;
  ioctl(fd,CAMBIAR_MODO,&modo);
  leidos=read(fd,&strleido,100);
  printf("Leido en modo 2\n");

  //cambio de nuevo a modo 1
  modo=IO_MODO1;
  ioctl(fd,CAMBIAR_MODO,&modo);
  leidos=read(fd,&strleido,100);
  printf("Leido en modo 1\n");
```

Prueba de iocontrol (cont.)

```
  // cambio final a modo 2
  modo=IO_MODO2;
  ioctl(fd,CAMBIAR_MODO,&modo);
  escritos=write(fd,mistring,strlen(mistring)+1);
  printf("Escrito en modo 2\n");

  if (close(fd)==-1) {perror("Error al cerrar el fichero");
    exit(1);}
  printf("Fichero cerrado\n");

  //Segundo uso
  fd=open("/dev/ioctl0", O_RDWR);
  if (fd==-1) {perror("Error al abrir el fichero");
    exit(1);}
  printf("Fichero abierto\n");
```


Prueba de iocontrol (cont.)

```
  modo=1234; //para probar que se lee bien
  ioctl(fd,LEER_MODO,&modo);
  printf("Operando en modo %d \n", modo);

  escritos=write(fd,mistring,strlen(mistring)+1);
  printf("Escrito\n");

  if (close(fd)==-1) {
    perror("Error al cerrar el fichero");
    exit(1);
  }
  printf("Fichero cerrado\n");

  exit(0);
}
```

II. Programación de E/S

Bloque II

- Operaciones de control en el driver
- Sincronización
- Temporización
- Acceso al hardware
- Interrupciones
- Threads del kernel

Sincronización

La sincronización es uno de los aspectos más importantes y más complicados en la implementación de drivers

Implica a las actividades concurrentes

- threads y procesos de usuario
- threads del núcleo
- rutinas de servicio de interrupción

Todos ellos compiten por el uso de la CPU y deben sincronizarse para compartir e intercambiar datos

Hay que tener en cuenta que los puntos de entrada se pueden ejecutar concurrentemente y además el kernel es expulsable

Sincronización (cont.)

Evitar el fenómeno conocido como *race conditions*:

```
1if (device_free == TRUE) {
2   device_free = FALSE;
3   ...        // usa el dispositivo
4   device_free = TRUE;
5}
```

- Para un sistema con dos tareas que usan el dispositivo, se puede producir una situación en la que ambas pueden llegar a utilizarlo simultáneamente
- La tarea 1 ejecuta la línea 1 y encuentra el dispositivo libre, y en ese momento es expulsada por la tarea 2, que también lo encuentra libre y lo utiliza
- También puede ocurrir en la asignación de memoria que chequea si el puntero es nulo

Sincronización (cont.)

Mecanismos:

- exclusión mutua (acceso a datos o recursos compartidos)
  - semáforos
  - mutexes (macros que usan semáforos para acceso mutuamente exclusivo)
  - semáforos de múltiples lectores y un escritor (*reader-writer locks*)
  - esperas activas (*spinlocks*)
  - esperas activas de múltiples lectores
- variables atómicas y estructuras de datos que no necesitan sincronización (colas circulares)

Sincronización (cont.)

- espera
  - semáforos
  - "*completions*" (más eficientes)
  - colas de eventos
- temporización

Semáforos

Los semáforos se usan como mecanismos de exclusión mutua y/o espera entre procesos.

El kernel de Linux tiene una implementación de semáforo contador, en el que se tiene una *cola* de procesos en espera, y un *valor* que es un entero positivo o cero

Funciona con dos operaciones básicas:

- "P" o esperar:
  - si valor>0 entonces valor=valor-1
  - si valor==0 entonces suspender el proceso (metiéndolo en la cola)
- "V" o señalizar:
  - si hay un proceso esperando, activarlo
  - si no, valor=valor+1

Semáforos (cont.)

Si el semáforo se utiliza para sincronización de eventos se inicializa con el valor 0.

Si se utiliza para exclusión mutua se inicializa con el valor 1.

Con cualquier otro valor positivo nos puede servir para controlar una lista de recursos disponibles:

- el número de recursos es el valor inicial
- según se van consumiendo el semáforo se va decrementando
- cuando se acaban los recursos el proceso se encola
- posiblemente haya que utilizarlo con un mutex

Semáforos (cont.)

Se definen en `<asm/semaphore.h>` como `struct semaphore`, y sobre él se pueden ejecutar las siguientes operaciones:

- Inicializar el semáforo (*val*=1 para exclusión mutua)

```
void sema_init (struct semaphore *sem, int val);
```

- Tomar el semáforo, esperando si no está libre

```
void down (struct semaphore *sem);
int down_interruptible (struct semaphore *sem);
int down_trylock (struct semaphore *sem);
```

  - la versión interrumpible por el usuario (por una señal) es más aconsejable; devuelve cero si no se interrumpe
  - se suele devolver `-ERESTARTSYS` si se ha interrumpido
  - la versión no bloqueante devuelve un valor distinto de cero si no ha tomado el semáforo (siempre hay que comprobar el valor de retorno)

Semáforos (cont)

- Liberar el semáforo (despertando a otro thread que esté esperando)

```
void up(struct semaphore *sem);
```

Ojo!

- Estos semáforos tienen inversión de prioridad

Mutexes

Existen funciones y macros que ayudan al uso de los semáforos del kernel como mutexes:

- Declaración e inicialización de un mutex

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

  - el resultado es la variable semáforo `name`
  - la versión que lo devuelve tomado debe liberarlo posteriormente
- Inicialización de mutex en tiempo de ejecución

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

Semáforos de múltiples lectores

En un sistema multiprocesador se puede mejorar la eficiencia si múltiples lectores acceden concurrentemente al recurso

- exclusión mutua entre los lectores y un escritor
- exclusión mutua entre escritores
- prioridad para los escritores

Los semáforos de múltiples lectores están en `<linux/rwsem.h>`

El tipo de datos es `struct rw_semaphore`

Semáforos de múltiples lectores (cont.)

Las principales funciones son:

- inicializar

```
void init_rwsem (struct rw_semaphore *sem);
```

- lectores

```
void down_read (struct rw_semaphore *sem);
int down_read_trylock (struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

- escritores

```
void down_write (struct rw_semaphore *sem);
int down_write_trylock (struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
```

Esperas activas: spinlocks

Hay operaciones que no se pueden suspender

- p.e., rutinas de servicio de interrupción

En multiprocesadores puede ser más eficiente no dormirse

El *spinlock* sólo puede tener dos valores *bloqueado* o *no bloqueado*:

- se implementa como el estado de un bit concreto de un entero
- si se consigue bloquear la función que lo usa retorna inmediatamente
- si no se consigue tomar la función no se suspende: entra en un lazo que continuamente chequea el spinlock hasta que esté disponible

Esperas activas: spinlocks (cont.)

La espera activa tiene peligros

- sólo vale para sincronizarse con threads o rutinas de interrupción en distintos procesadores
- habitualmente es preciso enmascarar interrupciones en el procesador local
- si es en el mismo procesador, puede haber un *deadlock*

Además, en monoprocesadores no tiene ventaja

Esperas activas: API

La API está en `<linux/spinlock.h>`; el tipo es `spinlock_t`

Inicializar

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
void spin_lock_init(spinlock_t *lock);
```

Tomar

```
void spin_lock(spinlock_t *lock);
```

Tomar inhibiendo interrupciones en el procesador local y almacenando en `flags` el estado de interrupción

```
void spin_lock_irqsave(spinlock_t *lock,
                       unsigned long flags);
```


Esperas activas: API (cont.)

Tomar inhibiendo interrupciones (hay que estar seguro de que nadie más las ha inhibido)

```
void spin_lock_irq(spinlock_t *lock);
```

Tomar inhibiendo interrupciones SW (pero no las HW)

```
void spin_lock_bh(spinlock_t *lock);
```

Liberar (usar la correspondiente a la función de tomar)

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock,
                            unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

Esperas activas: API (cont.)

UNIVERSIDAD DE CANTABRIA

También hay una API de esperas activas de múltiples lectores, con objetos del tipo `rwlock_t`

- Inicializar

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED;
void rw_lock_init(rwlock_t *lock);
```


Esperas activas: API (cont.)

UNIVERSIDAD DE CANTABRIA

- Leer

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock,
                       unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock,
                            unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

Esperas activas: API (cont.)

- Escribir

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock,
                        unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock,
                             unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```


Precauciones con la exclusión mutua

Si se intenta tomar un recurso dos veces desde el mismo flujo de código (p.e. se llama a una función que también lo toma), puede haber un *deadlock*

Es preciso diseñar y documentar muy bien la sincronización

- p.e., se puede tomar un semáforo en cada punto de entrada, y así ya despreocuparse en el resto del código
  - el bloqueo puede ser grande
  - cuidado con la suspensión

También puede haber un *deadlock* si se toman múltiples recursos:

- una solución puede ser tomarlos siempre en el mismo orden

Evitando las secciones críticas

Uso de *buffers* circulares

- se pueden programar para que no necesiten exclusión mutua
- hay una implementación disponible en el kernel `<linux/kfifo.h>`

Uso de variables atómicas, en `<asm/atomic.h>`, con el tipo `atomic_t` (entero de 24 bits o más)

Las operaciones suelen ser rápidas (una instrucción ensamblador)

Variables atómicas: API

Dar valor

```
void atomic_set(atomic_t *v, int i);
atomic_t v = ATOMIC_INIT(0);
```

Leer

```
int atomic_read(atomic_t *v);
```

Sumar y restar

```
void atomic_add(int i, atomic_t *v); //v=v+i
void atomic_sub(int i, atomic_t *v); //v=v-i
```

Incrementar y decrementar

```
void atomic_inc(atomic_t *v); //v=v+1
void atomic_dec(atomic_t *v); //v=v-1
```

Variables atómicas: API (cont.)

Incrementar, decrementar, restar y luego comparar con cero (*true*= resultado igual a cero)

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

Sumar y luego comprobar si el resultado es negativo, en cuyo caso devuelve *true*

```
int atomic_add_negative(int i, atomic_t *v);
```


Variables atómicas: API (cont.)

Sumar, restar, incrementar, decrementar pero devolviendo el nuevo valor de la variable atómica

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

Hay que recordar que el carácter atómico es para una sola variable

- si hay que sincronizar varios valores, hay que usar un semáforo o similar

Sincronización de espera

El mecanismo más habitual es la "cola de eventos", usando el tipo `wait_queue_head_t` definido en `<linux/wait.h>`

También se pueden usar los semáforos:

- aunque son mecanismos optimizados para exclusión mutua, es decir, para encontrarlos normalmente libres
- en sincronización de espera lo normal es esperar

Un mecanismo más simple y ligero es el de "*completions*"

- Mecanismo para que un thread espere a otro indicando que el trabajo se ha completado

"Completions"

Están definidos en `<linux/completion.h>`, con el tipo `struct completion`

Inicialización:

```
void init_completion (struct completion *c);
```

Espera, señalización a un thread, y señalización a todos los threads que esperan

```
void wait_for_completion (struct completion *c);
void complete (struct completion *c);
void complete_all (struct completion *c);
```

- si se señaliza a todos hay que volver a inicializar la estructura, si no se puede volver a usar directamente
- se recuerdan los eventos

Colas de eventos: API

Inicialización:

```
void init_waitqueue_head(wait_queue_head_t *q);
```

Macros de espera a que se cumpla una condición:

```
void wait_event(wait_queue_head_t q,
                int condition);
void wait_event_interruptible(wait_queue_head_t q,
                int condition);
```

- `condition` se evalúa como un booleano antes de la espera
- al despertarse vuelve a evaluarse;
- si no se cumple se repite la espera
- la preferida es la versión interrumpible (por una señal) que devuelve un valor distinto de cero si se ha interrumpido

Colas de eventos: API (cont.)

Espera con tiempo límite (*timeout*); el tiempo se expresa como un intervalo en "*jiffies*" o "*ticks*"

- el número de *jiffies* por segundo está en `<linux/param.h>`, en la constante `HZ`

```
void wait_event_timeout(wait_queue_head_t q,
                int condition, int timeout);
void wait_event_interruptible_timeout
               (wait_queue_head_t q,
                int condition, int timeout);
```

Colas de eventos: API (cont.)

Señalización a todos los que esperan

```
void wake_up(wait_queue_head_t *q);
```

Señalización a los que esperan de modo interrumpible

```
void wake_up_interruptible(wait_queue_head_t *q);
```


Consejos para la sincronización de espera

Nunca hacerla dentro de una sección crítica

- p.e., con un *spinlock* tomado

No hacer suposiciones sobre el estado del sistema al despertarse

- volver a comprobar la condición de espera

Diseñar bien el sistema de señalización

- para cada espera debe haber una señalización

II. Programación de E/S

Bloque II

- Operaciones de control en el driver
- Sincronización
- Temporización
- Acceso al hardware
- Interrupciones
- Threads del kernel

Temporización

En ocasiones una función del kernel debe dormirse durante un tiempo

- en espera de una acción del dispositivo o del usuario
- con una función de "dormirse" o "retrasarse"

En ocasiones una función del kernel debe invocarse cada cierto tiempo, o al cabo de un tiempo de forma asíncrona al proceso

- con temporizadores del núcleo

También es necesario medir el tiempo transcurrido

- contando "***jiffies***"
- o mirando la hora en un reloj

Medir el tiempo

Hay un contador de *jiffies* que se puede consultar

- es la variable jiffies, declarada en `<linux/jiffies.h>` como

```
volatile unsigned long jiffies;
```

Para comparar o hacer aritmética con jiffies es necesario prever que el contador puede "dar la vuelta",

- para ello existen estas macros que retornan un booleano

```
int time_after(unsigned long a, unsigned long b); //a>b
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```


Medir el tiempo (cont)

Para obtener la diferencia entre dos medidas, se puede usar el "truco" de pasar a valor con signo:

```
diff = (long)t2 - (long)t1;
```

Y para obtener, por ejemplo los milisegundos de diferencia:

```
msec = (diff*1000)/HZ;
```

Hay operaciones para conversión de *jiffies* a otros formatos de tiempo, por ejemplo el `struct timespec` de POSIX (en `<linux/time.h>`)

```
unsigned long timespec_to_jiffies
      (struct timespec *value)
void jiffies_to_timespec(unsigned long jiffies,
                         struct timespec *value);
```

Registros específicos de procesador

Para más precisión pueden usarse otros relojes

En los procesadores Pentium existe en "*time-stamp counter*" (TSC) que mide ciclos de reloj

Para usarlo, en `<asm/msr.h>` existen estas macros para enteros de 32 bits sin signo, y de 64 bits, resp.:

```
rdtsc(low32,high32)
rdtscl(low32)
rdtscll(var64)
```


Leyendo la hora y fecha

Existe en `<linux/time.h>` una llamada para saber la hora, con granularidad de *jiffie*

```
struct timespec current_kernel_time(void);
```

Dormir el thread durante un tiempo

Para dormir un tiempo alto, podemos usar un timeout expresado en *jiffies*

La API está en `<linux/sched.h>`

```
signed long schedule_timeout (signed long timeout);
```

- retorna el tiempo que le queda por dormir, si se interrumpió por una señal
- es necesario previamente poner el procesador en estado interrumpible; ejemplo:

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout(delay);
```


Dormir el thread durante un tiempo (cont.)

También disponemos de funciones expresadas en milisegundos, en `<linux/delay.h>`

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible
        (unsigned int millisecs);
```

- La segunda retorna el número de milisegundos que nos faltan por dormir (cero si no se interrumpe)

Retrasos cortos

Para retrasos cortos, de menos de un milisegundo no se pueden usar *jiffies*

En `<linux/delay.h>` podemos encontrar las esperas activas:

```
void ndelay(unsigned long nsecs); //nanosegundos
void udelay(unsigned long usecs); //microsegundos
void mdelay(unsigned long msecs); //milisegundos
```


Temporizadores del kernel

Los temporizadores se usan para planificar la ejecución de una función en un instante de tiempo dado

- de forma asíncrona a los threads de usuario
- tienen la resolución de un *jiffie*
- la función que se ejecuta es en respuesta a una interrupción software
  - hay que prever la sincronización (p.e., con *spinlocks* o variables atómicas)
  - no se puede dormir o llamar al planificador (`schedule()`)
  - no puede usar funciones bloqueantes (p.e., `kmalloc()`)
- puede volver a registrarse para ejecutar más tarde (p.e., actividad periódica)

Temporizadores del kernel (cont.)

La estructura de datos de un timer se encuentra en <linux/timer.h>:

```
struct timer_list {
   /*...*/                  // otros miembros internos
   unsigned long expires; //valor absoluto de jiffies
   void (*function)(unsigned long);
                            // función manejadora
   unsigned long data;    // argumento para el manejador
              // para más datos hacer cast de un puntero
};
```


Temporizadores del kernel (cont.)

Inicializar el temporizador

```
void init_timer(struct timer_list *timer);
```

Añadir a la lista de temporizadores activos

```
void add_timer(struct timer_list *timer);
```

Cambia el instante de expiración

```
int mod_timer(struct timer_list *timer,
              unsigned long expires);
```

Temporizadores del kernel (cont.)

Elimina el temporizador antes de que el tiempo expire

```
int del_timer(struct timer_list *timer);
```

Elimina el temporizador, garantizando que al finalizar la función no se está ejecutando

```
int del_timer_sync(struct timer_list *timer);
```

- debe asegurarse que la función del timer no usa *add_timer* sobre sí misma al eliminarlo
- es la función preferida para eliminar

Ejemplo de temporizador

Mostraremos sólo lo que es nuevo respecto a los ejemplos anteriores

```
// Datos que se le pasan a la función manejadora, incluyendo el
// propio temporizador

struct datos_temporizador {
  unsigned long jiffies_previos;
  void * buffer;
  struct timer_list timer;
  atomic_t apagando;
};

static struct datos_temporizador datos_timer;
```

Ejemplo de temporizador (cont.)

En la función de instalación del driver

```
// inicializamos los datos
datos_timer.buffer = kmalloc(MAX, GFP_KERNEL);
datos_timer.jiffies_previos=jiffies;
atomic_set(&datos_timer.apagando,0);

// creamos el temporizador
init_timer(&datos_timer.timer);
datos_timer.timer.expires=jiffies+1000; // 1000 jiffies
datos_timer.timer.function=manejador;
datos_timer.timer.data=(unsigned long) &datos_timer; // cast
add_timer(&datos_timer.timer);
```

En la función de desinstalación del driver:

```
atomic_set(&datos_timer.apagando,1);
del_timer_sync(&datos_timer.timer);
```


Ejemplo de temporizador (cont.)

```
// Manejador

void manejador (unsigned long arg) {
  struct datos_temporizador *dat =
     (struct datos_temporizador *) arg;
  unsigned long j=jiffies;

  sprintf(dat->buffer,
          "Numero de jiffies actual:%li diferencia:%li\n",
          j,(long)j-(long)dat->jiffies_previos);
  dat->timer.expires+=1000;
  dat->jiffies_previos=j;
  if(atomic_read(&dat->apagando)==0) {
    add_timer(&dat->timer); //solo si no estamos apagando
  }
}
```

Ejemplo de temporizador (cont.)

```
ssize_t temporiza_read (struct file *filp, char *buff,
                        size_t count, loff_t *offp)
{
  ssize_t cuenta;
  unsigned long not_copied;

  printk(KERN_INFO "temp> (read) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }
  not_copied=copy_to_user(buff,datos_timer.buffer,
                          (unsigned long)cuenta);
  if (not_copied!=0) {
    printk(KERN_WARNING "temp> (read) AVISO, no se leen datos\n");
    return(-EFAULT);
  }
  return cuenta;
}
```


II. Programación de E/S

Bloque II

- Operaciones de control en el driver
- Sincronización
- Temporización
- Acceso al hardware
- Interrupciones
- Threads del kernel

Acceso al hardware

La principal utilidad de un driver es servir de abstracción de un dispositivo hardware

El acceso al hardware es muy dependiente de la arquitectura

- no centramos aquí en la arquitectura PC
- entrada/salida por puertos e instrucciones específicos
  - conceptualmente no hay diferencia con la entrada/salida mapeada en memoria
  - en la práctica hay que tener en cuenta la optimización del compilador

Reserva de puertos de entrada/salida

Debemos asegurarnos el acceso exclusivo a un puerto

Disponemos de funciones en `<linux/ioport.h>`

```
struct resource *request_region
      (unsigned long first, unsigned long n,
       const char *name);
```

- intenta reservar `n` puertos, empezando en `first`
- retorna nulo si falla, y otro valor si no
- las reservas se ven en `/proc/ioports`

Para liberar la reserva

```
void release_region(unsigned long first,
                    unsigned long n);
```

Reserva de puertos de entrada/salida (cont.)

Existe otra función que está en desuso:

```
int check_region(unsigned long first,
                 unsigned long n);
```

- devuelve un valor negativo si la región está ocupada
- si la región está libre no garantiza que la reserva vaya a tener éxito
- al usar `request_region` se debe comprobar si realmente ha ido bien

Acceso a puertos de entrada/salida

Se dispone en `<asm/io.h>` de funciones para acceso a los puertos de entrada/salida:

```
unsigned inb (unsigned port);
void outb (unsigned char byte, unsigned port);
```

Estas funciones leen o escriben, respectivamente, un byte (8 bits) en el puerto indicado

Hay funciones también para leer enteros de 16 o 32 bits.

```
unsigned inw(unsigned port);
void outw(unsigned short word,unsigned port);
unsigned inl(unsigned port);
void outl(unsigned longword,unsigned port);
```

Acceso al hardware (cont.)

Hay funciones equivalentes que hacen una pausa después de leer o escribir:

```
// para arquitecturas x386
unsigned inb_p (unsigned port);
void outb_p (unsigned char byte, unsigned port);
```

Estas funciones son de tipo inline y se pueden usar también en el espacio de usuario con las siguientes características:

- El driver se debe compilar con la opción `-O` para expandir las funciones.
- Se deben usar las llamadas al sistema `ioperm` (para puertos individuales) e `iopl` (para un espacio de I/O completo) para obtener los permisos de operación sobre los puertos.

II. Programación de E/S

Bloque II

- Operaciones de control en el driver
- Sincronización
- Temporización
- Acceso al hardware
- Interrupciones
- Threads del kernel

Interrupciones y manejadores

Las interrupciones son condiciones de excepción de dispositivos externos que el procesador interpreta como un evento

Algunos de los casos para los que se utilizan podrían ser:

- Indicación de final de alguna operación (finalización de transferencia en una operación DMA)
- Aviso de disponibilidad de datos (caracter disponible para ser leído del puerto)
- Aviso de que un dispositivo está listo para ejecutar algún comando (una impresora genera una interrupción cuando ha imprimido un caracter y está lista para imprimir el siguiente)

Interrupciones y manejadores (cont.)

Los manejadores de interrupción se especifican en la instalación del módulo o en la apertura del dispositivo y se eliminan en la desinstalación del módulo o el cierre del dispositivo.

Estos manejadores de interrupción ejecutan antes de que se termine cualquier otro procesado del kernel o de una aplicación

- Es importante minimizar la longitud de los manejadores de interrupción para conseguir mejores respuestas temporales
- Hay que tener en cuenta que durante la ejecución de un manejador de interrupción se pueden inhibir todas las interrupciones

Instalación de interrupciones

Las líneas de interrupción son un recurso limitado y normalmente escaso

El kernel guarda un registro de estas líneas al igual que de los puertos de I/O

- en `/proc/interrupts`
- se lleva un contador de las interrupciones ocurridas desde el arranque del sistema

Los módulos pueden reservar y liberar estos canales de interrupción a través funciones declaradas en `<linux/interrupt.h>`

Instalación de interrupciones (cont.)

Reservar una interrupción e instalar un manejador

```
int request_irq
      (unsigned int irq,
       irqreturn_t (*handler)
           (int, void *, struct pt_regs *),
       unsigned long flags, const char *dev_name,
       void *dev_id);
```

- el valor retornado es cero si va bien, o un código de error
- `irq` es el número de interrupción
- `handler` es el manejador, que veremos luego
- `dev_name` es un nombre que identifica al dueño de la IRQ

Instalación de interrupciones (cont.)

- `flags` es una combinación de opciones
  - `SA_INTERRUPT` para interrupciones rápidas (que inhiben las demás interrupciones) o 0 para las normales (llamadas tradicionalmente lentas)
  - `SA_SHIRQ`: indica que la interrupción puede ser compartida entre dispositivos
- `dev_id` es un puntero que se utiliza para compartir líneas de interrupción
  - es un identificador único que se utiliza en la liberación de la interrupción
  - lo puede usar además el driver apuntando a su área de datos privada para identificar el dispositivo
  - si no se comparte la IRQ se puede poner a NULL

Instalación de interrupciones (cont.)

Liberar una reserva

```
void free_irq (unsigned int irq, void *dev_id);
```

Instalación de interrupciones (cont.)

Ejemplo de instalación de una interrupción de puerto paralelo (IRQ 7):

```
// handler normal, función manejadora es pport_isr
result =request_irq(7,pport_isr,0,"pport",NULL);
if (result!=0){
   printk(KERN_INFO "can't get irq 7\n");
} else {
   // habilitar la interrupción del puerto paralelo
   // short_base es la dirección base del dispositivo
   outb(0x10,short_base+2);
}
```


Instalación de interrupciones (cont.)

El manejador de interrupciones se puede instalar:

- en la inicialización del driver
  - en este caso se bloquea una línea de interrupción que podría no utilizarse nunca
- cuando se abre por primera vez
  - es un modo de compartir líneas de interrupción

La desinstalación del manejador se puede hacer:

- en la desintalación del driver
- después de que se cierre el dispositivo por última vez
  - requiere llevar una cuenta de aperturas por dispositivo

Instalación de interrupciones (cont.)

Hay funciones para averiguar las líneas de interrupción libres mediante autodetección y pruebas (en `<linux/interrupt.h>`)

Otro fichero relacionado con las interrupciones es `/proc/stat`

- contiene una línea que comienza con *intr*
- sigue con el número de interrupciones totales producidas
- continúa con las que se han producido para cada línea (comenzando por la 0) desde el arranque del sistema

Manejadores de interrupción

El objetivo del manejador es:

- indicar al dispositivo que la interrupción se ha reconocido, generalmente limpiando un bit en un registro del HW
- leer o escribir datos en el dispositivo
- despertar si es necesario a un thread que espera al dispositivo

Manejadores rápidos y normales:

- un manejador ***rápido*** es atómico
- un manejador ***normal*** no es atómico ya que puede ser expulsado por otros manejadores

Manejadores de interrupción (cont.)

Para escribir un manejador hay que tener en cuenta:

- no puede transferir datos desde o hacia el espacio de usuario, porque no ejecuta en el contexto de un proceso
- no debe dormirse o suspenderse
- el código debe ser lo más breve posible
  - puede descargar parte de la actividad a otro contexto (*tasklet* o *bottom-half*)

Manejadores de interrupción (cont.)

El manejador tendrá habitualmente la siguiente estructura:

```
irqreturn_t driver_interrupt (int irq, void *dev_id,
                              struct pt_regs *regs)
{
    ...
    wake_up_interruptible (&driver_read_queue);
    return IRQ_HANDLED;
}
```

La interrupción en este caso despierta a una de las tareas que están esperando en una cola de eventos

Argumentos del manejador

Aunque el manejador puede no utilizar los argumentos, éstos son útiles en los siguientes casos:

- `irq`: es el número de interrupción (si hay un solo manejador para diferentes líneas de interrupción)
- `dev_id`: es el mismo puntero usado al registrar la interrupción; se suele usar para identificar el dispositivo
- `regs`: raramente se utiliza y contiene una imagen del contexto del procesador antes de entrar en el código de interrupción

Valor retornado; se usa uno de estos valores

- `IRQ_HANDLED`: la interrupción se atendió OK
- `IRQ_NONE`: la interrupción no necesitaba atención; el kernel usa este valor para identificar interrupciones espúreas

Habilitación de interrupciones

En la sincronización con manejadores de interrupción el mecanismo de inhibición suele ser bastante habitual a diferentes escalas (en kernels de tiempo real se suelen inhibir todas)

A partir del kernel 2.6 en Linux no hay un mecanismo para habilitar e inhibir globalmente las interrupciones:

- se hacía con `cli` y `sti`
- pensado más para arquitecturas monoprocesadoras

En su lugar se ofrecen dos posibilidades de habilitación e inhibición:

- de línea: afecta sólo a una línea en todo el sistema
- de procesador: afecta sólo localmente a un procesador

Habilitación de interrupciones (cont.)

El kernel ofrece tres funciones para inhibir y habilitar las interrupciones de una determinada línea (en `<asm/irq.h>`):

```
void disable_irq(int irq); // espera al fin de la ISR
void disable_irq_nosync(int irq); // no espera
void enable_irq(int irq);
```

Existen en `<asm/system.h>` otras dos funciones que sirven para inhibir y habilitar todas las interrupciones en el procesador local

- permiten conseguir exclusión mutua entre un manejador y una parte del driver que comparten datos si está en el mismo procesador
- en un multiprocesador es mejor usar los spinlocks

Habilitación de interrupciones (cont.)

Inhibir

```
void local_irq_save(unsigned long flags);
   // inhibe las interrupciones y guarda en flags (que
   // se pasa por valor) el estado actual de interr.
void local_irq_disable(void);
   // no salva el estado; usar con cautela
```

Habilitar

```
void local_irq_restore(unsigned long flags);
   // restaura el estado de interrupción
void local_irq_enable(void);
   // habilita las interrupciones; usar con cautela
```

II. Programación de E/S

Bloque II

- Operaciones de control en el driver
- Sincronización
- Temporización
- Acceso al hardware
- Interrupciones
- Threads del kernel

Gestión de interrupciones en dos mitades

Es preciso aligerar el código de los manejadores de interrupción muy largos

Linux parte el código del manejador de interrupción en dos mitades:

- superior (*top half*): es la rutina que responde directamente a la interrupción
- inferior (*bottom half*): es una rutina que es planificada por el *top half* y que será ejecutada más tarde

Lo importante es que las interrupciones continúan habilitadas durante la ejecución del *bottom half*

Gestión de interrupciones en dos mitades (cont.)

La actividad del manejador de interrupción habitualmente será:

- salvar los datos del dispositivo en algún buffer
- y planificar el *bottom half*

El *bottom half* por su parte se encargará, por ejemplo, de:

- despertar a los procesos en espera
- iniciar una nueva operación de I/O

Hay dos mecanismos para implementar el *bottom half*

- *tasklets*: rápidos, requieren ser atómicos
- *colas de trabajo*: más lentos, pero no requieren ser atómicos (pueden dormirse, usar semáforos, etc.)

Tasklets

Son funciones con las siguientes propiedades:

- se ejecutan en el contexto de las interrupciones software
- se ejecutan una sola vez, incluso aunque se planifiquen varias veces antes de su ejecución
- se puede inhibir y activar más tarde
- se puede planificar a sí mismo
- tienen dos prioridades (normal o alta)
- se ejecutan en la misma CPU que la función que lo planifica
  - el *tasklet* ejecuta después de su manejador

Tasklets (cont.)

- se puede ejecutar inmediatamente si el sistema está ocioso, pero en cualquier caso nunca más tarde del primer *tick*
- un *tasklet* puede ser concurrente con otros *tasklets* pero se serializa respecto a sí mismo (incluso en diferentes procesadores)
- puede ser necesaria la exclusión mutua entre distintos *tasklets* en sistemas multiprocesadores
- puede necesitar exclusión mutua entre un *tasklet* y su ISR

Tasklets: API

El *tasklet* se representa con una variable del tipo `struct tasklet_struct` definido en `<linux/interrupt.h>`:

```
struct tasklet_struct {
   /* ... */
   void (*func) (unsigned long);
   unsigned long data;
};
```

- El *tasklet* ejecutará la función `func`, a la que se le pasa el argumento `data`

Se debe inicializar con

```
void tasklet_init(struct tasklet_struct *t,
                  void (*func) (unsigned long),
                  unsigned long data);
```

Tasklets: API (cont.)

También se puede declarar e inicializar con la macro,

```
DECLARE_TASKLET(name,func,data);
```

- el *tasklet* se representa en la variable `name`

Planificar el *tasklet* con prioridad normal o alta

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
```

- si ya había comenzado, se planifica de nuevo
- este comportamiento permite a un *tasklet* planificarse a sí mismo

Tasklets: API (cont.)

Inhibir (esperando a que acabe o sin esperar) y habilitar el *tasklet*

```
void tasklet_disable(struct tasklet_struct *t);
void tasklet_disable_nosync(struct tasklet_struct *t);
void tasklet_enable(struct tasklet_struct *t);
```

- un *tasklet* debe ser habilitado tantas veces como ha sido inhibido

Eliminar el *tasklet*

```
void tasklet_kill(struct tasklet_struct *t);
```

- si el *tasklet* ha sido planificado espera a que haya ejecutado
- si el *tasklet* se planifica a sí mismo, no debe hacerlo más cuando se va a llamar a esta función (similar a `del_timer_sync`)

Colas de trabajos (workqueues)

Se parecen a los *tasklets*, pero:

- no se ejecutan como interrupciones software
- no requieren ser atómicos (pueden suspenderse)
- tienen menor prioridad
- se puede ordenar el retraso por un tiempo dado de la ejecución de la cola

Al igual que los *tasklets* se ejecutan en el mismo procesador que las planificó

Se representan con datos del tipo `struct workqueue_struct`, definido en `<linux/workqueue.h>`

Colas de trabajos: API

Crear una cola de trabajo con un thread propio por cada procesador, o uno compartido:

```
struct workqueue_struct *create_workqueue
       (const char *name);
struct workqueue_struct
     *create_singlethread_workqueue
       (const char *name);
```

Para dar un trabajo a una cola de trabajos se usa el tipo `struct work_struct` que se declara con la macro

```
DECLARE_WORK(name, void(*function)(void *),
                   void *data);
```

- La variable se llama `name`; el trabajo ejecutará `function` usando como argumento `data`

Colas de trabajos: API (cont.)

Se puede inicializar una `struct work_struct` al ejecutar con las macros:

```
INIT_WORK(struct work_struct *work,
          void(*function)(void *),void *data);
PREPARE_WORK(struct work_struct *work,
             void(*function)(void *),void *data);
```

- usar `INIT_WORK` la primera vez

Para añadir un trabajo a la cola

```
int queue_work(struct workqueue_struct *queue,
               struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue,
                       struct work_struct *work,
                        unsigned long delay);
// espera delay jiffies para comenzar el trabajo
```


Colas de trabajos: API (cont.)

Cancelar un trabajo pendiente

```
int cancel_delayed_work(struct work_struct *work);
```

- devuelve un valor distinto de cero si ha ido bien
- devuelve cero si hay trabajos ejecutando en otros procesadores

Esperar a que el trabajo cancelado acabe si había empezado

```
void flush_workqueue(struct workqueue_struct *queue);
```

Eliminar una cola de trabajos

```
void destroy_workqueue
       (struct workqueue_struct *queue);
```