Entrada/Salida - *Drivers*

Sistemas Operativos
DC - FCEN - UBA

02 de octubre de 2025

Drivers y Controllers

Drivers Módulos de software que pueden ser añadidos al SO para manejar los dispositivos de E/S.

Controllers Componente mecánico y/o electrónico que trabaja como una interfaz entre un dispositivo y el driver.

Drivers y Controllers

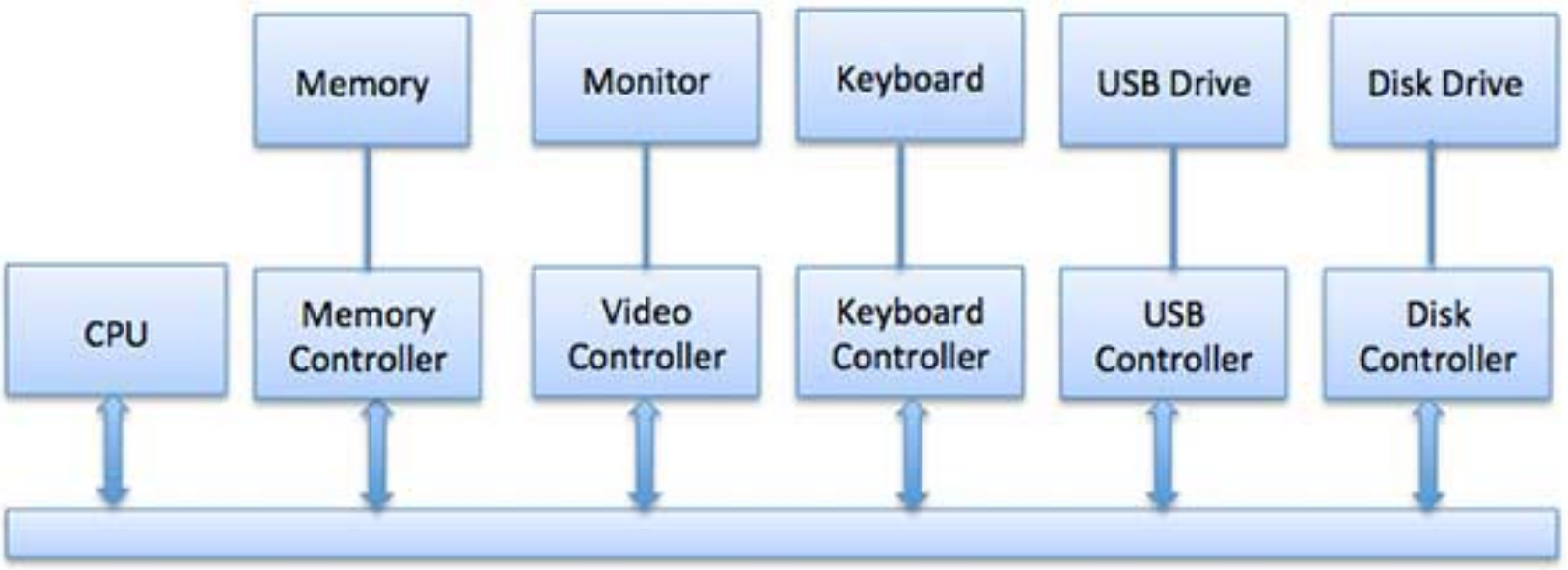


Figure: Controladores de hardware (*Controllers*)

E/S Asíncrona vs síncrona

Síncrona La ejecución de la CPU que solicita la E/S, espera por su culminación.

Asíncrona Cada E/S procede concurrentemente con la ejecución del CPU que la solicita.

Polling vs Interrupciones

- Vías para detectar la llegada de cualquier tipo de entrada.
- Las dos técnicas le permiten a la CPU atender los eventos que suceden en cualquier momento, y que no están relacionados a los procesos en ejecución.
- Polling:
  - Más simple.
  - Revisión periódica del estatus del dispositivo.
  - Utiliza un registro de estatus.
- Interrupciones:
  - El *controller* coloca una señal en el bus, cuando necesita atención de la CPU.
  - Utiliza los *handlers* o rutinas de software para manejar cada tipo de interrución.

Software para E/S

- **Bibliotecas a nivel de usuario**: Ejemplo, **stdio** del lenguaje C y C++.
- **Módulos de nivel del Kernel**: Son los drivers.
- **Hardware**: Por ejemplo, los *firmware*.

Software para E/S

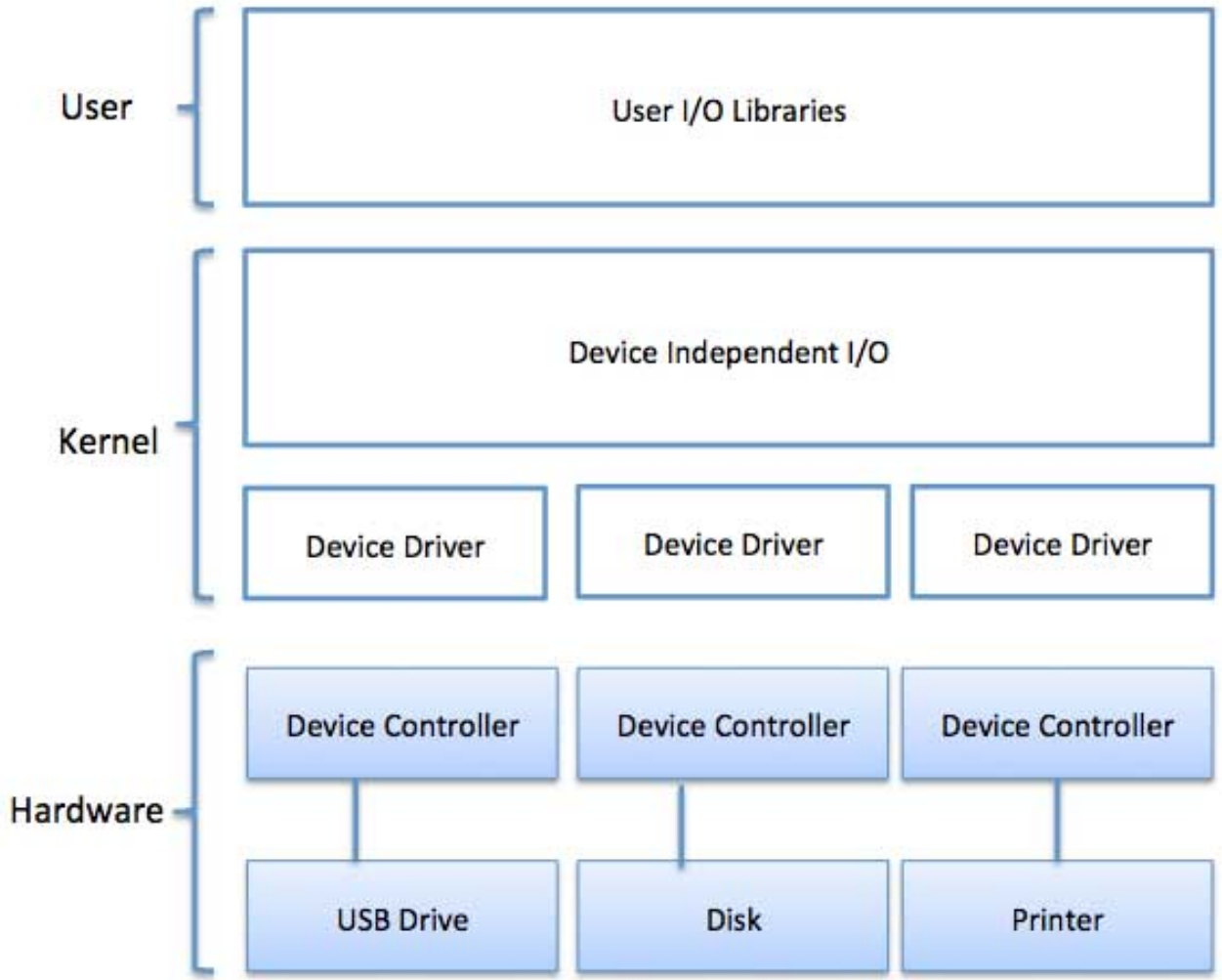


Figure: Software para E/S

Drivers

- Acepta solicitudes del software independiente del dispositivo, que a su vez, las recibe del nivel de usuario.
- Interactúa con el *controller* para recibir o enviar una E/S.
- Maneja los errores respectivos.
- Se asegura que la solicitud de ejecute exitosamente.

Handlers de interrupciones

- Conocidas como rutinas de servicios de interrupciones o **ISR** (por su siglas en inglés).
- Específicamente se les denomina *callback functions*.
- Se alojan en el *driver*.
- Se ordenan numéricamente, a lo que se le denomina direcciones.

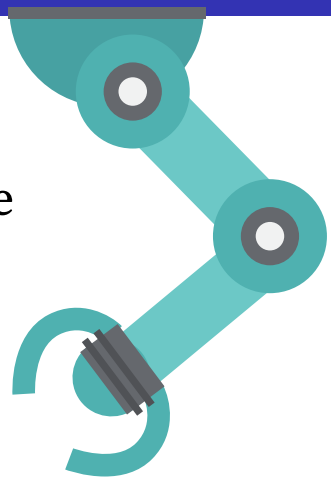
Sofware a nivel de usuario

- Interfaz simplificada.
- Consiste en procedimientos/funciones alojados en bibliotecas.
- Están en el espacio de usuario (**stdio**).

Subsistema de E/S del Kernel

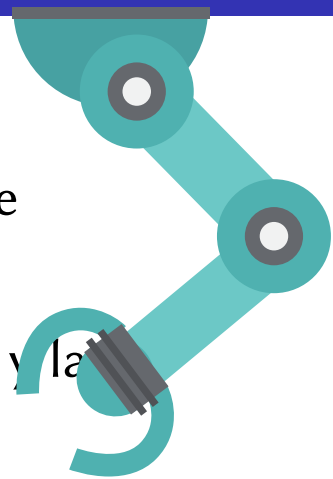
- Scheduling.
- Buffering.
- Caching.
- Spooling.
  - Un *spool* es un buffer que mantiene los datos enviados a un dispositivo.
  - No acepta envíos intercalados, solo solicitudes completas.
  - Típicamente usado para el servicio de impresión.
  - Mantiene una cola de archivos enviados. Se atienden uno a la vez.
- Manejo de errores.

El bot y la caja



Una pequeña empresa de logística acaba de adquirir un **ROBOT** que permite localizar y obtener cajas en su depósito.

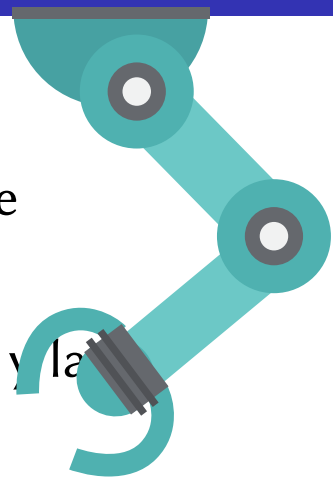
El bot y la caja



Una pequeña empresa de logística acaba de adquirir un **ROBOT** que permite localizar y obtener cajas en su depósito.

Cuando se le ingresa un código en el registro de 32 bits `LOC_TARGET` y la constante `START` en el registro `LOC_CTRL`, el robot comienza la operación de búsqueda, escribiendo el valor `BUSY` en el registro `LOC_STATUS`.

El bot y la caja

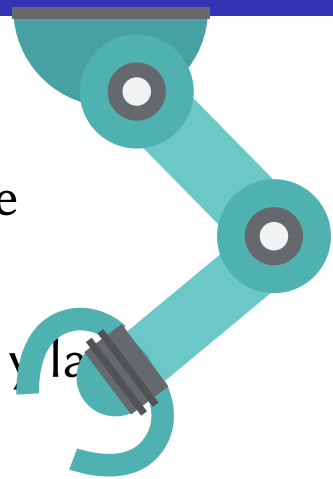


Una pequeña empresa de logística acaba de adquirir un **ROBOT** que permite localizar y obtener cajas en su depósito.

Cuando se le ingresa un código en el registro de 32 bits `LOC_TARGET` y la constante `START` en el registro `LOC_CTRL`, el robot comienza la operación de búsqueda, escribiendo el valor `BUSY` en el registro `LOC_STATUS`.

Al encontrar la caja, la deposita en la bandeja de salida, escribe el valor `JOYA` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`.

El bot y la caja



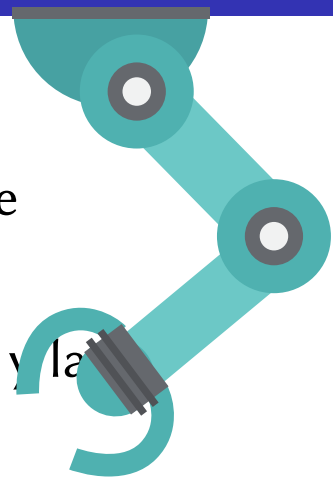
Una pequeña empresa de logística acaba de adquirir un **ROBOT** que permite localizar y obtener cajas en su depósito.

Cuando se le ingresa un código en el registro de 32 bits `LOC_TARGET` y la constante `START` en el registro `LOC_CTRL`, el robot comienza la operación de búsqueda, escribiendo el valor `BUSY` en el registro `LOC_STATUS`.

Al encontrar la caja, la deposita en la bandeja de salida, escribe el valor `JOYA` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`.

Si no puede encontrar la caja, escribe el valor `BAJON` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`.

El bot y la caja



Una pequeña empresa de logística acaba de adquirir un **ROBOT** que permite localizar y obtener cajas en su depósito.

Cuando se le ingresa un código en el registro de 32 bits `LOC_TARGET` y la constante `START` en el registro `LOC_CTRL`, el robot comienza la operación de búsqueda, escribiendo el valor `BUSY` en el registro `LOC_STATUS`.

Al encontrar la caja, la deposita en la bandeja de salida, escribe el valor `JOYA` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`.

Si no puede encontrar la caja, escribe el valor `BAJON` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`.

En todos los casos el contenido de `LOC_TARGET` se mantiene hasta tanto se vuelva a escribir otro valor.

El bot y la caja

El robot vino con el siguiente **SOFTWARE**:

```
int main (int argc, char *argv[]) {
    int robot = open("/dev/chinbot", "w");
    int codigo;
    int resultado;
    while (1) {
        printf("Ingrese el código de la caja\n");
        scanf ("%d", &codigo);
        resultado = write(robot, codigo);
        if (resultado == 1) {
            printf("Su orden ha llegado\n");
        } else {
            printf("No podemos encontrar su caja %d\n", codigo);
        }
    }
}
```

El bot y la caja

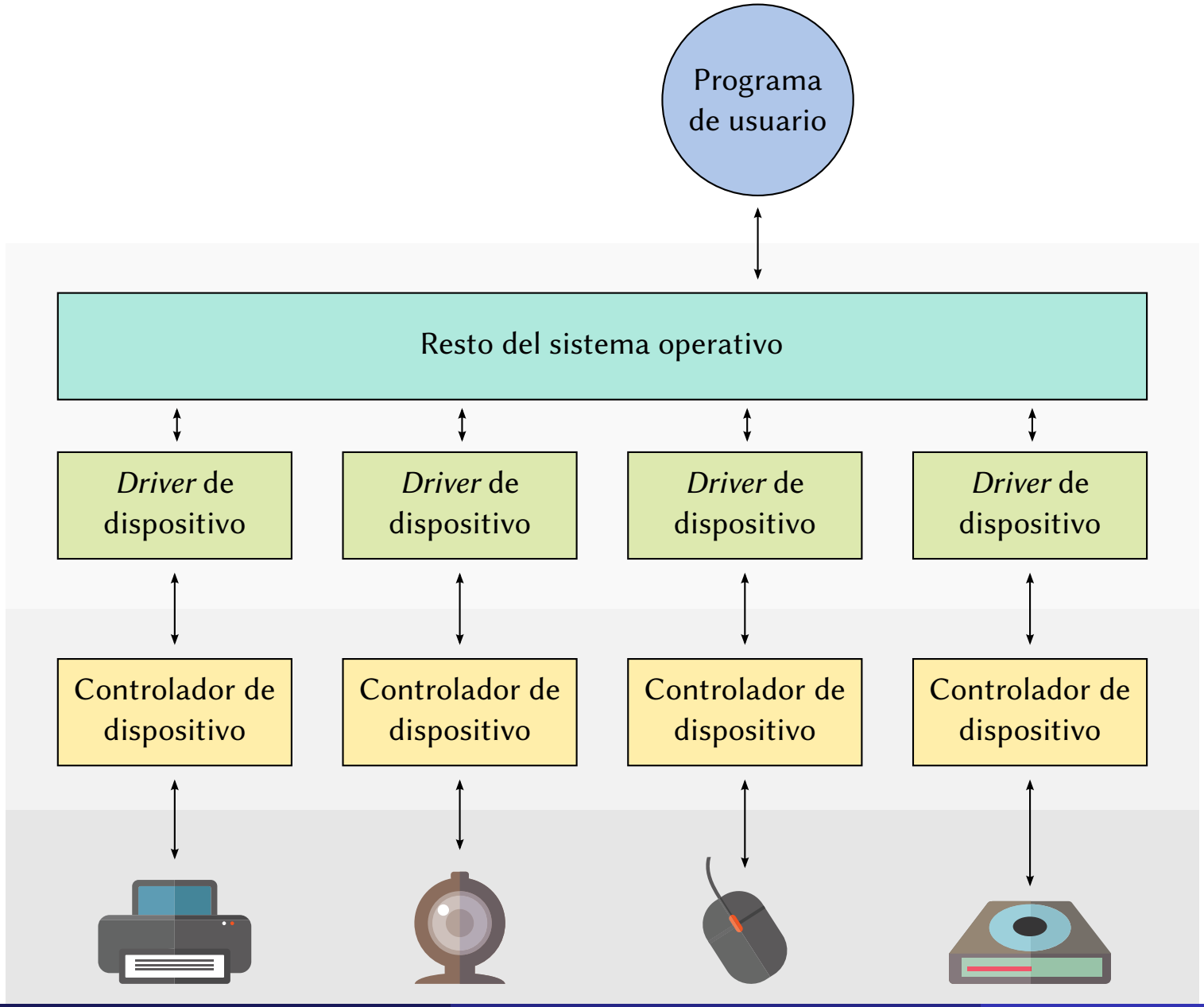
Desafortunadamente, el **DRIVER** que vino con el robot parece no ser compatible con el **SISTEMA OPERATIVO** que utiliza la empresa. Al intentar comunicarse con los fabricantes para obtener soporte, la respuesta que obtuvieron fue “谢谢。很快回来。”. Por lo tanto, han decidido recurrir a nuestra ayuda.

El bot y la caja

Desafortunadamente, el **DRIVER** que vino con el robot parece no ser compatible con el **SISTEMA OPERATIVO** que utiliza la empresa. Al intentar comunicarse con los fabricantes para obtener soporte, la respuesta que obtuvieron fue “谢谢。很快回来。”. Por lo tanto, han decidido recurrir a nuestra ayuda.

- Identificar en el siguiente diagrama los elementos resaltados del enunciado.

El SO y los dispositivos de E/S



Pensando nuestro primer driver

- ¿Cuándo el código de usuario que vino con el robot necesita hacer uso del *driver* del dispositivo?

Pensando nuestro primer driver

```c
int main (int argc, char *argv[]) {
    int robot = open("/dev/chinbot", "w");
    int codigo;
    int resultado;
    while (1) {
        printf("Ingrese el código de la caja\n");
        scanf ("%d", &codigo);
        resultado = write(robot, codigo);
        if (resultado == 1) {
            printf("Su orden ha llegado\n");
        } else {
            printf("No podemos encontrar su caja %d\n", codigo);
        }
    }
}
```

Pensando nuestro primer driver

```c
int main (int argc, char *argv[]) {
    int robot = open("/dev/chinbot", "w"); // open device
    int codigo;
    int resultado;
    while (1) {
        printf("Ingrese el código de la caja\n");
        scanf ("%d", &codigo);
        resultado = write(robot, codigo); // write on device
        if (resultado == 1) {
            printf("Su orden ha llegado\n");
        } else {
            printf("No podemos encontrar su caja %d\n", codigo);
        }
    }
}
```

Pensando nuestro primer driver

- ¿Cuándo el código de usuario que vino con el robot necesita hacer uso del *driver* del dispositivo?

Pensando nuestro primer driver

- ¿Cuándo el código de usuario que vino con el robot necesita hacer uso del *driver* del dispositivo?
- ¿Qué funciones debería proveer el *driver* que programemos?

La API de un Driver

Un *driver* debe implementar los siguientes procedimientos para ser cargado por el sistema operativo.

- `int driver_init()`

  Invocada durante la carga del SO.

- `int driver_open()`

  Invocada al solicitarse un *open*.

- `int driver_close()`

  Invocada al solicitarse un *close*.

- `int driver_read(int *data)`

  Invocada al solicitarse un *read*.

- `int driver_write(int *data)`

  Invocada al solicitarse un *write*.

- `int driver_remove()`

  Invocada durante la descarga del SO.

Funciones del kernel para drivers

Además para la programación de un *driver*, se dispone de las siguientes *syscalls* (listado NO exhaustivo...):

- `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.

Pensando nuestro primer driver

- ¿Cuándo el código de usuario que vino con el robot necesita hacer uso del *driver* del dispositivo?
- ¿Qué funciones debería proveer el *driver* que programemos?
- Pensar, a grandes rasgos, cómo podríamos implementar la función `int driver_write(void* data)` del *driver*.

Pensando nuestro primer driver

```
int driver_write(void* data) {



    OUT(LOC_TARGET, *data);
    OUT(LOC_CTRL, START);

    while (IN(LOC_STATUS) != BUSY) {}
    while (IN(LOC_STATUS) != READY) {}

    resultado = IN(LOC_CTRL);
    if (resultado == JOYA)
        return 1;
    else if (resultado == BAJON)
        return 0;
    return -1;
}
```

Pensando nuestro primer driver

```
int driver_write(void* data) {



    OUT(LOC_TARGET, *data);
    OUT(LOC_CTRL, START);

    while (IN(LOC_STATUS) != BUSY) {}
    while (IN(LOC_STATUS) != READY) {}

    resultado = IN(LOC_CTRL);
    if (resultado == JOYA)
        return 1;
    else if (resultado == BAJON)
        return 0;
    return -1;
}
```

- ¿Este código funciona bien?

Ojo con los punteros que nos pasa el usuario

```
int driver_write(void* data) {
    // Copio los datos que me pasa usuario
    int codigo;
    copy_from_user(&codigo, data, sizeof(int));

    OUT(LOC_TARGET, codigo);
    OUT(LOC_CTRL, START);

    while (IN(LOC_STATUS) != BUSY) {}
    while (IN(LOC_STATUS) != READY) {}

    resultado = IN(LOC_CTRL);
    if (resultado == JOYA)
        return 1;
    else if (resultado == BAJON)
        return 0;
    return -1;
}
```

Ojo con los punteros que nos pasa el usuario

```
int driver_write(void* data) {
    // Copio los datos que me pasa usuario
    int codigo;
    copy_from_user(&codigo, data, sizeof(int));

    OUT(LOC_TARGET, codigo);
    OUT(LOC_CTRL, START);

    while (IN(LOC_STATUS) != BUSY) {}
    while (IN(LOC_STATUS) != READY) {}

    resultado = IN(LOC_CTRL);
    if (resultado == JOYA)
        return 1;
    else if (resultado == BAJON)
        return 0;
    return -1;
}
```

- ¿Ahora sí?

Ay, la concurrencia...

```
int driver_write(void* data) {
    int codigo;
    copy_from_user(&codigo, data, sizeof(int));

    mutex.lock(); // Inicia sección crítica
    OUT(LOC_TARGET, codigo);
    OUT(LOC_CTRL, START);

    while (IN(LOC_STATUS) != BUSY) {}
    while (IN(LOC_STATUS) != READY) {}

    resultado = IN(LOC_CTRL);
    mutex.unlock(); // Fin sección crítica

    if (resultado == JOYA)
        return 1;
    else if (resultado == BAJON)
        return 0;
    return -1;
}
```

Cosas para tener en cuenta

- Un *driver* corre dentro del contexto de un proceso.

Cosas para tener en cuenta

- Un *driver* corre dentro del contexto de un proceso.
- Esto significa que puede acceder a sus datos.

Cosas para tener en cuenta

- Un *driver* corre dentro del contexto de un proceso.
- Esto significa que puede acceder a sus datos.
- ¡Cuidado con los punteros que nos pasa el usuario! (`copy_from_user()`, `copy_to_user()`).

Cosas para tener en cuenta

- Un *driver* corre dentro del contexto de un proceso.
- Esto significa que puede acceder a sus datos.
- ¡Cuidado con los punteros que nos pasa el usuario! (`copy_from_user()`, `copy_to_user()`).
- Muchos procesos pueden querer ejecutar el *driver* a la vez. El resultado: horribles *race conditions*.

Cosas para tener en cuenta

- Un *driver* corre dentro del contexto de un proceso.
- Esto significa que puede acceder a sus datos.
- ¡Cuidado con los punteros que nos pasa el usuario! (`copy_from_user()`, `copy_to_user()`).
- Muchos procesos pueden querer ejecutar el *driver* a la vez. El resultado: horribles *race conditions*.
- ¿Cuándo inicializamos las primitivas de sincronización? ¿Y las estructuras de datos que pueda necesitar el *driver*?

Cosas para tener en cuenta

- Un *driver* corre dentro del contexto de un proceso.
- Esto significa que puede acceder a sus datos.
- ¡Cuidado con los punteros que nos pasa el usuario! (`copy_from_user()`, `copy_to_user()`).
- Muchos procesos pueden querer ejecutar el *driver* a la vez. El resultado: horribles *race conditions*.
- ¿Cuándo inicializamos las primitivas de sincronización? ¿Y las estructuras de datos que pueda necesitar el *driver*? Respuesta: al cargar el *driver* en el *kernel* (`driver_init()`).

Cosas para tener en cuenta

- Un *driver* corre dentro del contexto de un proceso.
- Esto significa que puede acceder a sus datos.
- ¡Cuidado con los punteros que nos pasa el usuario! (`copy_from_user()`, `copy_to_user()`).
- Muchos procesos pueden querer ejecutar el *driver* a la vez. El resultado: horribles *race conditions*.
- ¿Cuándo inicializamos las primitivas de sincronización? ¿Y las estructuras de datos que pueda necesitar el *driver*? Respuesta: al cargar el *driver* en el *kernel* (`driver_init()`).
- Un *driver* no se *linkea* contra bibliotecas, así que solo se pueden usar funciones que sean parte del *kernel*.

Métodos de acceso

- ¿Que **método de acceso** emplea nuestro *driver*?

Métodos de acceso

```
int driver_write(void* data) {
    int codigo;
    copy_from_user(&codigo, data, sizeof(int));

    mutex.lock();
    OUT(LOC_TARGET, codigo);
    OUT(LOC_CTRL, START);

    while (IN(LOC_STATUS) != BUSY) {}
    while (IN(LOC_STATUS) != READY) {}

    resultado = IN(LOC_CTRL);
    mutex.unlock();

    if (resultado == JOYA)
        return 1;
    else if (resultado == BAJON)
        return 0;
    return -1;
}
```

Métodos de acceso

```
int driver_write(void* data) {
    int codigo;
    copy_from_user(&codigo, data, sizeof(int));

    mutex.lock();
    OUT(LOC_TARGET, codigo);
    OUT(LOC_CTRL, START);

    while (IN(LOC_STATUS) != BUSY) {} // Polling
    while (IN(LOC_STATUS) != READY) {} // Polling

    resultado = IN(LOC_CTRL);
    mutex.unlock();

    if (resultado == JOYA)
        return 1;
    else if (resultado == BAJON)
        return 0;
    return -1;
}
```

Métodos de acceso

- ¿Que **método de acceso** emplea nuestro *driver*?

Métodos de acceso

- ¿Que **método de acceso** emplea nuestro *driver*?
- Así que ***polling***... ¿Y eso *es bueno o malo*?

Métodos de acceso

- ¿Que **método de acceso** emplea nuestro *driver*?
- Así que ***polling***... ¿Y eso *es bueno o malo*?
- ¿Qué alternativa tenemos? ¿Qué ventajas y desventajas tiene?

Métodos de acceso

- ¿Que **método de acceso** emplea nuestro *driver*?
- Así que ***polling***... ¿Y eso *es bueno o malo*?
- ¿Qué alternativa tenemos? ¿Qué ventajas y desventajas tiene?
- Para poder implementar el *driver* usando **interrupciones**, ¿debería cambiar algo en el *hardware* de nuestro robot?

Métodos de acceso

- ¿Que **método de acceso** emplea nuestro *driver*?
- Así que ***polling***... ¿Y eso *es bueno o malo*?
- ¿Qué alternativa tenemos? ¿Qué ventajas y desventajas tiene?
- Para poder implementar el *driver* usando **interrupciones**, ¿debería cambiar algo en el *hardware* de nuestro robot?
- Parece que el manual del robot, escrito en un dudoso castellano, contiene la siguiente información:

  > "*Robot es compatible con el acceso de interrupción.*
  > *Se selecciona este modo, una operación terminada*
  > *`CHINBOT_INT` interrupción lanzará.*"

  Aprovechando esta información, modificar el código anterior para que utilice interrupciones.

Interrupciones

```
mutex acceso;
semaforo listo;
bool esperando;

int driver_init() {
    acceso = mutex_create();
    listo = semaforo_create(0);
    esperando = false;
    irq_register(CHINBOT_INT, handler);
}

void handler() {
    if (esperando && IN(LOC_STATUS) == READY) {
        esperando = false;
        listo.signal();
    }
}
```

Interrupciones

```
int driver_write(void* data) {
    int codigo;
    copy_from_user(&codigo, data, sizeof(int));

    acceso.lock();
    OUT(LOC_TARGET, codigo);
    OUT(LOC_CTRL, START);

    esperando = true;
    listo.wait();

    resultado = IN(LOC_CTRL);
    acceso.unlock();

    if (resultado == JOYA)
        return 1;
    else if (resultado == BAJON)
        return 0;
    return -1;
}
```

Momento para preguntas

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*
- Métodos de acceso:

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*
- Métodos de acceso:
  - *Polling*

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*
- Métodos de acceso:
  - *Polling*
  - *Interrupciones*

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*
- Métodos de acceso:
  - *Polling*
  - *Interrupciones*
- API de un *Driver* (`driver_init`, `driver_read`, `driver_write`, ...)

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*
- Métodos de acceso:
  - *Polling*
  - *Interrupciones*
- API de un *Driver* (`driver_init`, `driver_read`, `driver_write`, ...)
- Manejo de memoria entre el *Driver* y usuario (`copy_from_user`, `copy_to_user`)

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*
- Métodos de acceso:
  - *Polling*
  - *Interrupciones*
- API de un *Driver* (`driver_init`, `driver_read`, `driver_write`, ...)
- Manejo de memoria entre el *Driver* y usuario (`copy_from_user`, `copy_to_user`)
- Cuidado con la concurrencia en el *Driver* (`mutex`, `semaforo`)

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*
- Métodos de acceso:
  - *Polling*
  - *Interrupciones*
- API de un *Driver* (`driver_init`, `driver_read`, `driver_write`, ...)
- Manejo de memoria entre el *Driver* y usuario (`copy_from_user`, `copy_to_user`)
- Cuidado con la concurrencia en el *Driver* (`mutex`, `semaforo`)

Resumen

Hoy vimos...

- Repaso de E/S, *Drivers*
- Métodos de acceso:
  - *Polling*
  - *Interrupciones*
- API de un *Driver* (`driver_init`, `driver_read`, `driver_write`, ...)
- Manejo de memoria entre el *Driver* y usuario (`copy_from_user`, `copy_to_user`)
- Cuidado con la concurrencia en el *Driver* (`mutex`, `semaforo`)

Cómo seguimos...

Con esto se puede resolver toda la guía práctica 5.