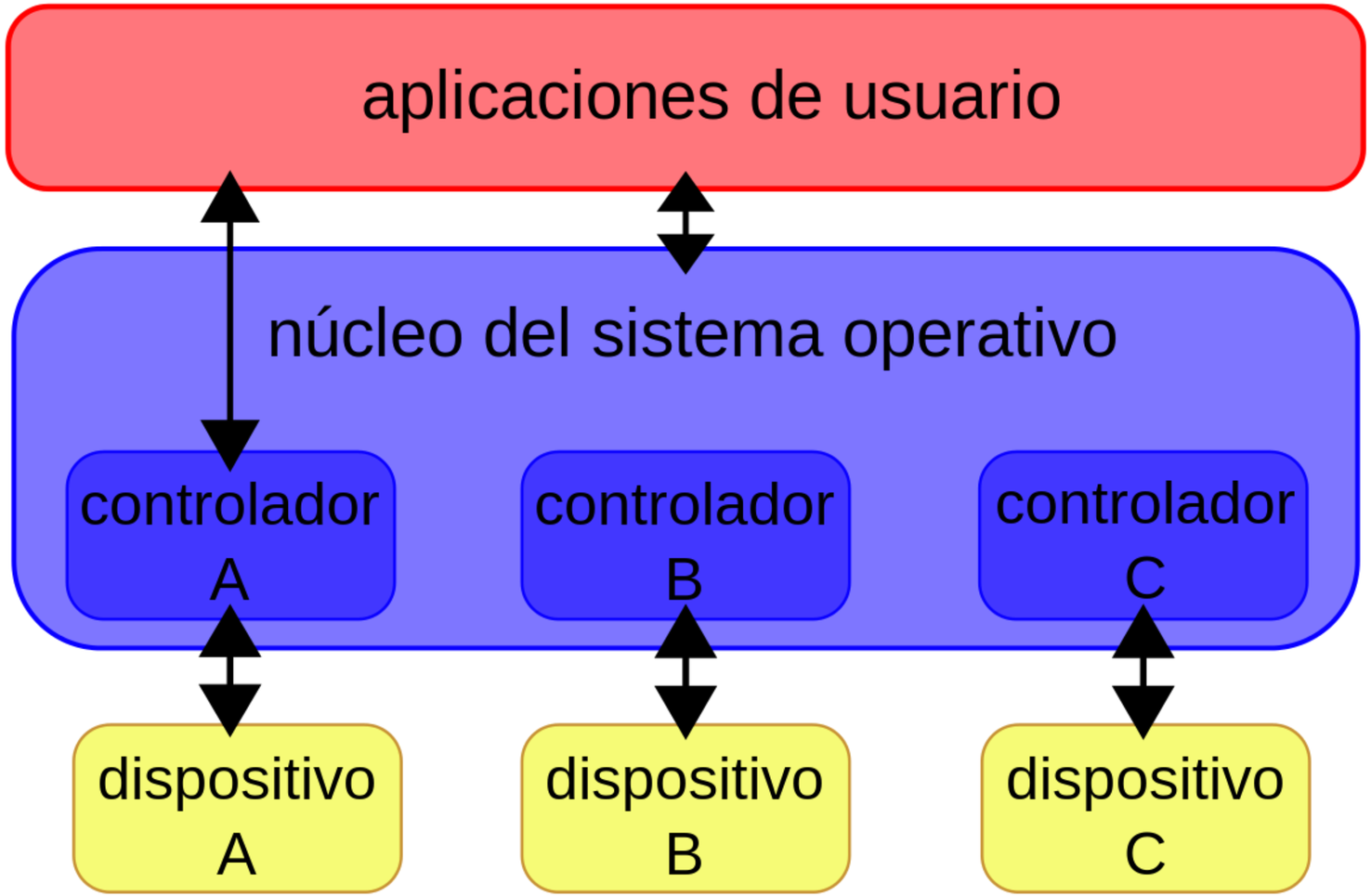
DRIVERS

Meneses Navarro Erick Sebastian
Mex Lozano Aerin Musette

¿QUÉ SON?

Un controlador o driver es un componente intermedio entre el hardware y el software que permite al sistema operativo comunicarse con los diferentes componentes del ordenador.


aplicaciones de usuario
núcleo del sistema operativo
controlador A
controlador B
controlador C
dispositivo A
dispositivo B
dispositivo C

Están compuestos por líneas de código con funciones específicas dirigidas a componentes particulares

¿CÓMO FUNCIONAN?

Se ha mencionado que, la estructura de los drivers son líneas de código que contienen instrucciones que le dicen al sistema operativo como usar el hardware que este tiene a su disposición

ENTONCES

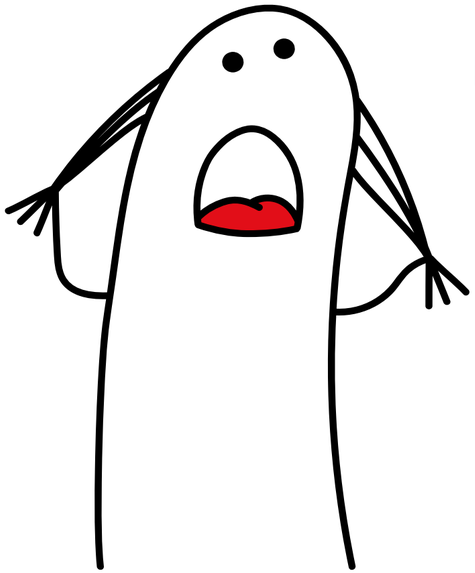
¿Qué hace exactamente el driver para que el sistema operativo logre esto?

Si tu quieres desarrollar un driver, ¿Como empezar?

REQUISITOS PREVIOS

PROFUNDO CONOCIMIENTO DEL LENGUAJE C

- Uso de punteros
- Funciones de manipulación de bits



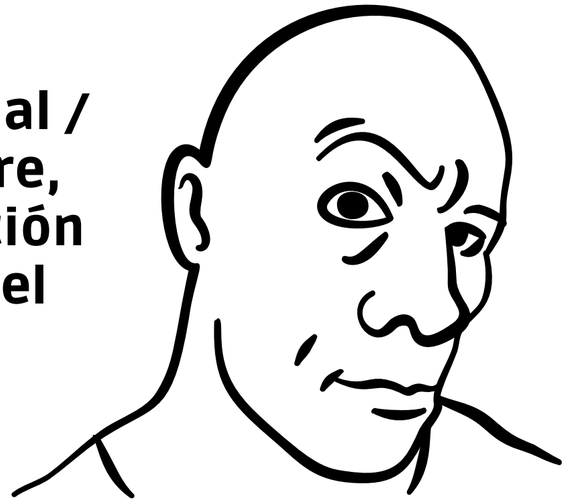
PROGRAMACIÓN CON MICROPROCESADORES

- Direccionamiento de memoria
- Interrupciones

SABER DISTINGUIR

ESPACIO DEL KERNEL

Puente entre el usuario final / programador y el hardware, cualquier subrutina o función que forma parte del kernel

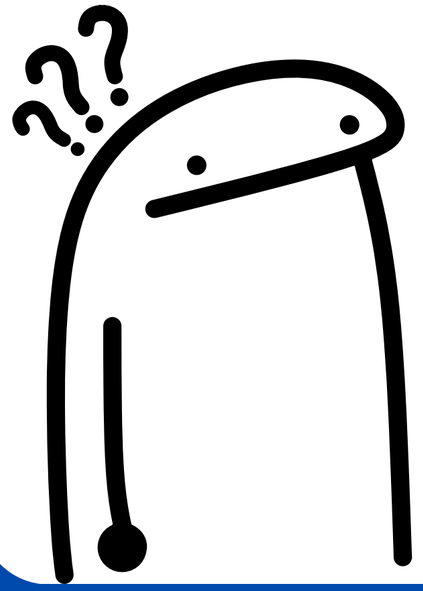


ESPACIO DEL USUARIO

Aplicaciones necesitan interactuar con el hardware del sistema, pero no directamente si no con las funciones compatibles con el kernel.

¿FUNCIONES?

Tenemos funciones de interfaz entre el espacio kernel y el espacio del usuario



El espacio kernel de linux ofrece subrutinas en el espacio usuario, permitiendo interacciones directas con el hardware.

En sistemas Linux, los propios dispositivos son vistos desde el punto de vista del usuario como ficheros, es entonces que este diálogo se hace a través de las funciones o subrutinas de lectura y escritura de ficheros.

Evento	Función del usuario	Función del kernel
Carga del módulo	insmod	init_module
Abrir dispositivo	fopen	file operations: open
Leer dispositivo	fread	file operations: read
Escribir dispositivo	fwrite	file operations: write
Cerrar dispositivo	fclose	file operations: release
Quitar módulo	rmmod	cleanup_module

EJEMPLO

Veamos un pequeño ejemplo donde se crea un código base para dispositivos USB para Linux

Para empezar, debemos crear un "esqueleto", código base para registrar el dispositivo en el subsistema de la usb en la estructura usb_driver

Esto porque a pesar de haber posibilidad conectar varios dispositivos USB, siempre habrá uno que no esté en la lista de dispositivos compatibles

```
static struct usb_driver skel_driver = {
        .name        = "skeleton",
        .probe       = skel_probe,
        .disconnect  = skel_disconnect,
        .suspend     = skel_suspend,
        .resume      = skel_resume,
        .pre_reset   = skel_pre_reset,
        .post_reset  = skel_post_reset,
        .id_table    = skel_table,
        .supports_autosuspend = 1,
};
```

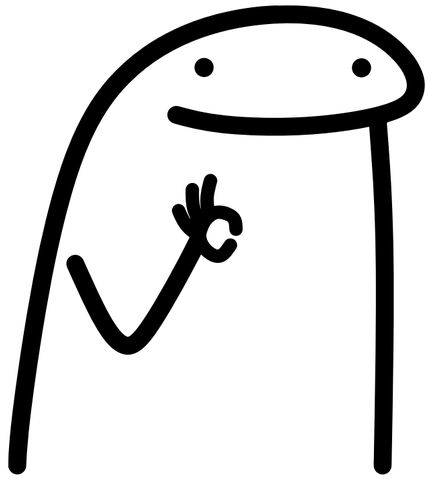
```
static int __init usb_skel_init(void)
{
        int result;

        /* register this driver with the USB subsystem */
        result = usb_register(&skel_driver);
        if (result < 0) {
                pr_err("usb_register failed for the %s driver. Error number %d\n",
                       skel_driver.name, result);
                return -1;
        }

        return 0;
}
module_init(usb_skel_init);
```

Para registrar el driver es necesario llamar a la funcion usb_registrer ubicada en la función inicial del driver

```
static void __exit usb_skel_exit(void)
{
        /* deregister this driver with the USB subsystem */
        usb_deregister(&skel_driver);
}
module_exit(usb_skel_exit);
```

Se requiere una función para quitar el módulo y cancelar el registro con el subsistema de la usb, llamando a module_exit

Ahora que tenemos la carga y descarga del módulo del dispositivo USB, es hora de ver el desarrollo del funcionamiento.



```
static int skel_probe(struct usb_interface *interface,
    const struct usb_device_id *id)
```

Cuando conectamos la USB necesita ser verificado para que coincida con el ID registrado, la usb_strcture, número de interfaz e interfaz ID.

Se determinan los puntos finales marcados como “bulk-in” y “bulk-out”, se crean los buffer para tener la información que se reciba o se envíe al dispositivo y un “urb”(USB request block) para inicializar el dispositivo.

Al momento de desconectar la USB, se tiene que limpiar toda la información privada usada en el dispositivo, además de acabar la ejecución de cualquier “urb” pendiente.

```
/* we can only write as much as 1 urb will hold */
size_t writesize = min_t(size_t, count, MAX_TRANSFER);

/* copy the data from user space into our urb */
copy_from_user(buf, user_buffer, writesize);

/* set up our urb */
usb_fill_bulk_urb(urb,
                  dev->udev,
                  usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
                  buf,
                  writesize,
                  skel_write_bulk_callback,
                  dev);

/* send the data out the bulk port */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval) {
        dev_err(&dev->interface->dev,
            "%s - failed submitting write urb, error %d\n",
            __func__, retval);
}
```

Ahora que el dispositivo está conectado al sistema y el driver está enlazado al dispositivo, cualquiera de las funciones del esqueleto se pasaron al subsistema usb y serán llamadas desde el programa usuario tratando de hablar con el dispositivo.

```
/* we can only write as much as 1 urb will hold */
size_t writesize = min_t(size_t, count, MAX_TRANSFER);

/* copy the data from user space into our urb */
copy_from_user(buf, user_buffer, writesize);

/* set up our urb */
usb_fill_bulk_urb(urb,
                  dev->udev,
                  usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
                  buf,
                  writesize,
                  skel_write_bulk_callback,
                  dev);

/* send the data out the bulk port */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval) {
        dev_err(&dev->interface->dev,
            "%s - failed submitting write urb, error %d\n",
            __func__, retval);
}
```

La función de apertura del dispositivo será llamada, mientras el programa trata de abrir el dispositivo mediante E/S esto con las file operations, después de que se realice ,las demás funciones como la de lectura y escritura serán llamadas para recibir y enviar los datos hacia el dispositivo

```
/* do an immediate bulk read to get data from the device */
retval = usb_bulk_msg (skel->dev,
                       usb_rcvbulkpipe (skel->dev,
                       skel->bulk_in_endpointAddr),
                       skel->bulk_in_buffer,
                       skel->bulk_in_size,
                       &count, 5000);
/* if the read was successful, copy the data to user space */
if (!retval) {
        if (copy_to_user (buffer, skel->bulk_in_buffer, count))
                retval = -EFAULT;
        else
                retval = count;
}
```

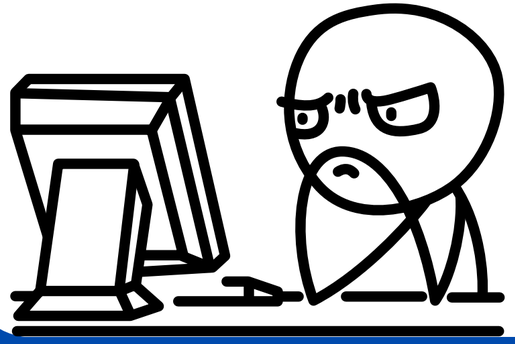
La función de lectura no utiliza la urb para pasar la información si no que llama a la funcion usb_bulk_msg() para mandar y recibir la información del dispositivo al driver

```
/* decrement our usage count for the device */
--skel->open_count;
```

Cuando el programa usuario deja de utilizar el driver para comunicarse con el dispositivo se usa la función release() decrementando el valor de conteo de uso privado esperando peticiones potenciales de escritura

IMPORTANTE TOMAR EN CUENTA

Un detalle que se debe tomar en consideración es cuando de manera repentina se extrae la USB en cualquier momento del sistema aún cuando el programa se esté comunicando con él



```
static inline void skel_delete (struct usb_skel *dev)
{
    kfree (dev->bulk_in_buffer);
    if (dev->bulk_out_buffer != NULL)
        usb_free_coherent (dev->udev, dev->bulk_out_size,
            dev->bulk_out_buffer,
            dev->write_urb->transfer_dma);
    usb_free_urb (dev->write_urb);
    kfree (dev);
}
```

Es necesario poder detener cualquier lectura y escritura actuales e informando a los programas de espacio usuario que el dispositivo no está

```
static inline void skel_delete (struct usb_skel *dev)
{
    kfree (dev->bulk_in_buffer);
    if (dev->bulk_out_buffer != NULL)
        usb_free_coherent (dev->udev, dev->bulk_out_size,
            dev->bulk_out_buffer,
            dev->write_urb->transfer_dma);
    usb_free_urb (dev->write_urb);
    kfree (dev);
}
```

Es necesario poder detener cualquier lectura y escritura actuales e informando a los programas de espacio usuario que el dispositivo no está

Cuando expulsamos la USB de forma correcta, al llamar a la función release() se reconoce que no hay un dispositivo y se hace la respectiva limpieza, siempre que no haya archivos abiertos en el dispositivo.

REFERENCIAS

● Aviviano. (2023, 8 marzo). ¿Qué es un controlador? - Windows drivers. Microsoft Learn. https://learn.microsoft.com/es-es/windows-hardware/drivers/gettingstarted/wh at-is-a-driver-

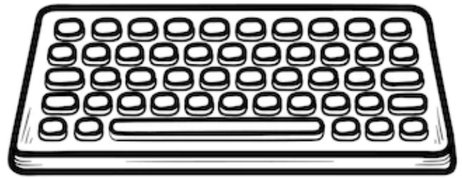
● Ferri-Benedetti, F. (2021, 25 junio). ¿Qué son los drivers o controladores? Softonic. https://www.softonic.com/articulos/que-son-los-drivers-controladores

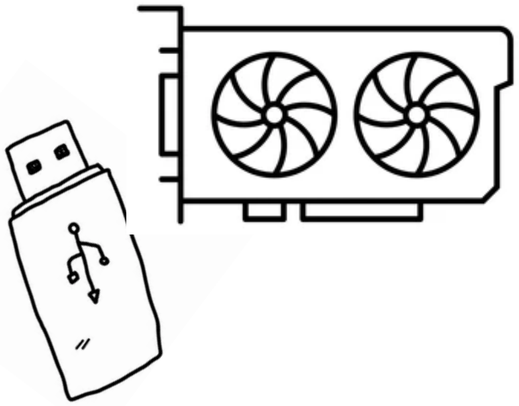
● Writing device drivers in Linux: A brief tutorial. (s. f.). http://freesoftwaremagazine.com/articles/drivers_linux/

● Writing USB Device Drivers — The Linux Kernel documentation. (s. f.). https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html

GRACIAS POR SU ATENCIÓN

DRIVERS





Meneses Navarro Erick Sebastian
Mex Lozano Aerin Musette