

ALUMNA: Priscila Haide Acosta Cano 361318. GRUPO: 5HW1

EXAMEN

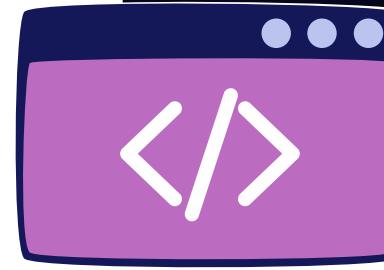
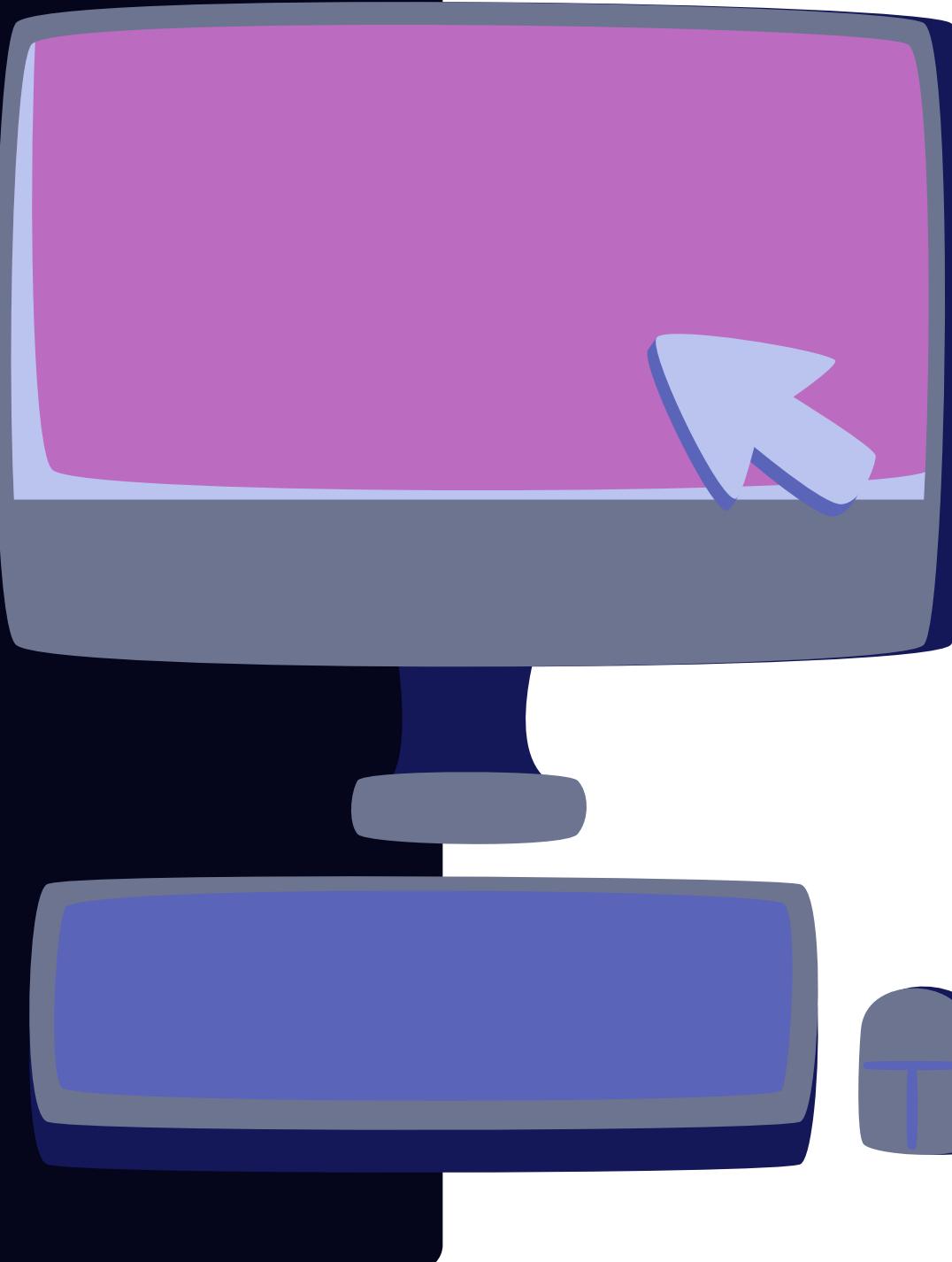
SISTEMAS OPERATIVOS I

DOCENTE: Ivan Miguel Chavero Jurado.

INTRODUCCION

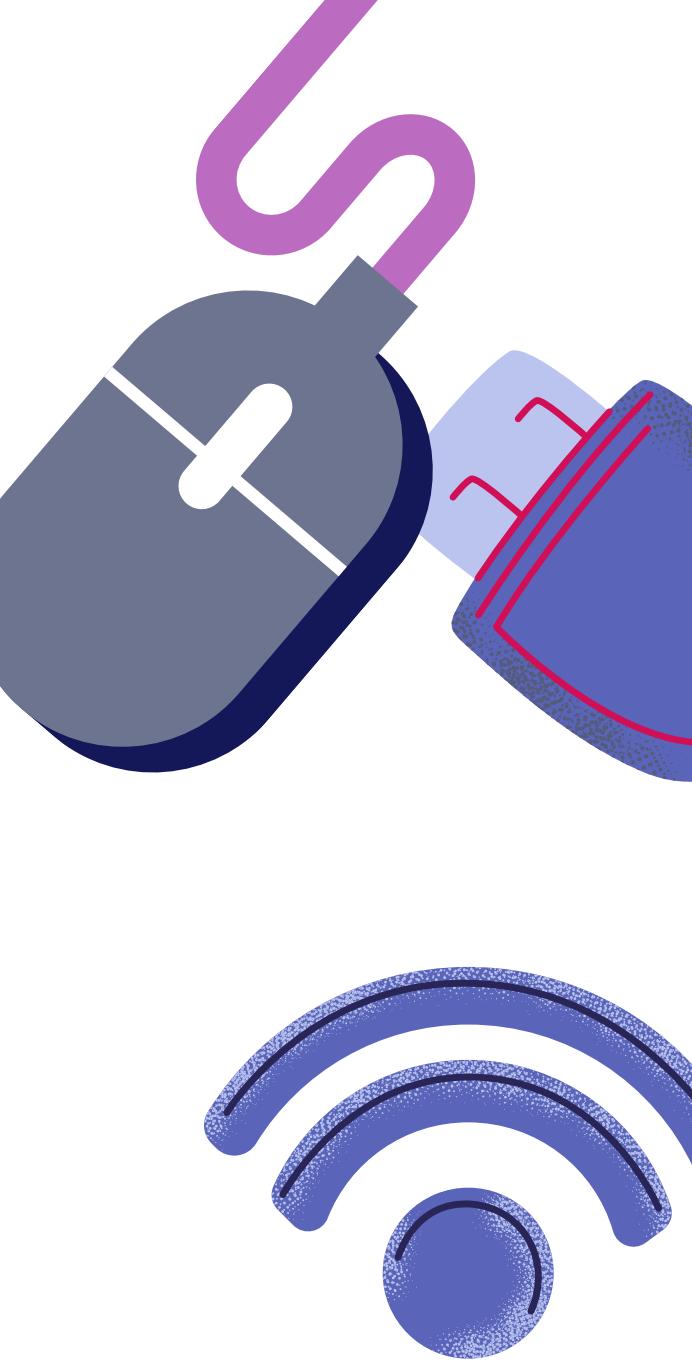
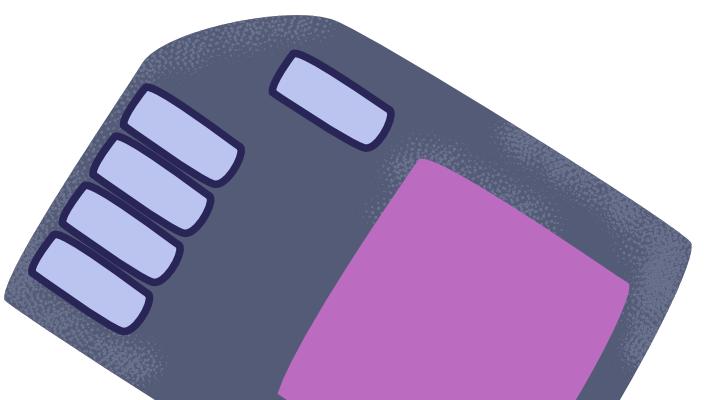
En el sistema operativo linux, los modulos del kernel son componentes que pueden cargarse y descargarse dinamicamente sin necesidad de reiniciarse el sistema. Estos módulos extienden la funcionalidad del kernel, permitiendo agregar controladores de dispositivos (drivers), sistemas de archivos, protocolos de red, entre otros.

Uno de los tipos más comunes de módulos es el módulo de dispositivo de carácter (character device). Estos permiten la interacción con hardware o recursos del sistema a través de operaciones de entrada/salida basadas en archivos, como lectura (read) y escritura (write).



CODIGO

- ESTE MODULO CREA UN DISPOSITIVO DE CARACTER EN LINUX QUE PERMITE REALIZAR OPERACIONES DE LECTURA Y ESCRITURA.
- AL ABRIR EL DISPOSITIVO, SE INCREMENTA UN CONTADOR QUE LLEVA EL REGISTRO DE LAS VECES QUE EL DISPOSITIVO HA SIDO ABIERTO.
- AL LEER, SE DEVUELVE UN MENSAJE FIJO. AL ESCRIBIR, SE ALMACENAN LOS DATOS EN UN BUFER.
- EL MODULO MANEJA LAS OPERACIONES BASICAS DE LOS DISPOSITIVOS DE CARACTER EN LINUX: OPEN, RELEASE, READ Y WRITE.



CODIGO

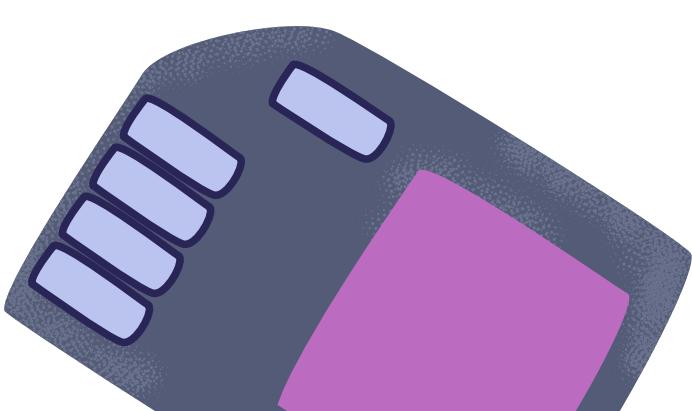
```
1 // pris.c
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/init.h>
5 #include <linux/fs.h>
6 #include <linux/uaccess.h>
7
8
9 MODULE_LICENSE("GPL");
10 MODULE_AUTHOR("PRISCILA_ACOSTA");
11 MODULE_DESCRIPTION("Un modulo pris");
12 MODULE_VERSION("1.0");
13
14 #define DEVICE_NAME "pris"
15 #define BUFFER_SIZE 1024
16
17
18 static int major_number;
19 static int device_open_count = 0;
20 static char device_buffer[BUFFER_SIZE];
21 static int exit = 0;
22
23
24 static int pris_device_open(struct inode *inode, struct file *file);
25 static int pris_device_close(struct inode *inode, struct file *file);
26 static ssize_t pris_device_read(struct file *filep, char *buffer, size_t len, loff_t *offset);
27 static ssize_t pris_device_write(struct file *filep, const char *buffer, size_t len, loff_t
    *offset);
28
29 // Define file operations structure
30 static struct file_operations fops = {
31     .open = pris_device_open,
32     .release = pris_device_close,
33     .read = pris_device_read,
34     .write = pris_device_write
35 };
```

```
36
37
38 // Open function
39 static int pris_device_open(struct inode *inode, struct file *file) {
40     device_open_count++;
41     printk(KERN_INFO "Se abrió el dispositivo: %s\n", DEVICE_NAME);
42     return 0;
43 }
44
45 //	printk(KERN_INFO "Cha-la head-cha-la No importa lo que suceda Siempre el ánimo mantendré:
46 //	%s\n", device_open_count);
47 // Close function
48 static int pris_device_close(struct inode *inode, struct file *file) {
49     printk(KERN_INFO "Cerrando dispositivo soy feliz el dia de ho-ho-ho-hoy\n");
50     return 0;
51 }
52
53 static int __init pris_init(void) {
54     major_number = register_chrdev(0, DEVICE_NAME, &fops);
55     if (major_number < 0) {
56         printk(KERN_ALERT "simple_module: Failed to register a major number\n");
57         return major_number;
58     }
59
60     printk(KERN_INFO "Cargando Módulo pris!\n");
61     printk(KERN_INFO "I was assigned major number %d. To talk to\n", major_number);
62     printk(KERN_INFO "the driver, create a dev file with\n");
63     printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, major_number);
64     printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
65     printk(KERN_INFO "the device file.\n");
66     printk(KERN_INFO "Remove the device file and module when done.\n");
67     return 0;
68 }
69
70 static void __exit pris_exit(void) {
71     printk(KERN_INFO "Adiós vaqueros!!!\n");
```

CODIGO

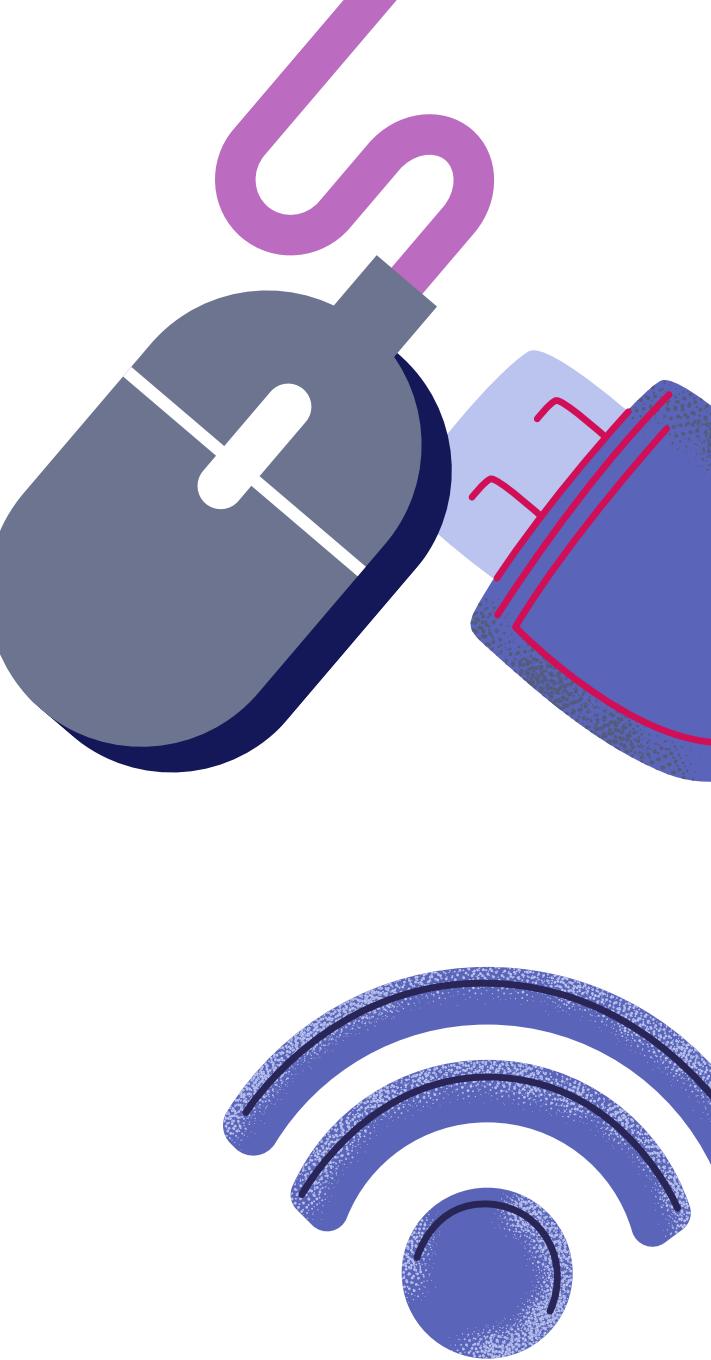
```
72 }
73
74
75 /* Device read function */
76 /*static ssize_t chirisco_device_read(struct file *filep, char *buffer, size_t len, loff_t *offset)
77 {
78     int bytes_read = 0;
79     if (*offset >= BUFFER_SIZE)
80         return 0;
81     if (len + *offset > BUFFER_SIZE)
82         len = BUFFER_SIZE - *offset;
83
84     if (copy_to_user(buffer, device_buffer + *offset, len) != 0) return -EFAULT;
85
86     *offset += len;
87     bytes_read = len;
88     printk(KERN_INFO "chardev: Sent %d bytes to the user\n", bytes_read);
89     return bytes_read;
90 }*/
91
92
93 static ssize_t pris_device_read(struct file *filep, char *buffer, size_t len, loff_t *offset) {
94     int errors = 0;
95     printk(KERN_INFO "Entrando a pris_device_read\n");
96     char *message = "Yo siempre estare feliz porque asi soy yo la-la-la-laaaaaa\n";
97     int message_len = strlen(message);
98
99     if (exit == 0) {
100         exit = 1;
101         errors = copy_to_user(buffer, message, message_len);
102     } else {
103         exit = 0;
104         return 0;
105     }
106
107     if (errors == 0) {
```

```
93 static ssize_t pris_device_read(struct file *filep, char *buffer, size_t len, loff_t *offset) {
105 }
106
107     if (errors == 0) {
108         return message_len;
109     } else {
110         exit = 0;
111         return -EFAULT;
112     }
113     //return errors == 0 ? message_len : -EFAULT;
114 }
115
116 /* Device write function */
117 static ssize_t pris_device_write(struct file *filep, const char *buffer, size_t len, loff_t
*offset) {
118     printk(KERN_INFO "Entrando a write\n");
119     int bytes_written = 0;
120     if (*offset >= BUFFER_SIZE) return 0;
121
122     if (len + *offset > BUFFER_SIZE) len = BUFFER_SIZE - *offset;
123
124     if (copy_from_user(device_buffer + *offset, buffer, len) != 0) return -EFAULT;
125
126     *offset += len;
127     bytes_written = len;
128     printk(KERN_INFO "chardev: Received %d bytes from the user\n", bytes_written);
129     printk(KERN_INFO "chardev: Received %s from the user\n", device_buffer);
130     printk(KERN_INFO "Saliendo a write\n");
131     return bytes_written;
132 }
133
134
135
136 /*static ssize_t chirisco_device_write(struct file *filep, const char *buffer, size_t len, loff_t
*offset) {
137     printk(KERN_INFO "Entrando a write\n");
138     printk(KERN_INFO "chardev: Received %s from the user\n", buffer);
139 }
```



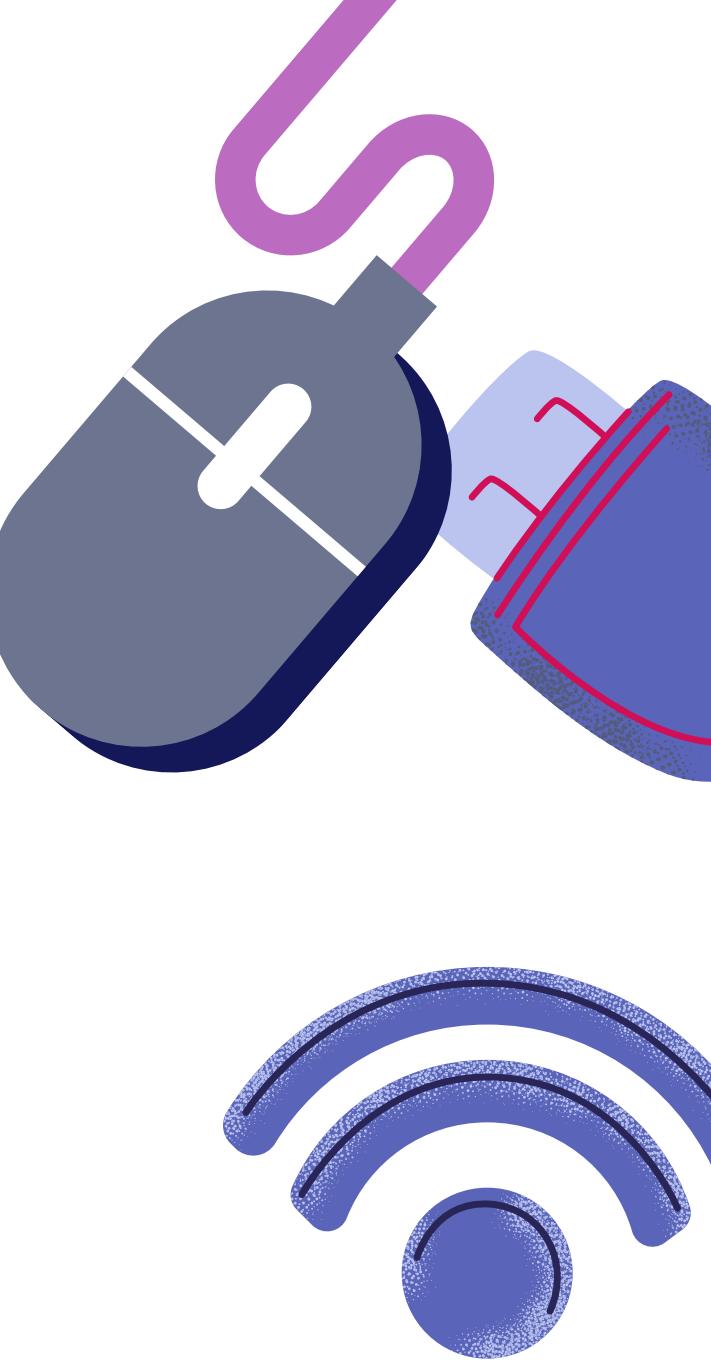
CODIGO

```
117 static ssize_t pris_device_write(struct file *filep, const char *buffer, size_t len, loff_t *offset...  
127     bytes_written = len;  
128     printk(KERN_INFO "chardev: Received %d bytes from the user\n", bytes_written);  
129     printk(KERN_INFO "chardev: Received %s from the user\n", device_buffer);  
130     printk(KERN_INFO "Saliendo a write\n");  
131     return bytes_written;  
132 }  
133  
134  
135  
136 /*static ssize_t chirisco_device_write(struct file *filep, const char *buffer, size_t len, loff_t  
   *offset) {  
137     printk(KERN_INFO "Entrando a write\n");  
138     printk(KERN_INFO "chardev: Received %s from the user\n", buffer);  
139     return 0;  
140 }*/  
141 module_init(pris_init);  
142 module_exit(pris_exit);  
143
```



Fucionamiento

- ESTE MODULO CREA UN DISPOSITIVO DE CARACTER EN LINUX QUE PERMITE REALIZAR OPERACIONES DE LECTURA Y ESCRITURA.
- AL ABRIR EL DISPOSITIVO, SE INCREMENTA UN CONTADOR QUE LLEVA EL REGISTRO DE LAS VECES QUE EL DISPOSITIVO HA SIDO ABIERTO.
- AL LEER, SE DEVUELVE UN MENSAJE FIJO. AL ESCRIBIR, SE ALMACENAN LOS DATOS EN UN BUFER.
- EL MODULO MANEJA LAS OPERACIONES BASICAS DE LOS DISPOSITIVOS DE CARACTER EN LINUX: OPEN, RELEASE, READ Y WRITE.



COMO SE USA EL MODULO

- **Cargar el modulo:** Se carga el modulo con el comando insmod.
- **Crear el dispositivo:** Usa el comando mknod para crear un archivo de dispositivo con el numero mayor asignado dinamicamente.
- **Interactuar con el dispositivo:** Usa cat para leer desde el dispositivo y usa echo para escribir en el dispositivo.
- **Desmontar el modulo:** Se desmonta con el comando rmmod y se elimina el archivo de dispositivo.



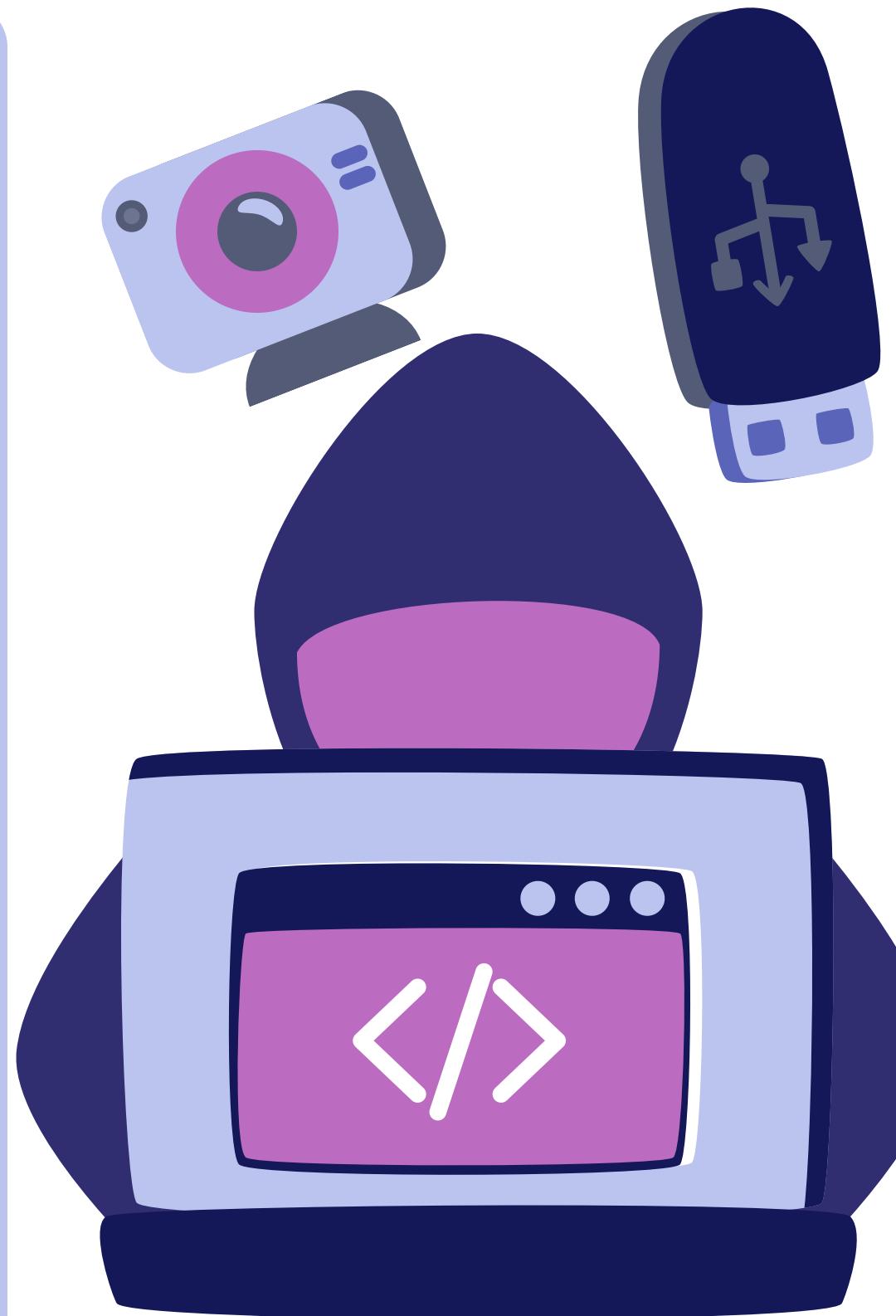
```
+ priscila@vbox:~/Escritorio/EXAMEN
~/Escritorio/EXAMEN
jun 04 11:19:38 vbox kernel: try various minor numbers. Try to cat and echo to
jun 04 11:19:38 vbox kernel: the device file.
jun 04 11:19:38 vbox kernel: Remove the device file and module when done.
^C
priscila@vbox:~/Documentos/EXAMEN$ sudo mknod /dev/pris c 508 0
[sudo] contraseña para priscila:
priscila@vbox:~/Documentos/EXAMEN$ cat /dev/pris
Yo siempre estare feliz porque asi soy yo la-la-la-laaaaaa
priscila@vbox:~/Documentos/EXAMEN$ echo "TEST" > /dev/pris
bash: /dev/pris: Permisos denegados
priscila@vbox:~/Documentos/EXAMEN$ sudo echo "TEST" > /dev/pris
bash: /dev/pris: Permisos denegados
priscila@vbox:~/Documentos/EXAMEN$ sudo chmod 666 /dev/pris
[sudo] contraseña para priscila:
priscila@vbox:~/Documentos/EXAMEN$ sudo echo "TEST" > /dev/pris
priscila@vbox:~/Documentos/EXAMEN$ cat /dev/pris
Yo siempre estare feliz porque asi soy yo la-la-la-laaaaaa
priscila@vbox:~/Documentos/EXAMEN$ echo "TEST" > /dev/pris
priscila@vbox:~/Documentos/EXAMEN$ sudo rmmod pris
priscila@vbox:~/Documentos/EXAMEN$ sudo dnf install kernel-devel kernel-headers
gcc make elfutils-libelf-devel
Actualizando y cargando repositorios:
Repositorios cargados.
Package "kernel-devel-6.14.8-300.fc42.aarch64" is already installed.
Package "kernel-devel-6.14.9-300.fc42.aarch64" is already installed.
```

CONCLUSION

El desarrollo del módulo de carácter "pris" demuestra de manera clara y práctica cómo se puede extender el funcionamiento del kernel de Linux mediante la creación de módulos personalizados. A través de este módulo, se exemplifican los principios fundamentales del manejo de dispositivos en modo carácter, incluyendo la asignación de números mayor y menor, el uso de funciones básicas de entrada/salida (open, read, write, release), y la comunicación entre el espacio de usuario y el espacio del kernel utilizando funciones seguras como `copy_to_user` y `copy_from_user`.

Este tipo de programación resulta esencial en el desarrollo de drivers personalizados, así como en sistemas embebidos o entornos donde se necesita un control directo sobre el hardware.

Además, el módulo refuerza buenas prácticas en la escritura de código para el kernel, como el uso de macros (`module_init` y `module_exit`), el manejo de búferes, y la generación de mensajes de diagnóstico mediante `printk`, lo cual es fundamental para depurar y entender el comportamiento del sistema a bajo nivel.



MUCHAS GRACIAS