Universidad Nacional de General Sarmiento

SISTEMAS OPERATIVOS Y REDES II

2024

TRABAJO PRÁCTICO 1

INFORME

INTEGRANTES

Farias, Federico Emanuel

PROFESORES

Chuquimango Chilon, Luis Benjamin

Echabarri, Alan Pablo Daniel

Introducción

En este trabajo práctico, se explora la creación y manejo de módulos en el kernel de Linux. Comenzamos con un módulo básico para entender el proceso de compilación y carga en el kernel, para luego crear un Char Device y sus respectivas operaciones. El trabajo se encuentra alojado en el siguiente repo: https://github.com/fridriik/SORII-TP1.

Módulo Hola Mundo

Los puntos 1 y 2 pertenecientes a esta sección se basan en la descarga del ejemplo del repositorio provisto y la verificación del funcionamiento del módulo de ejemplo para tener una introducción al tema.
Luego de hacer lo anterior se investigó los comandos para realizar la compilación y mostrar por consola el funcionamiento esperado.

Lo que hacemos para que funcione es realizar los siguientes pasos:

1. Colocar el archivo del código fuente del módulo junto con el archivo Makefile en el mismo directorio.
2. Abrir una terminal en ese directorio y ejecutar el comando “make” para compilar el módulo del kernel.
3. Cargar el módulo con el comando “sudo insmod modulo.ko”.
4. Verificar la carga del módulo con el comando “dmesg | tail”, siempre y cuando tengamos un print en la función init.
5. Descargar el módulo con el comando “sudo rmmod módulo”.
6. Verificar la descarga del módulo con el comando “dmesg | tail”, siempre y cuando tengamos un print en la función exit.
7. Utilizar el comando “make clean” para limpiar los archivos generados por el proceso de compilación.

```
alumno@alumno-virtualbox:~/TP1/SORII-TP1/helloworld$ make
make -C /lib/modules/5.4.0-174-generic/build M=/home/alumno/TP1/SORII-TP1/helloworld modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-174-generic'
  CC [M]  /home/alumno/TP1/SORII-TP1/helloworld/miModulo.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/alumno/TP1/SORII-TP1/helloworld/miModulo.mod.o
  LD [M]  /home/alumno/TP1/SORII-TP1/helloworld/miModulo.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-174-generic'
alumno@alumno-virtualbox:~/TP1/SORII-TP1/helloworld$ sudo insmod miModulo.ko
alumno@alumno-virtualbox:~/TP1/SORII-TP1/helloworld$ dmesg | tail
[  167.653022] UNGS : Driver registrado
alumno@alumno-virtualbox:~/TP1/SORII-TP1/helloworld$ sudo rmmod miModulo
alumno@alumno-virtualbox:~/TP1/SORII-TP1/helloworld$ dmesg | tail
[  167.653022] UNGS : Driver registrado
[  180.196542] UNGS : Driver desregistrado
alumno@alumno-virtualbox:~/TP1/SORII-TP1/helloworld$ make clean
make -C /lib/modules/5.4.0-174-generic/build M=/home/alumno/TP1/SORII-TP1/helloworld clean
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-174-generic'
  CLEAN   /home/alumno/TP1/SORII-TP1/helloworld/Module.symvers
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-174-generic'
alumno@alumno-virtualbox:~/TP1/SORII-TP1/helloworld$
```

Módulo Char Device

En esta sección se explicarán los pasos necesarios para crear, cargar y realizar diferentes operaciones sobre el módulo Char Device a construir.
Un Char Device funciona transfiriendo datos como un flujo bloque a bloque. A diferencia de un block device no poseen la operación seek y tienen asociado un archivo en /dev. También para que sea reconocido poseen un número identificatorio llamado Major (en el código fuente usaremos una variable con este nombre para que sea lo más declarativo posible).
Con este pequeño resumen del funcionamiento de un Char Device, se muestran los pasos realizados en la segunda parte del trabajo práctico.

Creando init y exit

Leyendo el libro *The Linux Kernel Module Programming Guide(2023)* provisto por los docentes ponemos énfasis en la sección 6 del mismo para definir las funciones necesarias que necesitamos.

Antes de empezar a definir necesitaremos crear los prototipos necesarios para las operaciones del Char Device. Estos prototipos sirven para que al compilar se puedan reconocer tanto la estructura de operaciones y sus posteriores definiciones.
Por un lado tenemos los prototipos de las operaciones

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char __user *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char __user *, size_t, loff_t *);
```

y luego la estructura de las operaciones.

```
static struct file_operations chardev_fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release,
};
```

Esta estructura en conjunto con el número Major hablado anteriormente y una estructura de clase se utilizarán para registrar el Char Device dentro de la función *init_module.*

Ahora comenzamos con la implementación de las funciones *init_module* y *cleanup_module,* refactorizado a *exit_module* en el archivo (se decide ir por este nombre para ser más explícito en su funcionalidad según el libro).
La primera, es responsable de establecer el entorno necesario para que el módulo funcione correctamente dentro del kernel. Esto incluye la asignación de recursos, como memoria o identificadores de dispositivos, y la configuración de cualquier estado inicial requerido.
Por otro lado, *cleanup_module,* se encarga de la limpieza una vez que el módulo ya no es necesario. Esta función revierte todas las acciones realizadas por *init_module*, liberando recursos y asegurando que el módulo pueda ser descargado sin dejar rastros en el sistema.

Definiendo device_open y device_release

Con los prototipos y la estructura de operaciones ya realizadas, comenzamos con la definición de las operaciones.
La función *device_open* se invoca cuando un proceso ejecuta una operación de apertura sobre el archivo del dispositivo correspondiente. Esta función es responsable de preparar el dispositivo para su uso. Por otro lado, *device_release* se llama cuando un proceso realiza una operación de cierre. Su propósito es liberar recursos previamente asignados y realizar cualquier limpieza necesaria, asegurando que el dispositivo se deje en un estado consistente para futuras operaciones.
Respecto a estas dos operaciones, es importante que tengamos como variable global lo siguiente para que no haya accesos múltiples al mismo Char Device.

```
static atomic_t already_open = ATOMIC_INIT(CDEV_NOT_USED);
```

- Atomic_t already_open: Variable que se utiliza para llevar un registro de si el dispositivo ya ha sido abierto. El uso de una variable atómica aquí es importante porque las operaciones de dispositivos pueden ser accedidas por múltiples procesos simultáneamente, y las operaciones atómicas garantizan que no se produzcan condiciones de carrera.
- ATOMIC_INIT(CDEV_NOT_USED): Inicializa la variable atómica con un valor dado. En este caso, CDEV_NOT_USED es una constante que indica que el dispositivo no está en uso.

Luego en *device_open* verificamos si el dispositivo ya está en uso, si no lo está, incrementa el contador de uso del módulo con try_module y permite la apertura del dispositivo. Por otro lado en *device_release,* se reinicia la variable atómica para indicar que el dispositivo ya no está en uso y decrementa el contador de uso del módulo con module_put.

Escritura y lectura

La función *device_write* es crucial en la interacción con Char Devices en Linux, permitiendo a los procesos escribir en el mismo. Esta operación es fundamental para la comunicación entre el espacio de usuario y el kernel, y es utilizada para enviar comandos o información al dispositivo. Es invocada por ejemplo cuando un proceso quiere escribir en el archivo dev
echo "tu mensaje" > /dev/dispositivo.

Primero definimos el tamaño que tendrá el mensaje

```
#define BUF_LEN 80
```

y luego el mensaje que dará el módulo cuando se lo pidamos

```
static char msg[BUF_LEN + 1];
```

Utilizamos *strncpy* para copiar el mensaje del espacio de usuario al búfer del kernel, nos aseguramos que el mensaje sea una cadena que termine en NULL y *printk* para registrar el mensaje en el log del kernel.

Luego necesitamos definir la operación *device_read.* Para ésto utilizamos una función auxiliar del kernel para facilitar la lectura de datos del búfer llamada *simple_read_from_buffer* que retorna la cantidad de bytes leídos, dependiendo de la cantidad de datos disponibles en el mensaje, y si se alcanza el final del mismo, la función retornará 0, indicando el fin del archivo (EOF).
Por último para mostrar el mensaje escrito por consola, primero damos permiso total con *sudo chmod 666 /dev/dispositivo* (en nuestro caso el Char Device). Luego como dijimos anteriormente escribimos un mensaje con el comando *echo* y finalmente con el comando *cat /dev/dispositivo* (en nuestro caso el Char Device) mostramos el mensaje.

```
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ make
make -C /lib/modules/5.4.0-174-generic/build M=/home/alumno/TP1/SORII-TP1/charde
v modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-174-generic'
  CC [M]  /home/alumno/TP1/SORII-TP1/chardev/chardev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/alumno/TP1/SORII-TP1/chardev/chardev.mod.o
  LD [M]  /home/alumno/TP1/SORII-TP1/chardev/chardev.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-174-generic'
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ sudo insmod chardev.ko
[sudo] password for alumno:
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ echo "hola" > /dev/chardev
bash: /dev/chardev: Permission denied
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ sudo echo "hola" > /dev/charde
v
bash: /dev/chardev: Permission denied
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ sudo chmod 666 /dev/chardev
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ echo "hola" > /dev/chardev
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ cat /dev/chardev
hola
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$
```

Lectura (al revés)

Los pasos a seguir son idénticos al anterior con la diferencia que ahora debemos modificar la función *device_read.* Además de retornar la cadena con la función *simple_read_from_buffer,* previamente, la vamos a invertir de la siguiente manera. Creando un array caracteres inicializado en 0. Luego definimos la longitud de *msg* y un índice para el bucle que utilizaremos para invertir la cadena, respectivamente. El resultado es el siguiente:

```
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ make
make -C /lib/modules/5.4.0-174-generic/build M=/home/alumno/TP1/SORII-TP1/chardev modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-174-generic'
  CC [M]  /home/alumno/TP1/SORII-TP1/chardev/chardev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/alumno/TP1/SORII-TP1/chardev/chardev.mod.o
  LD [M]  /home/alumno/TP1/SORII-TP1/chardev/chardev.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-174-generic'
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ sudo insmod chardev.ko
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ sudo chmod 666 /dev/chardev
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ echo "Hola" > /dev/chardev
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$ cat /dev/chardev
aloH
alumno@alumno-virtualbox:~/TP1/SORII-TP1/chardev$
```

Conclusiones

Siguiendo la guía del libro *The Linux Kernel Module Programming Guide*, este trabajo permitió aplicar lo aprendido en la creación y manejo de módulos del kernel de Linux, comprendiendo el proceso de compilación y manejo de dispositivos. También gracias a la guía se reforzó el entendimiento sobre operaciones atómicas y gestión de recursos en el kernel, preparando el camino para futuros desarrollos.