Práctica 2 - Módulos, Drivers y Syscalls

Requisitos >

Para realizar esta práctica puede utilizar exactamente la misma versión del código fuente de Linux utilizada en la práctica 1. Se puede usar la misma máquina virtual de la práctica 1 o una de su elección si resulta más cómodo (por ejemplo una VM con interfaz gráfica y un IDE).

Si se usa la misma VM de la práctica 1 este directorio es `/home/so/kernel/linux-<version>/` .

https://gitlab.com/unlp-so/codigo-para-practicas/-/tree/main/practica2

Materiales de Referencia >

Ref 1: https://www.kernel.org/doc/html/latest/process/adding-syscalls.html
Ref 2 "Linux Kernel Hacking: A Crash Course - Speaker Deck":
https://speakerdeck.com/georgiknox/linux-kernel-hacking-a-crash-course

System Calls

Conceptos Generales

1. ¿Qué es una System Call? ¿Para qué se utiliza?

Una System Call (o llamada al sistema) es el mecanismo utilizado por un proceso que se ejecuta en modo usuario para solicitar un servicio al Sistema Operativo (SO).

Las System Call se utilizan para que un proceso de usuario pueda acceder a funciones o servicios protegidos y gestionados por el Sistema Operativo, como la lectura o escritura de archivos (que implica acceder al hardware), la creación de nuevos procesos, o la comunicación con dispositivos. El SO actúa como un servidor que atiende estas solicitudes.

En sistemas Unix, como GNU/Linux, la interfaz de programación (API) principal que los procesos utilizan para invocar System Call es la biblioteca libc.

Para realizar una System Call, el proceso de usuario debe indicar el número de la System Call que desea ejecutar y los parámetros necesarios. Luego, se emite una interrupción por software

(TRAP) para cambiar del modo usuario al modo kernel y pasar el control al SO. El SO verifica el número de la System Call y ejecuta el código correspondiente en modo kernel, en el contexto del proceso que la invocó.

2. ¿Para qué sirve la macro syscall? Describa el propósito de cada uno de sus parámetros.
   Ayuda: http://www.gnu.org/software/libc/manual/html_mono/libc.html#System-Calls

La macro `syscall` en Linux se utiliza para invocar llamadas al sistema directamente desde programas en C sin necesidad de utilizar las funciones de la biblioteca estándar (`libc`). Su propósito es proporcionar una interfaz de bajo nivel para interactuar con el kernel del sistema operativo.

Sintaxis de syscall:

```
#include <unistd.h>
#include <sys/syscall.h>

long syscall(long number, ...);
```

Parámetros:

- `number` *(long int):* Es el número de la llamada al sistema que se desea invocar. Cada syscall tiene un número único definido en los archivos de encabezado del sistema, generalmente en `/usr/include/asm/unistd.h` o `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`.
- *Argumentos adicionales (* `...` *):* Son los parámetros específicos que requiere la llamada al sistema. El número y tipo de estos parámetros dependen de la syscall específica que se está utilizando. Por ejemplo:
  - `syscall(SYS_write, fd, buffer, size);` (Corresponde a `write(fd, buffer, size)`).
  - `syscall(SYS_getpid);` (Obtiene el ID del proceso actual, similar a `getpid()`).

3. Ejecute el siguiente comando e identifique el propósito de cada uno de los archivos que encuentra `ls -lh /boot | grep vmlinuz`

Salida del Comando:

```
so@so:~$ ls -lh /boot | grep vmlinuz
-rw-r--r-- 1 root root 7,9M ene  2 10:31 vmlinuz-6.1.0-29-amd64
-rw-r--r-- 1 root root 7,9M feb  7 06:43 vmlinuz-6.1.0-31-amd64
-rw-r--r-- 1 root root 8,3M mar 24 11:59 vmlinuz-6.13.7
```

Propósito de los Archivos:

- `vmlinuz-6.1.0-29-amd64`: Es un Kernel comprimido versión 6.1.0-29 para arquitectura amd64 (64 bits). Por lo que se ve fue instalado el 2 de enero. Puede ser un kernel anterior que el sistema aún conserva.
- `vmlinuz-6.1.0-31-amd64`: Es un Kernel comprimido versión 6.1.0-31, instalado el 7 de febrero. Puede ser una actualización del anterior.
- `vmlinuz-6.13.7`: Es el Kernel comprimido que compilamos en la práctica 1.

4. Acceda al código fuente de GNU Linux, sea visitando https://kernel.org/ o bien trayendo el código del kernel (cuidado, como todo software monolítico son unos cuantos gigas) `git clone https://github.com/torvalds/linux.git`
5. ¿Para qué sirven el siguiente archivo?
    1. `arch/x86/entry/syscalls/syscall_64.tbl`

El archivo `arch/x86/entry/syscalls/syscall_64.tbl` contiene la tabla de llamadas al sistema (syscalls) para la arquitectura x86_64. Este archivo asigna un número de syscall a cada llamada del sistema en la arquitectura x86_64. Es fundamental para que el kernel pueda despachar correctamente las llamadas al sistema cuando un programa las invoca utilizando la instrucción `syscall` de la CPU.

6. ¿Para qué sirve la herramienta `strace`? ¿Cómo se usa?

La herramienta `strace` se utiliza para monitorear, registrar y analizar las llamadas al sistema (`syscalls`) realizadas por un proceso en sistemas Unix/Linux. Es especialmente útil para depuración, diagnóstico de errores y análisis de rendimiento.

Para usarla podemos hacerlo de la siguiente forma: `$ strace ./a.out`. En este caso el comando va a ejecutar `a.out` y mostrará todas las syscalls que realiza en tiempo real. Para ver solo las syscalls podríamos ejecutar `$ strace ./a.out > /dev/null`

Opciones útiles de `strace`

- *Mostrar solo ciertas syscalls:* Si solo quieres ver llamadas relacionadas con archivos, puedes filtrar: `strace -e open,read,write,close ./a.out`. Esto mostrará solo las llamadas `open`, `read`, `write` y `close`.
- *Adjuntar `strace` a un proceso en ejecución:* Para analizar un proceso ya en ejecución, usamos su PID: `strace -p <PID>`. Esto mostrará las syscalls en vivo del proceso identificado por `<PID>`.
- *Guardar la salida en un archivo:* Puedes registrar las syscalls en un archivo para analizarlas después: `strace -o log.txt ./a.out`. Esto guarda la salida en `log.txt` en lugar de imprimirla en pantalla.
- *Contar la frecuencia de cada syscall:* Para ver un resumen con el número de veces que se ejecutó cada syscall: `strace -c ./a.out`. Esto es útil para detectar cuellos de botella en el

rendimiento.

7. ¿Para qué sirve la herramienta `ausyscall`? ¿Cómo se usa?

La herramienta `ausyscall` forma parte del paquete Audit de Linux y se utiliza para listar y traducir los números de las syscalls a sus nombres y viceversa en un sistema Linux.

¿Cómo se usa `ausyscall`?

1. *Listar todas las syscalls disponibles en la arquitectura actual:* `ausyscall --dump`. Esto imprimirá una lista de todas las llamadas al sistema junto con sus números.
2. *Obtener el número de una syscall por su nombre:* `ausyscall --exact open`. Si ejecutas esto, te devolverá el número de la llamada al sistema `open`.
3. *Obtener el nombre de una syscall a partir de su número:* `ausyscall 2`. Si ejecutas esto, devolverá el nombre de la syscall asociada al número `2` (que en muchas arquitecturas es `open`).
4. *Listar syscalls para una arquitectura específica:* `ausyscall --dump --arch x86_64`. Esto mostrará solo las syscalls correspondientes a la arquitectura `x86_64`.

Práctica Guiada

Información >

La System Calls que vamos a implementar accederán a la estructura `task_struct` (https://alex-xjk.github.io/post/taskstruct/) que representa cada proceso en el sistema. Ha evolucionado con el tiempo, pero en las versiones más recientes del kernel (6.x), sigue teniendo los mismos principios básicos con nuevas adiciones y modificaciones. Es la estructura utilizada por el `scheduler` (https://docs.kernel.org/scheduler/sched-eevdf.html) para planificar las tareas del Sistema Operativo.

Estas estructuras junto a otras conforman lo que en los libros de Sistemas Operativos se denomina la PCB(Process Control Block).

Accederemos con nuestra llamada al sistema a algunos datos almacenados en los de la estructura `task_struct`.

Para ello modificaremos los siguientes archivos del código fuente del Kernel para declarar nuestras system calls

- `arch/arm64/include/asm/unistd.h`
- `arch/x86/entry/syscalls/syscall_64.tbl`

- `include/uapi/asm-generic/unistd.h`

Y además agregaremos estos dos nuevos archivos dónde colocaremos la implementación de nuestras system call

- `kernel/Makefile`
- `kernel/my_sys_call.c`

Agregamos una nueva System Call

1. Añadiremos el siguiente archivo con el código de nuestra system call: `kernel/my_sys_call.c`

```
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/sched/signal.h>
#include <linux/slab.h> // Para kmalloc y kfree

SYSCALL_DEFINE1 (my_sys_call, int, arg) {
    printk(KERN_INFO "My syscall called with arg: %d\n", arg);
    return 0;
}

SYSCALL_DEFINE2(get_task_info, char __user *, buffer, size_t, length) {
    struct task_struct *task; char kbuffer[1024]; // Buffer en el espacio
del kernel
    int offset = 0;
    for_each_process(task) {
        offset += snprintf(kbuffer + offset, sizeof(kbuffer) - offset, "PID:
%d | Nombre: %s | Estado: %d \n",task->pid, task->comm,
task_state_index(task));
        if (offset >= sizeof(kbuffer)) // Evita sobrepasar el tamaño del
buffer
            break;
        printk(KERN_INFO "PID: %d | Nombre: %s\n", task->pid, task->comm);
    }
    // Copia la información al espacio de usuario
    if (copy_to_user(buffer, kbuffer, min(length, (size_t)offset)))
        return -EFAULT;
    return min(length, (size_t)offset);
}
```

```c
SYSCALL_DEFINE2(get_threads_info, char __user *, buffer, size_t, length) {
    struct task_struct *task, *thread;
    char *kbuffer;
    int offset = 0;
    // Asignar memoria dinámica para el buffer
    kbuffer = kmalloc(2048, GFP_KERNEL);
    if (!kbuffer)
        return -ENOMEM;
    for_each_process(task) {
        offset += snprintf(kbuffer + offset, 2048 - offset, "Proceso: %s
(PID: %d)\n", task->comm, task->pid);
        for_each_thread(task, thread) {
            offset += snprintf(kbuffer + offset, 2048 - offset," ├── Hilo:
%s (TID: %d)\n", thread->comm, thread->pid);
            if (offset ⩾ 2048)
                break;
        }
        if (offset ⩾ 2048)
            break;
    }
    if (copy_to_user(buffer, kbuffer, min(length, (size_t)offset))) {
        kfree(kbuffer);
        return -EFAULT;
    }
    kfree(kbuffer);
    return min(length, (size_t)offset);
}
```

- Mirando el código anterior, investigue y responda lo siguiente?
  - ¿Para qué sirven los macros SYS_CALL_DEFINE?
  - ¿Para que se utilizan la macros for_each_process y for_each_thread?
  - ¿Para que se utiliza la función copy_to_user?
  - ¿Para qué se utiliza la función printk?, ¿porque no la típica printf?
  - Podría explicar que hacen las sytem call que hemos incluido?

¿Para qué sirven los macros SYS_CALL_DEFINE?

Los macros `SYSCALL_DEFINE` se utilizan para definir llamadas al sistema (syscalls) en el kernel de Linux. Existen versiones que admiten diferentes números de argumentos. Cuando una syscall es definida con este macro, se registra automáticamente en la tabla de syscalls del kernel.

¿Para que se utilizan la macros for_each_process y for_each_thread?

- `for_each_process(task)` recorre todos los procesos en ejecución en el sistema.

    - La variable `task` es un puntero a una estructura `task_struct`, que representa un proceso en el kernel.
    - Se usa para iterar sobre cada proceso activo.
- `for_each_thread(task, thread)` se usa dentro de un contexto en el que ya se está recorriendo un proceso con `for_each_process`.
    - `task`: proceso principal.
    - `thread`: variable que iterará sobre los hilos del proceso `task`.
    - Se usa para recorrer todos los hilos de un proceso.

¿Para que se utiliza la función copy_to_user?

La función `copy_to_user(destino, fuente, tamaño)` se usa para copiar datos del espacio de memoria del kernel al espacio de usuario. Es necesaria ya que el kernel y los procesos de usuario operan en espacios de memoria separados. Acceder directamente a la memoria del usuario desde el kernel puede causar errores o vulnerabilidades. `copy_to_user` se encarga de realizar esta copia de manera segura, evitando accesos indebidos.

¿Para qué se utiliza la función printk?, ¿porque no la típica printf?

`printk` es la versión de `printf` en el kernel y se usa para registrar mensajes en el buffer de logs del sistema. `printf` no se usa ya que es una unción del espacio de usuario, mientras que `printk` es específica del kernel. En el kernel, no hay acceso directo a `stdout` o `stderr` como en los programas de usuario.

Podría explicar que hacen las sytem call que hemos incluido?

El código define tres syscalls personalizadas:

- `SYSCALL_DEFINE1(my_sys_call, int, arg)`
    - Recibe un entero como argumento.
    - Imprime el argumento en los logs del kernel.
    - No realiza ninguna acción más y retorna `0`.
    - Propósito: solo sirve como prueba para ver que la syscall es invocada correctamente.
- `SYSCALL_DEFINE2(get_task_info, char __user *, buffer, size_t, length)`
    - Itera sobre todos los procesos del sistema y obtiene su:
        - PID (Process ID).
        - Nombre.
        - Estado actual.
    - Copia esta información al `buffer` en el espacio de usuario mediante `copy_to_user`.
    - Propósito: permite a un programa en espacio de usuario obtener una lista de procesos en ejecución.

- `SYSCALL_DEFINE2(get_threads_info, char __user *, buffer, size_t, length)`
  - Recorre todos los procesos y luego lista sus hilos usando `for_each_thread`.
  - Copia la información al espacio de usuario.
  - Usa `kmalloc` para asignar memoria dinámica y `kfree` para liberarla.
  - Propósito: permite obtener información detallada sobre procesos y sus hilos en ejecución.

2. Modificaremos uno de los archivos Makefile del código del Kernel para indicar la compilación de nuestro código agregado en el paso anterior: `kernel/Makefile`

```
obj-y = fork.o exec_domain.o panic.o \
    cpu.o exit.o softirq.o resource.o \
    sysctl.o capability.o ptrace.o user.o \
    signal.o sys.o umh.o workqueue.o pid.o task_work.o \
    extable.o params.o \
    kthread.o sys_ni.o nsproxy.o \
    notifier.o ksysfs.o cred.o reboot.o \
    async.o range.o smpboot.o ucount.o regset.o \
    my_sys_call.o
```

3. Añadir una entrada al final de la tabla que contiene todas las System Calls, la syscall table. En nuestro caso, vamos a dar soporte para nuestra syscall a la arquitectura x86_64. *Atención:*
   - El archivo donde añadiremos la entrada para la system call está estructurado en columnas de la siguiente forma: `<number> <abi> <name> <entry point>`.
   - Buscaremos la última entrada cuya ABI sea “common” y luego agregaremos una línea para nuestra system call.
   - Debemos asignar un número único a nuestra system call, de modo que aumentaremos en 1 el número de la última.

```
444 common landlock_create_ruleset sys_landlock_create_ruleset
445 common landlock_add_rule sys_landlock_add_rule
446 common landlock_restrict_self sys_landlock_restrict_self
447 common memfd_secret sys_memfd_secret
448 common process_mrelease sys_process_mrelease
449 common futex_waitv sys_futex_waitv
450 common set_mempolicy_home_node sys_set_mempolicy_home_node
451 common my_sys_call sys_my_sys_call
452 common get_task_info sys_get_task_info
453 common get_threads_info sys_get_threads_info
```

En mi caso quedó así:

```
467 common my_sys_call sys_my_sys_call
468 common get_task_info sys_get_task_info
469 common get_threads_info sys_get_threads_info
```

Ahora incluimos la declaración de nuestras system calls en los headers del kernel junto a las otras system calls. Es importante recordar que debemos aumentar el valor de `__NR_syscalls` de acuerdo a la cantidad de system calls que hemos agregado, ya que este es el tamaño de un array interno dónde están los punteros a los manejadores de las system calls.
`include/uapi/asm-generic/unistd.h`

```
#define __NR_set_mempolicy_home_node 450
__SYSCALL(__NR_set_mempolicy_home_node, sys_set_mempolicy_home_node)

#define __NR_my_sys_call 451
__SYSCALL(__NR_my_sys_call, sys_my_sys_call)

#define __NR_get_task_info 452
__SYSCALL(__NR_get_task_info, sys_get_task_info)

#define __NR_get_threads_info 453
__SYSCALL(__NR_get_threads_info, sys_get_threads_info)

#undef __NR_syscalls
#define __NR_syscalls 454
```

En mi caso quedo así:

```
#define __NR_my_sys_call 467
__SYSCALL(__NR_my_sys_call, sys_my_sys_call)

#define __NR_get_task_info 468
__SYSCALL(__NR_get_task_info, sys_get_task_info)

#define __NR_get_threads_info 469
__SYSCALL(__NR_get_threads_info, sys_get_threads_info)

#undef __NR_syscalls
#define __NR_syscalls 470
```

4. Lo próximo que debemos realizar es compilar el Kernel con nuestros cambios. Una vez seguidos todos los pasos de la compilación como lo vimos en el trabajo práctico 1, acomodamos la imagen generada y arrancamos el sistema con el nuevo kernel.

> **Nota** >
>
> Odio con todo mi ser tener que compilar todo devuelta :)
>
> Igualmente es ejecutar los pasos de la Práctica 1 desde el comando `make -jX`

5. Ahora vamos a verificar que nuestras system calls nuevas ya son parte del kernel, para esto ejecutamos: `$ grep get_task_info "/boot/System.map-$(uname -r)"`. Aquí deberíamos ver el mapa de símbolos correspondiente a nuestra system call en el System.map del Kernel recientemente compilado.

Salida del Comando:

```
so@so:/$ grep get_task_info "/boot/System.map-$(uname -r)"
ffffffff810fc310 t __pfx___do_sys_get_task_info
ffffffff810fc320 t __do_sys_get_task_info
ffffffff810fc530 T __pfx___x64_sys_get_task_info
ffffffff810fc540 T __x64_sys_get_task_info
ffffffff810fc560 T __pfx___ia32_sys_get_task_info
ffffffff810fc570 T __ia32_sys_get_task_info
ffffffff82653dc0 d event_exit__get_task_info
ffffffff82653e40 d event_enter__get_task_info
ffffffff82653ec0 d __syscall_meta__get_task_info
ffffffff82653f00 d args__get_task_info
ffffffff82653f10 d types__get_task_info
ffffffff82f3ab40 d __event_exit__get_task_info
ffffffff82f3ab48 d __event_enter__get_task_info
ffffffff82f3f208 d __p_syscall_meta__get_task_info
```

6. Nuestro último paso es realizar un programa que llame a la System Call.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <string.h>

#define SYS_get_task_info 452

void print_task_info(const char *info) {
    printf("\nInformación de los procesos en ejecución:\n");
    printf("----------------------------------------\n");
    printf("%s", info);
    printf("\n----------------------------------------\n");
```

```
}

int main() {
    char buffer[1024]; // Buffer donde se almacenará la información de las
tareas
    long bytes_read;
    // Llamada al sistema para obtener la información de los procesos
    bytes_read = syscall(SYS_get_task_info, buffer, sizeof(buffer));
    // Comprobamos si la llamada al sistema fue exitosa
    if (bytes_read < 0) {
        perror("Error al invocar la llamada al sistema");
        return 1;
    }
    // Mostrar la información obtenida de los procesos
    print_task_info(buffer);
    return 0;
}
```

> **Nota**
>
> Cuando utilizamos llamadas al sistema, por ejemplo open() que permite abrir un archivo, no es necesario invocarlas de manera explícita, ya que por defecto la librería `libc` tiene funciones que encapsulan las llamadas al sistema.

Luego lo compilamos para obtener nuestro programa. Para ello ejecutamos: `$ gcc -o get_task_info get_task_info.c`

Por último nos queda ejecutar nuestro programa y ver el resultado. `$ ./get_task_info`

> **Cambios en el código que nos proveen**
>
> En el código nos dejan esta línea `#define SYS_get_task_info 452` pero nosotros tenemos que cambiarla para que quede con el número de syscall que definimos en la tabla, en mi caso queda `#define SYS_get_task_info 468`

Salida post ejecución del programa:

```
so@so:~/practica2$ ./get_task_info

Información de los procesos en ejecución:
----------------------------------------
PID: 1 | Nombre: systemd | Estado: 1
```

```
PID: 2 | Nombre: kthreadd | Estado: 1
PID: 3 | Nombre: pool_workqueue_ | Estado: 1
PID: 4 | Nombre: kworker/R-rcu_g | Estado: 8
PID: 5 | Nombre: kworker/R-sync_ | Estado: 8
PID: 6 | Nombre: kworker/R-slub_ | Estado: 8
PID: 7 | Nombre: kworker/R-netns | Estado: 8
PID: 9 | Nombre: kworker/0:1 | Estado: 8
PID: 10 | Nombre: kworker/0:0H | Estado: 8
PID: 11 | Nombre: kworker/u16:0 | Estado: 8
PID: 12 | Nombre: kworker/R-mm_pe | Estado: 8
PID: 13 | Nombre: rcu_tasks_kthre | Estado: 8
PID: 14 | Nombre: rcu_tasks_rude_ | Estado: 8
PID: 15 | Nombre: rcu_tasks_trace | Estado: 8
PID: 16 | Nombre: ksoftirqd/0 | Estado: 1
PID: 17 | Nombre: rcu_preempt | Estado: 8
PID: 18 | Nombre: rcu_exp_par_gp_ | Estado: 1
PID: 19 | Nombre: rcu_exp_gp_kthr | Estado: 1
PID: 20 | Nombre: migration/0 | Estado: 1
PID: 21 | Nombre: idle_inject/0 | Estado: 1
PID: 22 | Nombre: cpuhp/0 | Estado: 1
PID: 23 | Nombre: cpuhp/1 | Estado: 1
PID: 24 | Nombre: idle_inject/1 | Estado: 1
PID: 2
---------------------------------------
```

Con lo visto en la Práctica 1 sobre Makefiles, construya un Makefile de manera que si ejecuto:

- *make*, nuestro programa se compila get_task_info.c
- *make clean*, limpia el ejecutable y el código objeto generado
- *make run*, ejecuta el programa

El Makefile quedaría así:

```
# Nombre del ejecutable
TARGET = get_task_info

# Archivos fuente
SRC = get_task_info.c

# Compilador y flags
CC = gcc
CFLAGS = -Wall -Wextra -O2

# Reglas
all: $(TARGET)
```

```
$(TARGET): $(SRC)
    $(CC) $(CFLAGS) -o $(TARGET) $(SRC)

run: $(TARGET)
    ./$(TARGET)

clean:
    rm -f $(TARGET) *.o
```

Explicación del contenido del mismo:

- *Variables:*
  - `TARGET`: Variable que guarda el nombre del archivo que querés generar al compilar.
  - `SRC`: Variable que guarda el nombre del archivo fuente en C. Si tuvieras varios `.c`, los ponés separados por espacio (ej: `main.c helper.c`).
  - `CC`: Es el compilador que vamos a usar (por defecto `gcc` para C).
  - `CFLAGS`: Son los flags que le pasamos al compilador:
    - `-Wall`: Muestra todas las advertencias comunes.
    - `-Wextra`: Muestra todas las advertencias adicionales.
    - `-O2`: Optimización de nivel 2 (más rápido sin perder seguridad).
- *Regla principal `make` o `make all`:*
  - `all $(TARGET)`: Regla por defecto que se ejecuta si solo escribimos `make`. Le decís que, para `make`, debe construir lo que esté en `$(TARGET)` (que es `get_task_info`).
- *Compilar el archivo fuente:*
  - Las 2 líneas que están debajo de la regla principal hacen la regla de compilación que dice "Para construir `get_task_info`, necesito `get_task_info.c`". El comando abajo usa `gcc` (`$(CC)`), con flags (`$(CFLAGS)`), para generar el ejecutable `get_task_info` a partir de `get_task_info.c`.
- *Regla `make run`:*
  - La regla `run: $(TARGET)` hace correr el programa, se asegura que esté compilado y luego lo ejecuta.
- *Regla `make clean`:*
  - Elimina el ejecutable (`get_task_info`) y cualquier archivo `.o` (objeto intermedio), el parámetro `-f` evita errores si el archivo no existe.

Prueba de uso del Makefile:

```
so@so:~/practica2$ ls
get_task_info.c  Makefile
so@so:~/practica2$ make
```

```
gcc -Wall -Wextra -O2 -o get_task_info get_task_info.c
so@so:~/practica2$ ls
get_task_info  get_task_info.c  Makefile
so@so:~/practica2$ make run
./get_task_info

Información de los procesos en ejecución:
---------------------------------------
PID: 1 | Nombre: systemd | Estado: 1
PID: 2 | Nombre: kthreadd | Estado: 1
PID: 3 | Nombre: pool_workqueue_ | Estado: 1
PID: 4 | Nombre: kworker/R-rcu_g | Estado: 8
PID: 5 | Nombre: kworker/R-sync_ | Estado: 8
PID: 6 | Nombre: kworker/R-slub_ | Estado: 8
PID: 7 | Nombre: kworker/R-netns | Estado: 8
PID: 9 | Nombre: kworker/0:1 | Estado: 8
PID: 10 | Nombre: kworker/0:0H | Estado: 8
PID: 11 | Nombre: kworker/u16:0 | Estado: 8
PID: 12 | Nombre: kworker/R-mm_pe | Estado: 8
PID: 13 | Nombre: rcu_tasks_kthre | Estado: 8
PID: 14 | Nombre: rcu_tasks_rude_ | Estado: 8
PID: 15 | Nombre: rcu_tasks_trace | Estado: 8
PID: 16 | Nombre: ksoftirqd/0 | Estado: 1
PID: 17 | Nombre: rcu_preempt | Estado: 8
PID: 18 | Nombre: rcu_exp_par_gp_ | Estado: 1
PID: 19 | Nombre: rcu_exp_gp_kthr | Estado: 1
PID: 20 | Nombre: migration/0 | Estado: 1
PID: 21 | Nombre: idle_inject/0 | Estado: 1
PID: 22 | Nombre: cpuhp/0 | Estado: 1
PID: 23 | Nombre: cpuhp/1 | Estado: 1
PID: 24 | Nombre: idle_inject/1 | Estado: 1
PID: 2
---------------------------------------
so@so:~/practica2$ make clean
rm -f get_task_info *.o
so@so:~/practica2$ ls
get_task_info.c  Makefile
```

Monitoreando System Calls

1. Ejecute el programa anteriormente compilado `$ ./get_task_info`. ¿Cuál es el output del programa?

Salida post ejecución del programa:

```
so@so:~/practica2$ ./get_task_info

Información de los procesos en ejecución:
---------------------------------------
PID: 1 | Nombre: systemd | Estado: 1
PID: 2 | Nombre: kthreadd | Estado: 1
PID: 3 | Nombre: pool_workqueue_ | Estado: 1
PID: 4 | Nombre: kworker/R-rcu_g | Estado: 8
PID: 5 | Nombre: kworker/R-sync_ | Estado: 8
PID: 6 | Nombre: kworker/R-slub_ | Estado: 8
PID: 7 | Nombre: kworker/R-netns | Estado: 8
PID: 9 | Nombre: kworker/0:1 | Estado: 8
PID: 10 | Nombre: kworker/0:0H | Estado: 8
PID: 11 | Nombre: kworker/u16:0 | Estado: 8
PID: 12 | Nombre: kworker/R-mm_pe | Estado: 8
PID: 13 | Nombre: rcu_tasks_kthre | Estado: 8
PID: 14 | Nombre: rcu_tasks_rude_ | Estado: 8
PID: 15 | Nombre: rcu_tasks_trace | Estado: 8
PID: 16 | Nombre: ksoftirqd/0 | Estado: 1
PID: 17 | Nombre: rcu_preempt | Estado: 8
PID: 18 | Nombre: rcu_exp_par_gp_ | Estado: 1
PID: 19 | Nombre: rcu_exp_gp_kthr | Estado: 1
PID: 20 | Nombre: migration/0 | Estado: 1
PID: 21 | Nombre: idle_inject/0 | Estado: 1
PID: 22 | Nombre: cpuhp/0 | Estado: 1
PID: 23 | Nombre: cpuhp/1 | Estado: 1
PID: 24 | Nombre: idle_inject/1 | Estado: 1
PID: 2
---------------------------------------
```

2. Luego de ejecutar el programa ahora ejecute `$ sudo dmesg`. ¿Cuál es el output? porque? (recuerde printk y lea el man de dmesg).

Salida del comando:

```
root@so:/home/so/practica2# dmesg
[  306.161666] PID: 1 | Nombre: systemd
[  306.161675] PID: 2 | Nombre: kthreadd
[  306.161678] PID: 3 | Nombre: pool_workqueue_
[  306.161681] PID: 4 | Nombre: kworker/R-rcu_g
[  306.161683] PID: 5 | Nombre: kworker/R-sync_
[  306.161685] PID: 6 | Nombre: kworker/R-slub_
[  306.161687] PID: 7 | Nombre: kworker/R-netns
```

```
[  306.161689] PID: 8 | Nombre: kworker/0:0
[  306.161691] PID: 9 | Nombre: kworker/0:1
[  306.161694] PID: 10 | Nombre: kworker/0:0H
[  306.161696] PID: 11 | Nombre: kworker/u16:0
[  306.161699] PID: 12 | Nombre: kworker/R-mm_pe
[  306.161701] PID: 13 | Nombre: rcu_tasks_kthre
[  306.161757] PID: 14 | Nombre: rcu_tasks_rude_
[  306.161761] PID: 15 | Nombre: rcu_tasks_trace
[  306.161764] PID: 16 | Nombre: ksoftirqd/0
[  306.161766] PID: 17 | Nombre: rcu_preempt
[  306.161768] PID: 18 | Nombre: rcu_exp_par_gp_
[  306.161769] PID: 19 | Nombre: rcu_exp_gp_kthr
[  306.161772] PID: 20 | Nombre: migration/0
[  306.161774] PID: 21 | Nombre: idle_inject/0
[  306.161776] PID: 22 | Nombre: cpuhp/0
[  306.161778] PID: 23 | Nombre: cpuhp/1
```

El output es ese porque `printk()` es una función que el kernel de Linux usa para mostrar mensajes de depuración o estado en su buffer de logs. A diferencia de `printf()`, que imprime en la consola de usuario, `printk()` imprime en los logs del kernel que luego pueden ser consultados por ejemplo con el comando `dmesg` como hicimos.

> **Tip**
>
> `dmesg` nos va a mostrar todos los mensajes del kernel almacenados en su buffer de logs desde el arranque del mismo, si queremos ver solo lo relacionado a la salida del programa podemos ejecutar primero `dmesg -C` para borrar el contenido del buffer y limpiarlo, luego ejecutamos el programa y después ejecutamos `dmesg` para que solo muestre los mensajes del kernel relacionados al programa que hicimos.

3. Ejecute el programa anteriormente compilado con la herramienta strace `$ strace get_task_info`. *Aclaración: Si el programa strace no está instalado, puede instalarlo en distribuciones basadas en Debian con:* `$ sudo apt-get install strace`. En alguna parte del log de strace debería ver algo similar a lo siguiente: `syscall_0x1c4(0xffffdf859ba0, 0xffff9cc22078) 0x400, 0xaaaabe110740, 0xffff9cc790c0, 0xbd2cc5d5aef6ff14, 0xffff9cc22078) = 0x400`. Si luego ejecuto `# echo $((0x1C4))`
   - ¿Qué valor obtengo? porque?

Resultado de ejecutar `strace ./get_task_info`:

```
root@so:/home/so/practica2# strace ./get_task_info
execve("./get_task_info", ["./get_task_info"], 0x7ffc3f46a3c0 /* 29 vars */)
```

```
= 0
brk(NULL)                               = 0x5593a65ab000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f2788d09000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o
el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=22426, ...}, AT_EMPTY_PATH)
= 0
mmap(NULL, 22426, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2788d03000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"...,
832) = 832
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...},
AT_EMPTY_PATH) = 0
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64)
= 784
mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f2788b22000
mmap(0x7f2788b48000, 1396736, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f2788b48000
mmap(0x7f2788c9d000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x17b000) = 0x7f2788c9d000
mmap(0x7f2788cf0000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7f2788cf0000
mmap(0x7f2788cf6000, 53072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f2788cf6000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f2788b1f000
arch_prctl(ARCH_SET_FS, 0x7f2788b1f740) = 0
set_tid_address(0x7f2788b1fa10)         = 1927
set_robust_list(0x7f2788b1fa20, 24)     = 0
rseq(0x7f2788b20060, 0x20, 0, 0x53053053) = 0
mprotect(0x7f2788cf0000, 16384, PROT_READ) = 0
mprotect(0x5593695f4000, 4096, PROT_READ) = 0
mprotect(0x7f2788d41000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f2788d03000, 22426)           = 0
syscall_0x1d4(0x7ffd08b61730, 0x400, 0x5593695f4dd8, 0, 0x7f2788d15680,
```

```
0x7f2788d114f4) = 0x400
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x3), ...},
AT_EMPTY_PATH) = 0
getrandom("\x88\x70\x65\x00\x04\x38\x83\x75", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x5593a65ab000
brk(0x5593a65cc000)                     = 0x5593a65cc000
write(1, "\n", 1
)                       = 1
write(1, "Informaci\303\263n de los procesos en "..., 44Información de los
procesos en ejecución:
) = 44
write(1, "--------------------------------"..., 41----------------------------
--------------
) = 41
write(1, "PID: 1 | Nombre: systemd | Estad"..., 1017PID: 1 | Nombre: systemd
| Estado: 1
PID: 2 | Nombre: kthreadd | Estado: 1
PID: 3 | Nombre: pool_workqueue_ | Estado: 1
PID: 4 | Nombre: kworker/R-rcu_g | Estado: 8
PID: 5 | Nombre: kworker/R-sync_ | Estado: 8
PID: 6 | Nombre: kworker/R-slub_ | Estado: 8
PID: 7 | Nombre: kworker/R-netns | Estado: 8
PID: 11 | Nombre: kworker/u16:0 | Estado: 8
PID: 12 | Nombre: kworker/R-mm_pe | Estado: 8
PID: 13 | Nombre: rcu_tasks_kthre | Estado: 8
PID: 14 | Nombre: rcu_tasks_rude_ | Estado: 8
PID: 15 | Nombre: rcu_tasks_trace | Estado: 8
PID: 16 | Nombre: ksoftirqd/0 | Estado: 1
PID: 17 | Nombre: rcu_preempt | Estado: 8
PID: 18 | Nombre: rcu_exp_par_gp_ | Estado: 1
PID: 19 | Nombre: rcu_exp_gp_kthr | Estado: 1
PID: 20 | Nombre: migration/0 | Estado: 1
PID: 21 | Nombre: idle_inject/0 | Estado: 1
PID: 22 | Nombre: cpuhp/0 | Estado: 1
PID: 23 | Nombre: cpuhp/1 | Estado: 1
PID: 24 | Nombre: idle_inject/1 | Estado: 1
PID: 25 | Nombre: migration/1 | Estado: 1
PID: 26 | Nombre: ksoftirqd/1 | Estado: 1
) = 1017
write(1, "PID: 2\n", 7PID: 2
)                   = 7
write(1, "--------------------------------"..., 41----------------------------
--------------
) = 41
exit_group(0)                           = ?
+++ exited with 0 +++
```

> **A tener en cuenta**
>
> En mi caso no tengo que hacer `echo $((0x1c4))` ya que la salida de nuestro comando dio esta línea `syscall_0x1d4(0x7ffd08b61730, 0x400, 0x5593695f4dd8, 0, 0x7f2788d15680, 0x7f2788d114f4) = 0x400` así que tenemos que hacer `echo $((0x1d4))`

Luego de ejecutar `echo $((0x1d4))`:

```
root@so:/home/so/practica2# echo $((0x1d4))
468
```

Nos da el valor 468 que es el número de systemcall que invocamos en el programa. Ese 468 es la conversión en decimal del valor en hexadecimal 0x1d4.

Módulos y Drivers

> Referencia >
>
> **Referencia:** http://tldp.org/LDP/lkmpg/2.6/html/c38.html

Conceptos Generales

1. ¿Cómo se denomina en GNU/Linux a la porción de código que se agrega al kernel en tiempo de ejecución? ¿Es necesario reiniciar el sistema al cargarlo? Si no se pudiera utilizar esto. ¿Cómo deberíamos hacer para proveer la misma funcionalidad en Gnu/Linux?

En GNU/Linux, la porción de código que se agrega al kernel en tiempo de ejecución se denomina *módulo*. Estos *módulos* son "pedazos de código" que pueden ser cargados y descargados bajo demanda, extendiendo la funcionalidad del kernel. *No es necesario reiniciar el sistema al cargar o descargar un módulo.* Esta es precisamente una de las ventajas de utilizar módulos, ya que permiten modificar la funcionalidad del kernel "en caliente".

Si no se pudieran utilizar módulos, para proveer la misma funcionalidad en GNU/Linux, *todo el soporte debería estar incluido directamente en la imagen del kernel.* Esto implicaría que cualquier nueva funcionalidad o soporte de hardware requeriría *modificar el código fuente del kernel, recompilarlo e instalar el nuevo kernel, lo cual sí requeriría reiniciar el sistema* para que los cambios surtan efecto. Sin módulos, el kernel sería 100% monolítico.

2. ¿Qué es un driver? ¿Para qué se utiliza?

Un driver es un programa que permite que el sistema operativo se comunique con un dispositivo de hardware específico. Actúa como intermediario entre el Sistema Operativo y el Hardware. Se utiliza para:

- Hacer funcionar el hardware correctamente.
- Permitir que el Sistema Operativo gestione y controle el dispositivo.
- Aislar al usuario y al software de los detalles técnicos del hardware.

3. ¿Por qué es necesario escribir drivers?

Es necesario escribir drivers porque el sistema operativo no puede comunicarse directamente con todo tipo de hardware, ya que:

- Cada dispositivo es diferente, tienen funciones, interfaces y protocolos únicos, definidos por sus fabricantes. El sistema operativo no puede tener soporte nativo para todos los dispositivos posibles que existen o existirán. Por eso, se necesitan drivers que traduzcan las órdenes genéricas del sistema en instrucciones específicas para cada dispositivo.
- Un driver actúa como intermediario entre el Sistema Operativo y el dispositivo.
- Escribir drivers permite que el sistema trate todos los dispositivos similares de la misma forma, sin importar el modelo exacto.

4. ¿Cuál es la relación entre módulo y driver en GNU/Linux?

En GNU/Linux, muchos drivers están implementados como módulos del kernel, es decir, un driver puede ser un módulo, pero no todos los módulos son drivers.

5. ¿Qué implicancias puede tener un bug en un driver o módulo?

Un bug en un driver o módulo del kernel puede tener implicancias muy serias como por ejemplo comprometer la estabilidad, la seguridad, la compatibilidad y el funcionamiento completo del Sistema Operativo.

6. ¿Qué tipos de drivers existen en GNU/Linux?

En GNU/Linux, acorde a la clasificación del hardware, existen principalmente dos tipos de drivers:

- *Dispositivos de bloques:* Estos drivers gestionan dispositivos que acceden a los datos en grupos de bloques persistentes, generalmente de 1024 bytes. Las operaciones de lectura y escritura se realizan a nivel de bloques, y ejemplos de estos dispositivos son los discos.

- *Dispositivos de caracter:* Estos drivers gestionan dispositivos a los que se accede byte por byte, y cada byte solo puede ser leído una única vez. Ejemplos de dispositivos seriales como el mouse o la tarjeta de sonido utilizan drivers de caracter.

7. ¿Qué hay en el directorio /dev? ¿Qué tipos de archivo encontramos en esa ubicación?

El directorio `/dev` en GNU/Linux es fundamental para la comunicación con el hardware del sistema. Contiene los llamados archivos de dispositivos (device files), que son interfaces que permiten a los programas interactuar con el hardware como si fueran archivos comunes.

En esencia, `/dev` contiene representaciones del hardware y dispositivos virtuales en forma de archivos especiales. Cuando un programa quiere interactuar con un dispositivo (como un disco, teclado o terminal), accede al archivo correspondiente dentro de este directorio.

Específicamente encontramos archivos de:

- Dispositivos de caracteres: Proporcionan una interfaz simple para dispositivos que transmiten datos carácter por carácter (por ejemplo, teclados, terminales, puertos serie). No requieren almacenamiento en búfer.
- Dispositivos de bloques: Proporcionan acceso a dispositivos que manejan datos en bloques (como discos rígidos, SSD, pendrives). El acceso se realiza a través de una caché, y permiten operaciones más complejas de lectura y escritura.
- Pseudodispositivos: No representan hardware real, sino que ofrecen funcionalidades útiles del sistema. Ejemplos comunes son:
  - `/dev/null`: Descarta cualquier dato escrito en él.
  - `/dev/random`: Genera datos aleatorios. Estos también se representan como dispositivos de caracteres. Se pueden distinguir mediante el número mayor (major number).
- Enlaces simbólicos: Algunos archivos en `/dev` no son directamente dispositivos, sino enlaces simbólicos a otros archivos de dispositivo. Esto permite organizar y acceder a dispositivos de forma más intuitiva. Por ejemplo:
  - `/dev/cdrom` → `/dev/sr0`

8. ¿Para qué sirven el archivo /lib/modules/<version>/modules.dep utilizado por el comando modprobe?

El archivo `/lib/modules/<versión>/modules.dep` es clave para el funcionamiento del comando `modprobe`, ya que le indica las dependencias entre los módulos del kernel.

`modules.dep` es un archivo de texto que lista cada módulo y los módulos de los que depende para funcionar correctamente. Por ejemplo:
`kernel/drivers/net/ethernet/e1000e/e1000e.ko:`

`kernel/drivers/net/ethernet/libphy.ko`. Esto significa que el módulo `e1000e.ko` necesita que primero se cargue `libphy.ko`.

`modprobe` usa el archivo `modules.dep` de la siguiente forma:

- Lee `modules.dep`.
- Carga primero los módulos requeridos (dependencias).
- Luego carga el módulo solicitado.
  Esto permite usar `modprobe` sin preocuparse de la jerarquía de dependencias.

9. ¿En qué momento/s se genera o actualiza un initramfs?

El initramfs (Initial RAM Filesystem) se genera o actualiza en ciertos momentos clave del sistema:

- Instalación o actualización del kernel.
- Cambio de controladores o módulos importantes.
- Modificación de configuraciones críticas de arranque.
- Cambios en módulos incluidos en initramfs.

10. ¿Qué módulos y drivers deberá tener un initramfs mínimamente para cumplir su objetivo?

Para que el initramfs cumpla su función principal (montar correctamente el sistema de archivos raíz durante el arranque), necesita incluir al menos los siguientes módulos y drivers mínimos:

- *Módulos de sistema de archivos:* Permiten montar el sistema raíz. Algunos ejemplos comunes: `ext4`, `xfs`, `btrfs`.
- *Controladores de almacenamiento:* Permiten acceder a los discos donde está instalado el sistema: `ahci` (para SATA), `nvme` (para unidades NVMe), `sd_mod`, `scsi_mod` (dispositivos SCSI), etc.
- *Controladores del sistema de archivos raíz:* Si el sistema raíz está en:
  - Un disco LVM → se necesita el módulo `dm_mod` y `dm_crypt` si hay cifrado.
  - Un volumen RAID → se requiere `md_mod`.
  - Un disco cifrado (LUKS) → se necesita `dm_crypt`, `cryptsetup`.
- *Controladores de bus del sistema:* Para que el sistema reconozca el hardware: `pci_bus`, `usbcore` (si el disco está conectado por USB), `xhci_hcd`, `ehci_hcd`, `uhci_hcd` (USB host controllers), etc.
- *Drivers específicos del hardware de nuestra máquina:* Por ejemplo, si nuestro disco requiere un driver especial del fabricante.

Práctica Guiada

Desarrollando un módulo simple para Linux

> Objetivo >
>
> El objetivo de este ejercicio es crear un módulo sencillo y poder cargarlo en nuestro kernel con el fin de consultar que el mismo se haya registrado correctamente.

1. Crear el archivo memory.c con el siguiente código (puede estar en cualquier directorio, incluso fuera del directorio del kernel):

```
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
```

2. Crear el archivo Makefile con el siguiente contenido: `obj-m := memory.o`. Responda lo siguiente:
    1. Explique brevemente cual es la utilidad del archivo Makefile.
    - El archivo `Makefile` se utiliza para indicarle al sistema de compilación del kernel cómo construir nuesto módulo. En este caso el contenido del Makefile le dice al compilador "Quiero construir un módulo externo (obj-m) llamado `memory.ko`, y su código fuente está en `memory.c`"
    2. ¿Para qué sirve la macro MODULE_LICENSE? ¿Es obligatoria?
    - La macro `MODULE_LICENSE` no es obligatoria para compilar, pero sí muy recomendable. Si no se incluye, el módulo igual se puede cargar, pero con advertencias y limitaciones. La macro sirve para:
        - Indicar la licencia bajo la cual se distribuye tu módulo.
        - Informar al kernel si el módulo es "compatible" con el núcleo (por ejemplo, si es GPL).
        - Si usás una licencia compatible con el kernel (como `"GPL"` o `"Dual BSD/GPL"`), se habilitan funciones internas del kernel a las que solo pueden acceder los módulos GPL.
        - Si no ponés la macro, o usás una licencia no compatible, el kernel marca el módulo como propietario, lo cual restringe el acceso a ciertas APIs internas y emite un warning cuando cargás el módulo (`taint` del kernel).
3. Ahora es necesario compilar nuestro módulo usando el mismo kernel en que correrá el mismo, utilizaremos el que instalamos en el primer paso del ejercicio guiado. `$ make -C <KERNEL_CODE> M=$(pwd) modules`. Responda lo siguiente:

1. ¿Cuál es la salida del comando anterior?

- Ver abajo.

2. ¿Qué tipos de archivo se generan? Explique para qué sirve cada uno.

- Tipos de archivos generados:
  - `memory.o` : Es el archivo objeto generado a partir del código fuente `memory.c`. Contiene el código ya compilado, pero aún no es un módulo del kernel.
  - `memory.mod.o` : Código objeto que incluye información adicional sobre el módulo. Integra los metadatos requeridos para que el kernel lo entienda como módulo (como la licencia, símbolos exportados, etc.).
  - `memory.ko` : Es el archivo final del módulo del kernel (Kernel Object). Este es el archivo que se puede cargar en el kernel usando `insmod` o `modprobe`.
  - `modules.order` : Lista de los módulos que fueron compilados. Se usa para indicar el orden en que deben cargarse si hay múltiples módulos dependientes.
  - `Module.symvers` : Tabla de símbolos exportados por los módulos. Es útil para compilar otros módulos que dependan de este (por ejemplo, si uno exporta funciones para ser usadas en otro módulo).
  - `.memory.mod.cmd` (y otros archivos `.cmd`): Archivos auxiliares de construcción. Guardan información sobre cómo se compiló el módulo para evitar recompilar innecesariamente.

3. Con lo visto en la Práctica 1 sobre Makefiles, construya un Makefile de manera que si ejecuto:
   1. *make*, nuestro módulo se compila.
   2. *make clean*, limpia el módulo y el código objeto generado.
   3. *make run*, ejecuta el programa.

- Ver abajo de la salida del comando.

Salida del Comando:

```
so@so:~/practica2/modulo$ make -C /lib/modules/$(uname -r)/build M=$(pwd)
modules
make: se entra en el directorio '/home/so/kernel/linux-6.13'
make[1]: se entra en el directorio '/home/so/practica2/modulo'
  CC [M]  memory.o
  MODPOST Module.symvers
  CC [M]  memory.mod.o
  CC [M]  .module-common.o
  LD [M]  memory.ko
make[1]: se sale del directorio '/home/so/practica2/modulo'
make: se sale del directorio '/home/so/kernel/linux-6.13'
```

Makefile:

```
# Nombre del módulo (sin extensión)
obj-m := memory.o

# Directorio con el código fuente del kernel
KDIR := /lib/modules/$(shell uname -r)/build

# Directorio actual
PWD := $(shell pwd)

# Compilar el módulo
all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

# Limpiar archivos generados
clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean

# Cargar y descargar el módulo
run: all
    su -c "/sbin/insmod memory.ko"
    su -c "dmesg | tail -n 10"
    sleep 2
    su -c "/sbin/rmmod memory"
    su -c "dmesg | tail -n 10"
```

Uso del Makefile:

```
so@so:~/practica2/modulo$ ls
Makefile  memory.c
so@so:~/practica2/modulo$ make
make -C /lib/modules/6.13.7/build M=/home/so/practica2/modulo modules
make[1]: se entra en el directorio '/home/so/kernel/linux-6.13'
make[2]: se entra en el directorio '/home/so/practica2/modulo'
  CC [M]  memory.o
  MODPOST Module.symvers
  CC [M]  memory.mod.o
  CC [M]  .module-common.o
  LD [M]  memory.ko
make[2]: se sale del directorio '/home/so/practica2/modulo'
make[1]: se sale del directorio '/home/so/kernel/linux-6.13'
so@so:~/practica2/modulo$ ls
Makefile  memory.ko   memory.mod.c  memory.o       Module.symvers
memory.c  memory.mod  memory.mod.o  modules.order
```

```
so@so:~/practica2/modulo$ make run
make -C /lib/modules/6.13.7/build M=/home/so/practica2/modulo modules
make[1]: se entra en el directorio '/home/so/kernel/linux-6.13'
make[2]: se entra en el directorio '/home/so/practica2/modulo'
make[2]: se sale del directorio '/home/so/practica2/modulo'
make[1]: se sale del directorio '/home/so/kernel/linux-6.13'
su -c "/sbin/insmod memory.ko"
Contraseña:
su -c "dmesg | tail -n 10"
Contraseña:
[   21.507930] Console: switching to colour frame buffer device 160x50
[   21.564666] vmwgfx 0000:00:02.0: [drm] fb0: vmwgfxdrmfb frame buffer
device
[   69.666883] hrtimer: interrupt took 44429020 ns
[ 4640.512221] memory: loading out-of-tree module taints kernel.
[ 4640.512233] memory: module verification failed: signature and/or required
key missing - tainting kernel
[ 5861.459077] Hello world!
[ 6021.421239] Bye, cruel world
[ 6746.162179] Hello world!
[ 6757.947169] Bye, cruel world
[ 6947.794958] Hello world!
sleep 2
su -c "/sbin/rmmod memory"
Contraseña:
su -c "dmesg | tail -n 10"
Contraseña:
[   21.564666] vmwgfx 0000:00:02.0: [drm] fb0: vmwgfxdrmfb frame buffer
device
[   69.666883] hrtimer: interrupt took 44429020 ns
[ 4640.512221] memory: loading out-of-tree module taints kernel.
[ 4640.512233] memory: module verification failed: signature and/or required
key missing - tainting kernel
[ 5861.459077] Hello world!
[ 6021.421239] Bye, cruel world
[ 6746.162179] Hello world!
[ 6757.947169] Bye, cruel world
[ 6947.794958] Hello world!
[ 6956.835879] Bye, cruel world
so@so:~/practica2/modulo$ make clean
make -C /lib/modules/6.13.7/build M=/home/so/practica2/modulo clean
make[1]: se entra en el directorio '/home/so/kernel/linux-6.13'
make[2]: se entra en el directorio '/home/so/practica2/modulo'
  CLEAN   Module.symvers
make[2]: se sale del directorio '/home/so/practica2/modulo'
make[1]: se sale del directorio '/home/so/kernel/linux-6.13'
```

```
so@so:~/practica2/modulo$ ls
Makefile  memory.c
```

4. El paso que resta es agregar y eventualmente quitar nuestro módulo al kernel en tiempo de ejecución. Ejecutamos: `# insmod memory.ko`. Responda lo siguiente:
    1. ¿Para qué sirven el comando insmod y el comando modprobe? ¿En qué se diferencian?.

- Ambos comandos sirven para cargar módulos del kernel de Linux, pero tienen diferencias importantes.
    - `insmod` carga un único módulo en el kernel. No resuelve dependencias automáticamente. Solo sirve si el módulo no necesita otros módulos cargados previamente.
    - `modprobe` carga un módulo y sus dependencias automáticamente. Usa la configuración del sistema para buscar el módulo en `/lib/modules/$(uname -r)/`. Lee archivos como `modules.dep` para saber qué otros módulos debe cargar. Más inteligente y recomendado para uso común.

5. Ahora ejecutamos: `$ lsmod | grep memory`. Responda los siguiente:
    1. ¿Cuál es la salida del comando? Explique cuál es la utilidad del comando lsmod.

- Ver salida abajo. `lsmod` es un comando en Linux que muestra todos los módulos del kernel que están actualmente cargados en el sistema. Sirve para:
    - Ver qué módulos están activos en el kernel.
    - Diagnosticar problemas con drivers o módulos.
    - Ver dependencias entre módulos (qué módulo depende de cuál).
    - Confirmar si un módulo que cargaste con `insmod` o `modprobe` fue realmente insertado.

    2. ¿Qué información encuentra en el archivo /proc/modules?

- El archivo `/proc/modules` es una vista del sistema de archivos proc que muestra información sobre los módulos actualmente cargados en el kernel. Cada línea representa un módulo cargado y contiene 7 campos separados por espacios: `nombre tamaño usos dependencias estado dirección`
    - Nombre: El nombre del módulo (por ejemplo, `memory`, `i915`, etc).
    - Tamaño: El tamaño del módulo en bytes.
    - Usos: Cantidad de veces que el módulo está siendo usado (por otros módulos o procesos).
    - Dependencias: Lista de otros módulos de los que depende (separados por comas) o `-` si no depende de ninguno.
    - Estado: Generalmente `Live` si el módulo está activo.
    - Dirección de carga: Dirección de memoria donde el módulo está cargado.

3. Si ejecutamos more /proc/modules encontramos los siguientes fragmentos ¿Qué información obtenemos de aquí? (Ver código de abajo).

- `memory 8192 0 - Live 0x0000000000000000 (OE)`
  - Nombre: `memory`.
  - Tamaño: `8192` bytes.
  - Usos: `0` → no está siendo utilizado por ningún otro módulo
  - Dependencias: `-` → no depende de otros módulo.
  - Estado: `Live` → está cargado y activo.
  - Dirección: `0x0000000000000000` → dirección en memoria donde está cargado.
  - (OE): indica que el módulo fue cargado con símbolos de exportación que permiten ser enlazados por otros módulos (O = Open, E = Exported).
- `binfmt_misc 24576 1 - Live 0x0000000000000000`
  - Nombre: `binfmt_misc` (permite ejecutar binarios de otros formatos, como scripts sin shebang o binarios de Windows con Wine).
  - Tamaño: `24576` bytes.
  - Usos: `1` → está siendo usado por otro módulo o proceso.
  - Dependencias: `-` → no tiene dependencias explícitas.
  - Estado: `Live`.
  - Dirección: `0x0000000000000000`.
- `intel_rapl_msr 16384 0 - Live 0x0000000000000000`
  - Nombre: `intel_rapl_msr` (control de energía en procesadores Intel usando registros MSR).
  - Tamaño: `16384` bytes.
  - Usos: `0`.
  - Dependencias: `-`.
  - Estado: `Live`.
  - Dirección: `0x0000000000000000`.
- `intel_rapl_common 32768 1 intel_rapl_msr, Live 0x0000000000000000`
  - Nombre: `intel_rapl_common` (módulo común para control energético de Intel).
  - Tamaño: `32768` bytes.
  - Usos: `1`.
  - Dependencias: `intel_rapl_msr` → depende de este módulo.
  - Estado: `Live`.
  - Dirección: `0x0000000000000000`.

4. ¿Con qué comando descargamos el módulo de la memoria?

- Para descargar (remover) un módulo del kernel, como tu módulo `memory`, podés usar el comando: `# rmmod memory`. Sino podemos usar `# modprobe -r memory`, `modprobe -r` maneja dependencias automáticamente (a diferencia de `rmmod`), así que es más seguro si tu módulo depende de otros o es dependido.

```
memory 8192 0 - Live 0x0000000000000000 (OE)
binfmt_misc 24576 1 - Live 0x0000000000000000
intel_rapl_msr 16384 0 - Live 0x0000000000000000 intel_rapl_common 32768 1
intel_rapl_msr, Live 0x0000000000000000
```

Salida de `lsmod | grep memory`:

```
root@so:/home/so/practica2/modulo# lsmod | grep memory
memory                  8192  0
```

6. Descargue el módulo memory. Para corroborar que efectivamente el mismo ha sido eliminado del kernel ejecute el siguiente comando: `lsmod | grep memory`.

Ahora el comando no tiene ninguna salida porque *no encuentra* algún módulo que siga el patrón especificado en el `grep`.

7. Modifique el archivo memory.c de la siguiente manera: (Ver código de abajo).
    1. Compile y cargue en memoria el módulo.
    2. Invoque al comando dmesg.
    3. Descargue el módulo de memoria y vuelva a invocar a dmesg.

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void) {
    printk("Hello world!\n");
    return 0;
}

static void hello_exit(void) {
    printk("Bye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Luego de volverlo a cargar:

```
[ 4640.512221] memory: loading out-of-tree module taints kernel.
[ 4640.512233] memory: module verification failed: signature and/or required
key missing - tainting kernel
[ 5861.459077] Hello world!
```

Luego de descargarlo de la memoria:

```
[ 4640.512221] memory: loading out-of-tree module taints kernel.
[ 4640.512233] memory: module verification failed: signature and/or required
key missing - tainting kernel
[ 5861.459077] Hello world!
[ 6021.421239] Bye, cruel world
```

8. Responda lo siguiente:
    1. ¿Para qué sirven las funciones module_init y module_exit?. ¿Cómo haría para ver la información del log que arrojan las mismas?.

- Las funciones `module_init` y `module_exit` en un módulo del kernel de Linux son fundamentales para controlar cuándo se cargan y descargan los módulos.
    - `module_init(init_function)` se ejecuta automáticamente cuando el módulo es cargado con `insmod` o `modprobe`. Usás esta función para inicializar tu módulo: asignar recursos, registrar dispositivos, imprimir mensajes, etc.
    - `module_exit(exit_function)` se ejecuta automáticamente cuando el módulo es descargado con `rmmod` o `modprobe -r`. Es ideal para liberar recursos, desregistrar dispositivos, limpiar estructuras, etc.

    2. Hasta aquí hemos desarrollado, compilado, cargado y descargado un módulo en nuestro kernel. En este punto y sin mirar lo que sigue. ¿Qué nos falta para tener un driver completo?.

- Justamente nos falta la interacción con algún dispositvo ya sea virtual o físico.

    3. Clasifique los tipos de dispositivos en Linux. Explique las características de cada uno.

- Dispositivos de acceso aleatorio:
    - Almacenan y recuperan datos en bloques de tamaño fijo.
    - Permiten un acceso no secuencial a los datos, lo que significa que se puede leer o escribir en cualquier ubicación de manera aleatoria.
    - Son dispositivos de almacenamiento de datos que mantienen la integridad de los datos independientemente del orden en que se accede a ellos.
    - *Ejemplos:* discos duros (HDD), unidades de estado sólido (SSD), unidades USB y tarjetas de memoria.
    - *Acceso:*

        - El acceso se realiza mediante operaciones de lectura y escritura en bloques de datos.
        - Se pueden formatear con sistemas de archivos como ext4, NTFS, FAT32, etc.
        - Los sistemas de archivos proporcionan una abstracción para organizar y administrar los datos almacenados en estos dispositivos.
- **Dispositivos seriales (por ejemplo, mouse, sonido, etc.):**
    - Transmiten datos secuenciales, uno tras otro, en forma de caracteres.
    - Son dispositivos de entrada/salida que manejan datos de manera secuencial, caracter por caracter.
    - Los datos se transmiten y reciben en serie, uno detrás del otro, en lugar de en bloques.
    - *Ejemplos:* mouse, teclado, dispositivos de sonido (altavoces, micrófonos), GPS, etc.
    - *Acceso:*
        - Los datos se leen o escriben secuencialmente, carácter por carácter, sin estructura de bloques.
        - Los controladores de dispositivos proporcionan una interfaz para que el kernel del sistema operativo interactúe con estos dispositivos.
        - Los eventos generados por estos dispositivos (como movimientos del mouse, pulsaciones de teclas, datos de audio) se procesan y utilizan para diversas funciones y aplicaciones en el sistema.

Desarrollando un Driver

> Objetivo
>
> Ahora completamos nuestro módulo para agregarle la capacidad de escribir y leer un dispositivo. En nuestro caso el dispositivo a leer será la memoria de nuestra CPU, pero podría ser cualquier otro dispositivo.

1. Modifique el archivo memory.c para que tenga el siguiente código: https://gitlab.com/unlp-so/codigo-para-practicas/-/blob/main/practica2/crear_driver/1_memory.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
```

```c
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <linux/uaccess.h> /* copy_from/to_user */

MODULE_LICENSE("Dual BSD/GPL");

int memory_open(struct inode *inode, struct file *filp);
int memory_release(struct inode *inode, struct file *filp);
ssize_t memory_read(struct file *filp, char *buf, size_t count, loff_t *
        f_pos);
ssize_t memory_write(struct file *filp, const char *buf, size_t count,
        loff_t *f_pos);
void memory_exit(void);
int memory_init(void);

/* Structure that declares the usual file */
/* access functions */
struct file_operations memory_fops = {
read: memory_read,
      write: memory_write,
      open: memory_open,
      release: memory_release
};

/* Declaration of the init and exit functions */
module_init(memory_init);
module_exit(memory_exit);

/* Global variables of the driver */
/* Major number */
int memory_major = 60;
/* Buffer to store data */
char *memory_buffer;

int memory_init(void) {
    int result;
    /* Registering device */
    result = register_chrdev(memory_major, "memory", &memory_fops);
    if (result < 0) {
        printk("<1>memory: cannot obtain major number %d\n", memory_major);
        return result;
    }
    /* Allocating memory for the buffer */
    memory_buffer = kmalloc(1, GFP_KERNEL);
    if (!memory_buffer) {
        result = -ENOMEM;
```

```
        goto fail;
    }
    memset(memory_buffer, 0, 1);
    printk("<1>Inserting memory module\n");
    return 0;
fail:
    memory_exit();
    return result;
}

void memory_exit(void) {
    /* Freeing the major number */
    unregister_chrdev(memory_major, "memory");

    /* Freeing buffer memory */
    if (memory_buffer) {
        kfree(memory_buffer);
    }
    printk("<1>Removing memory module\n");
}

int memory_open(struct inode *inode, struct file *filp) {
    /* Success */
    return 0;
}

int memory_release(struct inode *inode, struct file *filp) {
    /* Success */
    return 0;
}

ssize_t memory_read(struct file *filp, char *buf,
        size_t count, loff_t *f_pos) {
    printk("memory_read()\n");
    /* Transfering data to user space */
    if (copy_to_user(buf,memory_buffer,1)) {
        // return 0 if copy_to_user fails
        return 0;
    }
    /* Changing reading position as best suits */
    if (*f_pos == 0) {
        *f_pos+=1;
        return 1;
    } else {
        return 0;
    }
```

```
}

ssize_t memory_write( struct file *filp, const char *buf,
        size_t count, loff_t *f_pos) {
    const char *tmp;
    tmp=buf+count-1;
    printk("memory_write()\n");
    if (copy_from_user(memory_buffer,tmp,1)) {
        // return 0 if copy_from_user fails
        return 0;
    }
    return 1;
}
```

2. Responda lo siguiente:
    1. ¿Para qué sirve la estructura ssize_t y memory_fops? ¿Y las funciones register_chrdev y unregister_chrdev?

- `ssize_t`: Es un tipo de dato firmado que se utiliza para representar el tamaño de una transferencia de datos (como la cantidad de bytes leídos o escritos).
    - Valor positivo → número de bytes transferidos.
        - Valor 0 → fin de archivo o sin datos.
        - Valor negativo → error (usualmente un código de error como `-EFAULT`, `-ENOMEM`, etc.).
- `memory_fops` (`struct file_operations`): Define qué funciones del driver deben llamarse cuando una aplicación de espacio de usuario realiza operaciones sobre el dispositivo (como `open()`, `read()`, `write()`, `release()`, etc.). Es el "puente" entre llamadas al sistema (`read()`, `write()`, etc.) y tu código.
- `register_chrdev(int major, const char *name, struct file_operations *)`: Registra un dispositivo de carácter con el número mayor especificado (`memory_major = 60`). Asocia ese número a un conjunto de funciones (las que están en `memory_fops`). Si el número mayor es 0, el kernel asigna uno dinámicamente.
- `unregister_chrdev(int major, const char *name)`: Libera el número mayor y desregistra el dispositivo del sistema.

    2. ¿Cómo sabe el kernel que funciones del driver invocar para leer y escribir al dispositivo?

- Gracias a la estructura `file_operations` (`memory_fops` en este caso), el kernel sabe qué funciones invocar para cada operación estándar:
    - Cuando un proceso hace `open("/dev/memory", O_RDONLY)`, el kernel llama a `memory_open`.
    - Cuando se hace `read(fd, ...)`, llama a `memory_read`.
    - Cuando se hace `write(fd, ...)`, llama a `memory_write`.

- Al cerrar con `close(fd)`, llama a `memory_release`.

3. ¿Cómo se accede desde el espacio de usuario a los dispositivos en Linux?

- Desde el espacio de usuario, los dispositivos se acceden a través de archivos en el sistema de archivos, usualmente dentro del directorio `/dev`.

4. ¿Cómo se asocia el módulo que implementa nuestro driver con el dispositivo?

- La asociación se hace en dos pasos:
  1. `register_chrdev(...)`: Asocia el número mayor al conjunto de funciones (`memory_fops`).
  2. Archivo en `/dev`: Se crea manualmente (con `mknod`) o automáticamente (con `udev`) un archivo de dispositivo que usa ese número mayor. Cuando un proceso accede al archivo, el kernel busca el número mayor y llama a la función adecuada.

5. ¿Qué hacen las funciones copy_to_user y copy_from_user? (https://developer.ibm.com/technologies/linux/articles/l-kernel-memory-access/).

- Estas funciones se usan para mover datos entre el espacio de kernel (donde corre el driver) y el espacio de usuario (donde corre la aplicación que llama al driver):
  - `copy_to_user(void *to, const void *from, unsigned long n)`: Copia `n` bytes desde una dirección del kernel (`from`) hacia una dirección en espacio de usuario (`to`).
  - `copy_from_user(void *to, const void *from, unsigned long n)`: Copia `n` bytes desde una dirección en espacio de usuario (`from`) hacia una dirección del kernel (`to`).

3. Ahora ejecutamos lo siguiente: `# mknod /dev/memory c 60 0`
4. Y luego: `# insmod memory.ko`. Responda lo siguiente:
   1. ¿Para qué sirve el comando mknod? ¿qué especifican cada uno de sus parámetros?.

- El comando `mknod` en Linux se utiliza para crear archivos de dispositivo en el sistema de archivos, típicamente dentro del directorio `/dev`. Los dispositivos pueden ser:
  - Dispositivos de carácter (`c`) → se accede byte a byte (por ejemplo: terminales, puertos seriales).
  - Dispositivos de bloque (`b`) → se accede en bloques (por ejemplo: discos, pendrives).
- Parámetros:
  - `<nombre_archivo>`: Nombre del archivo de dispositivo a crear, por ejemplo `/dev/memory`.
  - `<tipo>`: Tipo de dispositivo según los de arriba.
  - `<num_major>`: Identifica qué driver del kernel manejará las llamadas a este archivo (asignado por el driver en `register_chrdev`).
  - `<num_minor>`: El número menor identifica una instancia específica del dispositivo manejado por ese driver.

2. ¿Qué son el “major” y el “minor” number? ¿Qué referencian cada uno?
   - Puesto arriba.
5. Ahora escribimos a nuestro dispositivo: `echo -n abcdef > /dev/memory`.
6. Ahora leemos desde nuestro dispositivo: `more /dev/memory`.
7. Responda lo siguiente:
   1. ¿Qué salida tiene el anterior comando?, ¿Porque? *(ayuda: siga la ejecución de las funciones memory_read y memory_write y verifique con dmesg).*
   - **Ver abajo.**
   2. ¿Cuántas invocaciones a memory_write se realizaron?
   - Se realizaron 6 invocaciones.
   3. ¿Cuál es el efecto del comando anterior? ¿Por qué?
   - En primer lugar tenemos que tener en cuenta que `echo -n abcdef > /dev/memory` escribe **secuencialmente** los caracteres `a`, `b`, `c`, `d`, `e`, `f` al dispositivo `/dev/memory`, por esto es que hay **6 invocaciones** a `memory_write` que hace lo siguiente:
     - `tmp = buf + count - 1; copy_from_user(memory_buffer, tmp, 1);`
   - Ese código **escribe solo el último byte** recibido por el `write()` en la primera posición del `memory_buffer`, por eso solo se lee al final `f`.
   3. Hasta aquí hemos desarrollado un ejemplo de un driver muy simple pero de manera completa, en nuestro caso hemos escrito y leído desde un dispositivo que en este caso es la propia memoria de nuestro equipo.
   4. En el caso de un driver que lee un dispositivo como puede ser un file system, un dispositivo usb, etc. ¿Qué otros aspectos deberíamos considerar que aquí hemos omitido? *ayuda: semáforos, ioctl, inb, outb.*
   - Cuando tenemos un *driver más realista* tenemos que tener en cuenta los siguientes aspectos:
     - **Sincronización:** Cuando hay **acceso concurrente** al dispositivo necesitamos sincronizar. Se suelen usar semáforos.
     - **Acceso a puertos de E/S:** Para drivers de hardware como puertos serie, USB o PCI que requieren **I/O directo**, usás funciones de acceso a **puertos de entrada/salida**:
       - `inb(port)` – Lee 1 byte del puerto.
       - `outb(value, port)` – Escribe 1 byte al puerto.
     - **Control de Dispositivos** `ioctl()`: El Driver puede necesitar permitir que los programas de usuario le den órdenes **más complejas que solo leer o escribir**, como:
       - Cambiar la configuración del dispositivo.
       - Pedir el estado actual.
       - Iniciar/terminar operaciones especiales.

Salida de `more dev/memory`

```
root@so:/home/so/practica2/modulo# more /dev/memory
f
```

Salida de `dmesg`

```
[ 8109.512420] <1>Inserting memory module
[ 8425.278123] memory_write()
[ 8425.278135] memory_write()
[ 8425.278136] memory_write()
[ 8425.278137] memory_write()
[ 8425.278139] memory_write()
[ 8425.278140] memory_write()
[ 8434.993772] memory_read()
[ 8434.993851] memory_read()
```