

# **SISTEMAS OPERATIVOS**

## **Práctica 2**

## System Calls

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

Es el mecanismo utilizado por un proceso de usuario para solicitar un servicio al Sistema Operativo (SO). Permite que un proceso de usuario pueda acceder a funciones o servicios que deben ser protegidas por el SO. El SO provee una interfaz de programación (API) que los procesos pueden utilizar en cualquier momento para solicitar recursos gestionados por él mismo.

### 2. ¿Para qué sirve la macro syscall? Describa el propósito de cada uno de sus parámetros.

La macro syscall permite hacer llamadas al sistema directamente desde un programa en C, sin usar las funciones estándar como read, write, etc. Haciendo esto, no pasamos por ninguna librería sino que directamente se llama a la syscall.

long syscall(long number, ...);

number: El número de la syscall (como SYS\_write, SYS\_open, etc.).

...: Los argumentos que necesita esa syscall (por ejemplo, el descriptor de archivo, el buffer, etc.).

### 3. Ejecute el siguiente comando e identifique el propósito de cada uno de los archivos que encuentra

`ls -lh /boot | grep vmlinuz`

```
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 21 09:51 vmlinuz-6.13.7
so@so:~$
```

Todos son imágenes del kernel de Linux, es decir, versiones del núcleo del sistema operativo que el sistema puede usar para arrancar.

#### 1. `vmlinuz-6.1.0-29-amd64`

- Versión del kernel: 6.1.0-29
- Arquitectura: 64 bits ( `amd64` )
- Propósito: Versión anterior del kernel, conservada en caso de que la más nueva tenga errores.

#### 2. `vmlinuz-6.1.0-31-amd64`

- Versión del kernel: 6.1.0-31
- Arquitectura: 64 bits
- Propósito: Muy probablemente es el **kernel en uso actualmente**, por ser más reciente.

#### 3. `vmlinuz-6.13.7`

- Versión del kernel: 6.13.7
- No tiene sufijo de arquitectura (aunque seguramente también sea `amd64` )
- Propósito: Puede haber sido **instalado manualmente** o desde una fuente distinta a la del sistema estándar.

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>

```
root@so:/home/so/kernel/linux-6.13# git clone https://github.com/torvalds/linux.git
Clonando en 'linux'...
remote: Enumerating objects: 10788128, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
Recibiendo objetos: 100% (10788128/10788128), 5.28 GiB | 7.63 MiB/s, listo.
remote: Total 10788128 (delta 3), reused 1 (delta 1), pack-reused 10788124 (from 2)
Resolviendo deltas: 100% (8771287/8771287), listo.
Actualizando archivos: 100% (88814/88814), listo.
```

5. ¿Para qué sirven el siguiente archivo?

`arch/x86/entry/syscalls/syscall_64.tbl`

Este archivo define la tabla de syscalls (llamadas al sistema) para arquitecturas de 64 bits (x86\_64). Tiene:

- El número de cada syscall.
- Su nombre.
- El número de argumentos.
- En qué archivo del kernel se implementa.

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

Permite ver qué llamadas al sistema realiza un programa mientras se ejecuta.

Es ideal para depuración, análisis de errores o entender cómo trabaja un programa.

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

ausyscall se usa para obtener números de syscalls y generar reglas de auditoría en sistemas Linux.

Para obtener el número de una syscall:

`ausyscall --name syscall_name`

Por ejemplo, para obtener el número de la syscall read:

`ausyscall --name read`

Esto devolverá el número de la syscall para read

## Práctica guiada:

• ¿Para qué sirven los macros `SYS_CALL_DEFINE`?

`SYS_CALL_DEFINE1(my_sys_call, int, arg)`: Define una syscall llamada `my_sys_call` que toma un argumento de tipo `int`.

`SYS_CALL_DEFINE2(get_task_info, char __user *, buffer, size_t, length)`: Define una syscall llamada `get_task_info` que toma dos argumentos: un puntero al espacio de usuario `buffer` y un tamaño `length`.

`SYS_CALL_DEFINE2(get_threads_info, char __user *, buffer, size_t, length)`: Define otra syscall llamada `get_threads_info` que también toma dos argumentos.

- ¿Para que se utilizan la macros `for_each_process` y `for_each_thread`?

`for_each_process(task)`: Itera sobre todos los procesos que están en ejecución en el sistema. El argumento `task` es un puntero a cada estructura `task_struct` que representa un proceso. La macro proporciona acceso a todos los procesos activos del sistema.

`for_each_thread(task, thread)`: Itera sobre todos los hilos (threads) de un proceso específico. El primer argumento `task` es el proceso en el que queremos iterar, y el segundo `thread` es un puntero a cada hilo asociado a ese proceso.

- ¿Para que se utiliza la función `copy_to_user`?

Esta llamada intenta copiar los datos de kbuffer al buffer de espacio de usuario. Si ocurre algún error (por ejemplo, si no se puede copiar), retorna -EFAULT.

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

La función `printk` es utilizada en el kernel de Linux para imprimir mensajes en el registro de eventos del sistema

No se utiliza el printf porque no se puede utilizar este comando en modo kernel

- Podría explicar que hacen las system call que hemos incluido?

`my_sys_call`: Esta syscall toma un argumento entero (`arg`) y simplemente imprime un mensaje en los registros del sistema usando `printk`. No hace nada más que mostrar el valor de `arg`. Esta syscall es muy simple y se podría usar para probar la funcionalidad básica de las syscalls.

`get_task_info`: Esta syscall obtiene información sobre todos los procesos que están en ejecución. Recorre la lista de procesos con la macro `for_each_process` y, para cada proceso, imprime el PID, el nombre (`comm`), y el estado. Luego, copia esta información a un buffer del espacio de usuario usando `copy_to_user`.

`get_threads_info`: Esta syscall obtiene información sobre todos los procesos y sus hilos. Primero, recorre la lista de procesos con `for_each_process`. Luego, para cada proceso, recorre sus hilos con `for_each_thread`. Imprime el nombre y PID del proceso, y para cada hilo, imprime el nombre y PID del hilo. Finalmente, copia esta información a un buffer del espacio de usuario utilizando `copy_to_user`.

hay que dejar el archivo de la syscall (el .c) a la misma altura que las otras.(kernel/version/kernel)

[illegible]

agregamos al Makefile la syscall

```
obj-y += sched/  
obj-y += locking/  
obj-y += power/  
obj-y += printk/  
obj-y += irq/  
obj-y += rcu/  
obj-y += livepatch/  
obj-y += dma/  
obj-y += entry/  
obj-y += my_sys_call.o
```

el directorio en donde está la tabla con las sys calls es:

arch/x86/entry/syscalls/syscall\_64.tbl

hacemos un nano sobre ese archivo y al final del todo agregamos las syscalls que hicimos

```
464      common  getxattrat      sys_getxattrat  
465      common  listxattrat     sys_listxattrat  
466      common  removexattrat   sys_removexattrat  
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  
#
```

accedemos al archivo en linux-6.13/include/uapi/asm-generic (tabla de manejadores de system calls),y agregamos los punteros a las systems calls que creamos

incluimos en los headers del kernel las syscalls que hicimos

```
__SYSCALL(__NR_listxattrat, sys_listxattrat)  
#define __NR_removexattrat 466  
__SYSCALL(__NR_removexattrat, sys_removexattrat)  
#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
```

y compilamos de nuevo el kernel...

```
LD      arch/x86/boot/setup.elf  
OBJCOPY arch/x86/boot/setup.bin  
BUILD   arch/x86/boot/bzImage  
kernel: arch/x86/boot/bzImage is ready (#2)
```

Si tiramos un grep vemos que efectivamente ahí está la system call

```

so@so:~$ grep get_task_info "/boot/System.map-$(uname -r)"
ffffffff81181d00 t __pfx___do_sys_get_task_info
ffffffff81181d10 t __do_sys_get_task_info
ffffffff81181f20 T __pfx___x64_sys_get_task_info
ffffffff81181f30 T __x64_sys_get_task_info
ffffffff81181f50 T __pfx___ia32_sys_get_task_info
ffffffff81181f60 T __ia32_sys_get_task_info
ffffffff826d9b40 d event_exit__get_task_info
ffffffff826d9bc0 d event_enter__get_task_info
ffffffff826d9c40 d __syscall_meta__get_task_info
ffffffff826d9c80 d args__get_task_info
ffffffff826d9c90 d types__get_task_info
ffffffff82f3aed8 d __event_exit__get_task_info
ffffffff82f3aee0 d __event_enter__get_task_info
ffffffff82f3f2d8 d __p_syscall_meta__get_task_info
so@so:~$

```

y si hacemos un comandito que llame a la sys call... andaaaaa :D

### Monitoreando System Calls

1. Ejecute el programa anteriormente compilado

\$ ./get\_task\_info

Cual es el output del programa?

```

so@so:~/SD$ gcc -o get_task_info get_task_info.c
so@so:~/SD$ ./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: 11 | Nombre: kworker/u8:0 | 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: 0
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
PID: 28
-----
so@so:~/SD$

```

## 2. Luego de ejecutar el programa ahora ejecute

\$ sudo dmesg

¿Cuál es el output? porque?(recuerde printk y lea el man de dmesg)

```
9.283816] snd_intel8x0 0000:00:19.0: allow list rate for 1028:0177 is 48000
46.389462] clocksource: Long readout interval, skipping watchdog check: cs_nsec: 1720125282 wd_nsec: 1720124308
752.236099] clocksource: Long readout interval, skipping watchdog check: cs_nsec: 4328216706 wd_nsec: 4328214568
823.997239] PID: 1 | Nombre: systemd
823.997239] PID: 2 | Nombre: kthreadd
823.997242] PID: 3 | Nombre: pool_workqueue_
823.997244] PID: 4 | Nombre: kworker/R-rcu_g
823.997247] PID: 5 | Nombre: kworker/R-sync_
823.997250] PID: 6 | Nombre: kworker/R-slub_
823.997252] PID: 7 | Nombre: kworker/R-netns
823.997255] PID: 8 | Nombre: kworker/0:0
823.997268] PID: 12 | Nombre: kworker/R-mm_pe
823.997271] PID: 13 | Nombre: rcu_tasks_kthre
823.997273] PID: 14 | Nombre: rcu_tasks_rude_
823.997276] PID: 15 | Nombre: rcu_tasks_trace
823.997278] PID: 16 | Nombre: ksoftirqd/0
823.997280] PID: 17 | Nombre: rcu_preempt
823.997283] PID: 18 | Nombre: rcu_exp_par_gp_
823.997285] PID: 19 | Nombre: rcu_exp_gp_kthr
823.997288] PID: 20 | Nombre: migration/0
823.997290] PID: 21 | Nombre: idle_inject/0
823.997293] PID: 22 | Nombre: cpuhp/0
823.997295] PID: 23 | Nombre: cpuhp/1
823.997298] PID: 24 | Nombre: idle_inject/1
823.997300] PID: 25 | Nombre: migration/1
823.997303] PID: 26 | Nombre: ksoftirqd/1
1376.846531] PID: 1 | Nombre: systemd
1376.846536] PID: 2 | Nombre: kthreadd
1376.846539] PID: 3 | Nombre: pool_workqueue_
1376.846542] PID: 4 | Nombre: kworker/R-rcu_g
1376.846544] PID: 5 | Nombre: kworker/R-sync_
1376.846547] PID: 6 | Nombre: kworker/R-slub_
1376.846549] PID: 7 | Nombre: kworker/R-netns
1376.846552] PID: 8 | Nombre: kworker/0:0
1376.846554] PID: 12 | Nombre: kworker/R-mm_pe
1376.846556] PID: 13 | Nombre: rcu_tasks_kthre
1376.846559] PID: 14 | Nombre: rcu_tasks_rude_
1376.846561] PID: 15 | Nombre: rcu_tasks_trace
1376.846564] PID: 16 | Nombre: ksoftirqd/0
1376.846566] PID: 17 | Nombre: rcu_preempt
1376.846568] PID: 18 | Nombre: rcu_exp_par_gp_
1376.846570] PID: 19 | Nombre: rcu_exp_gp_kthr
1376.846573] PID: 20 | Nombre: migration/0
1376.846576] PID: 21 | Nombre: idle_inject/0
1376.846578] PID: 22 | Nombre: cpuhp/0
1376.846580] PID: 23 | Nombre: cpuhp/1
1376.846583] PID: 24 | Nombre: idle_inject/1
1376.846585] PID: 25 | Nombre: migration/1
1376.846587] PID: 26 | Nombre: ksoftirqd/1
```

El comando sudo dmesg muestra los mensajes del kernel que han sido registrados desde que se inició el sistema. Esto incluye:

Mensajes de inicio del sistema (boot),

Información de dispositivos conectados (como USB),

Errores o advertencias del hardware o del sistema,

Mensajes generados mediante la función printk() desde el kernel o desde módulos del kernel cargado

Es un log del kernel, es lo que la syscall que definimos imprime en el log kernel (no utilizar sudo, no lo reconoce, entrar en modo admin con “su”)

## 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 0x400, 0xaaabe110740,

strace es una herramienta que intercepta y muestra las llamadas al sistema (syscalls) que hace un programa mientras se ejecuta. Es súper útil para depurar o entender qué está haciendo un programa en interacción con el sistema operativo.

Si luego ejecuto

# echo \$((0x1C4))

- ¿Qué valor obtengo? porque?

```
root@so:/home/so/SO# echo $((0x1D4))  
468  
root@so:/home/so/SO#
```

Este número representa el número de syscall. Cada syscall en Linux tiene un número asignado. Por ejemplo: 69



## Módulos y drivers

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?

Se llaman módulos de kernel. No es necesario ni recompilar ni rootear el sistema para instalarlos. se instalan dinámicamente. Si no se pudiera usarlos, habría que recompilar el kernel agregando el código directamente y reiniciar para aplicar los cambios.

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

Es software que le provee al SO información sobre cómo utilizar un periférico (disco, monitor, etc)

3. ¿Por qué es necesario escribir drivers?

Porque cada tipo de hardware tiene características distintas, y el sistema operativo necesita saber cómo comunicarse correctamente con él.



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

Un driver se implementa como un módulo del kernel. Los módulos son la forma en que el software se carga directamente en el kernel.

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

Un bug en un driver o módulo supone un gran riesgo ya que estos fragmentos de software tienen acceso al espacio de direcciones del kernel, autos problemas que se pueden producir son :Bloqueo del sistema (kernel panic), fallas en dispositivos, pérdida de datos

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

- Dispositivos de bloque: grupo de bloques de datos persistentes. Se lee y escribe de a bloques.
- Dispositivos de carácter: se accede de a un byte a la vez. 1 byte se lee una única vez

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

En el directorio /dev se encuentran todos los dispositivos asociados al SO, identificados por el minor y mayor number. Dentro se encuentra el /dev/hda un archivo que en realidad no es un archivo sino la información contenida en el disco en sí, en otras palabras, operar aquí produce cambios directos en el disco

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

En ese archivo quedan escritas por defecto las dependencias que hay que respetar al cargar y descargar módulos. modprobe lo usa para saber qué otros módulos necesita cargar antes o junto al módulo solicitado.

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

Es un sistema de archivos temporal que se monta durante el arranque del sistema que contiene ejecutables, drivers y módulos necesarios para lograr iniciar el sistema. Luego del proceso de arranque el disco se desmonta. También se actualiza cuando se ejecutan comandos como update-initramfs o mkinitcpio

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

Los necesarios para:

Acceder al dispositivo de arranque (por ejemplo: driver de disco, driver de sistema de archivos)

Cargar el sistema de archivos raíz

Montar rootfs y continuar el proceso de booteo

Módulos y drivers mínimos del initramfs:

Driver del sistema de archivos – Para montar / (ej: ext4).

Driver del almacenamiento – Para acceder al disco (ej: ahci, nvme, usb-storage).

Controladores del bus – Para comunicar con el hardware (ej: pci, usb, scsi).

Soporte RAID/LVM – Si se usa (ej: md\_mod, dm\_mod).

Soporte de cifrado – Si hay particiones cifradas (ej: dm-crypt, aes).

Herramientas básicas – Para ejecutar scripts de arranque (ej: busybox, init, sh).

## Práctica guiada:

Desarrollando un módulo simple para Linux

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

```
#include  
MODULE_LICENSE("Dual BSD/GPL");
```

a. Explique brevemente cual es la utilidad del archivo `Makefile`.

`Makefile` se usa para definir cómo compilar el módulo para que sea compatible con el kernel.

b. ¿Para qué sirve la macro `MODULE_LICENSE`? ¿Es obligatoria?

Le dice al kernel bajo qué licencia está el módulo. Si no se incluye, el kernel muestra un warning y algunas funciones internas del kernel no son disponibles. No es obligatorio, pero recomendable

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.

a. ¿Cuál es la salida del comando anterior?

```
get_task_info.c Makefile memory.c my_sys_call.c  
so@so:~/SO$ 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/SO'  
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/SO'  
make: se sale del directorio '/home/so/kernel/linux-6.13'  
so@so:~/SO$
```

(fabri: a mi me salio esto pero funcó xd)

```
so@so:~/practica2$ make -C /lib/modules/$(uname -r) M=$(pwd) modules  
make: se entra en el directorio '/usr/lib/modules/6.13.7'  
make: *** No hay ninguna regla para construir el objetivo 'modules'. Alto.  
make: se sale del directorio '/usr/lib/modules/6.13.7'  
so@so:~/practica2$ ls  
Makefile memory.c memory.ko memory.mod memory.mod.c memory.mod.o memory.o modules.or
```

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

-`memory.ko`: (Kernel Object) Es el módulo compilado que se puede insertar en el kernel con `insmod`.

-`memory.o`: Archivo objeto intermedio compilado desde `memory.c`. Es usado para construir `.ko`.

-`memory.mod.c`: Código generado automáticamente por `make` para gestionar el módulo dentro del kernel.

-memory.mod.o: Archivo objeto compilado a partir de memory.mod.c.

-modules.order: Lista de módulos que se compilaron en orden, usado por make para mantener consistencia

-module.symvers: Contiene símbolos exportados por tu módulo, útil si otros módulos los usan.

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

make, nuestro módulo se compila

make clean, limpia el módulo y el código objeto generado

make run, ejecuta el programa

# Nombre del módulo (sin extensión)

MODULE\_NAME := memory

# Compilación del módulo

obj-m := \$(MODULE\_NAME).o

# Ruta al build del kernel

KDIR := /lib/modules/\$(shell uname -r)/build

PWD := \$(shell pwd)

all:

make -C \$(KDIR) M=\$(PWD) modules

clean:

make -C \$(KDIR) M=\$(PWD) clean

rm -f \*.ko \*.mod.c \*.o \*.symvers \*.order

run: all

sudo insmod \$(MODULE\_NAME).ko

@echo "Módulo cargado:"

lsmod | grep \$(MODULE\_NAME)

4. El paso que resta es agregar y eventualmente quitar nuestro módulo al kernel en tiempo de ejecución.

Ejecutamos:

# insmod memory.ko

export PATH=\$PATH:/sbin:/usr/sbin

tuvimos que agregar la variable de arriba para que encuentre el insmod

a. Responda lo siguiente:

b. ¿Para qué sirven el comando insmod y el comando modprobe? ¿En qué se diferencian?

- Insmod: Carga directamente un módulo (.ko) al kernel. Solo inserta el archivo especificado, sin verificar dependencias.
- modprobe: También carga un módulo al kernel, pero busca dependencias automáticamente. Usa los archivos de configuración en /lib/modules/\$(uname -r)/modules.dep para saber qué más debe cargar. Busca el módulo en los directorios estándar del sistema (/lib/modules/...).

5. Ahora ejecutamos:

\$ lsmod | grep memory

Responda lo siguiente:

a. ¿Cuál es la salida del comando? Explique cuál es la utilidad del comando lsmod.

La salida del comando es:

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

- memory: el nombre del módulo cargado (debería coincidir con el que definiste en el código fuente del módulo).
- 8192: tamaño en rmmbytes del módulo cargado.
- 0: cantidad de procesos/modulos que están usando este módulo (es decir, no tiene dependencias activas ahora mismo).

su función principal es agregar un módulo al kernel de Linux en tiempo de ejecución

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

Es un archivo virtual que lista todos los módulos del kernel que están actualmente cargados

c. Si ejecutamos more /proc/modules encontramos los siguientes fragmentos ¿Qué información obtenemos de aquí?:

muestra información sobre los módulos del kernel actualmente cargados en el sistema. Este archivo pertenece al sistema de archivos procfs, que expone información sobre el sistema y el kernel en forma de archivos virtuales. Cada línea del archivo /proc/modules representa un módulo cargado, y contiene campos con detalles específicos.

Este archivo lista los módulos del kernel cargados en memoria, mostrando su tamaño, dependencias, estado y dirección de memoria.

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

Para descargar (o eliminar) un módulo del kernel en Linux, se utiliza el comando rmmod o, preferiblemente, modprobe -r.

ex6. Descargue el módulo memory. Para corroborar que efectivamente el mismo ha sido eliminado del kernel ejecute el siguiente comando: lsmod | grep memory

```
memory                8192  0
root@so:/home/so/SD# rmmod memory
root@so:/home/so/SD# lsmod | grep memory
root@so:/home/so/SD#
```

7. Modifique el archivo memory.c de la siguiente manera:

```
#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);
```

## 8. Responda lo siguiente:

a. ¿Para qué sirven las funciones `module_init` y `module_exit`? ¿Cómo haría para ver la información del log que arrojan las mismas?.


-`module_init`: Especifica la función que se ejecutará al cargar el módulo (ej: `hello_init`).  
 -`module_exit`: Especifica la función que se ejecutará al descargar el módulo (ej: `hello_exit`).  
 Para ver los logs: Usar `dmesg` o consultar `/var/log/kern.log` (dependiendo de la distribución).


b. 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?.


controlar un dispositivo xd

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

Dispositivos de caracteres (Character Devices)

 Se accede carácter por carácter, es decir, byte a byte.

 No se puede hacer "seek" (no puedes moverte a una parte específica del archivo).

 Se usan para dispositivos que envían o reciben datos en forma de flujo continuo.

Dispositivos de bloques (Block Devices)

 Se accede en bloques (grupos de bytes, normalmente de 512 bytes o más).

 Permiten "seek" (puedes moverte a cualquier parte del dispositivo para leer o escribir).

 Usados para almacenar archivos o particiones enteras

## Desarrollando un driver

### 2. Responda lo siguiente:

a. ¿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 definido en POSIX, usado para representar el tamaño de un bloque de memoria o el número de bytes transferidos. Es igual que `size_t`, pero con signo, lo que permite devolver errores (por ejemplo, -1) en funciones como `read()` o `write()`.

`memory_fops`:

Es una estructura de tipo `file_operations` que define las funciones que un driver utilizará para manejar operaciones estándar sobre archivos (abrir, leer, escribir, cerrar, etc.). En un driver de carácter, esta estructura se asocia al dispositivo, y sus miembros son punteros a funciones que implementan la lógica específica del driver.

`register_chrdev`:

Esta función registra un controlador de dispositivo de carácter con el kernel. Toma como argumentos el número mayor (major number), el nombre del dispositivo, y un puntero a la estructura `file_operations`. Devuelve un número de dispositivo (major) si no se especifica uno explícitamente.

`unregister_chrdev`:

Libera el número mayor previamente registrado con `register_chrdev`, eliminando así la asociación entre el número de dispositivo y el driver.

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

El kernel sabe qué funciones del driver invocar gracias a la estructura `file_operations`, como `memory_fops`. Cuando el usuario abre un archivo especial (como `/dev/mydevice`), el kernel busca el número mayor del dispositivo y lo asocia con la estructura `file_operations` correspondiente.

Así, cuando el usuario ejecuta una operación como `read()` o `write()`, el kernel invoca los punteros a función definidos en esa estructura (`read`, `write`, etc.).

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

Desde el espacio de usuario, se accede a los dispositivos a través de archivos ubicados en el directorio `/dev`, como `/dev/sda` o `/dev/mydevice`. Estos archivos especiales son interfaces hacia los drivers del kernel.

Las llamadas al sistema estándar como `open()`, `read()`, `write()`, `ioctl()`, etc., se utilizan para interactuar con ellos.

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

Cuando ves un archivo de dispositivo en `/dev`, ese archivo está asociado con dos números:

1. Major Number, el que identifica el driver del dispositivo

Le dice al kernel qué módulo o driver debe manejar ese dispositivo.

2. Minor Number, identifica un dispositivo específico que ese driver maneja.

Por ejemplo, el disco `/dev/sda` puede tener particiones `/dev/sda1`, `/dev/sda2`, etc., cada una con su minor diferente.

La asociación entre el driver y el dispositivo puede hacerse de varias formas, dependiendo del tipo de dispositivo:

Para drivers de carácter simple, se hace mediante el registro de un número mayor con `register_chrdev`, y luego se crea un archivo en `/dev` con ese número mayor (por ejemplo, usando `mknod`).

Para dispositivos más complejos, como dispositivos conectados por PCI o USB, se usa una estructura de identificación (como `pci_device_id` o `usb_device_id`) que se compara con los dispositivos conectados. El kernel asocia automáticamente el driver con el dispositivo si hay coincidencia.

e. ¿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 transferir datos entre el espacio del kernel y el espacio de usuario, ya que el acceso directo entre ambos no está permitido por seguridad.

`copy_to_user(void __user *to, const void *from, unsigned long n)`

Copia `n` bytes desde el espacio del kernel al espacio de usuario.

`copy_from_user(void *to, const void __user *from, unsigned long n)`

Copia `n` bytes desde el espacio de usuario al espacio del kernel.

a. Responda lo siguiente:

i. ¿Para qué sirve el comando `mknod`? ¿qué especifican cada uno de sus parámetros?.

👉 `mknod` se usa para crear archivos especiales de dispositivo en `/dev`.

`mknod [nombre_archivo] [tipo] [major] [minor]`

Parámetros:

`nombre_archivo` → cómo se llamará el archivo en `/dev`, por ejemplo: `memory`

`tipo` → tipo de dispositivo:

`c` → dispositivo de carácter

`b` → dispositivo de bloque

`major` → número mayor → identifica el driver o módulo

`minor` → número menor → identifica el dispositivo específico dentro del driver

ii. ¿Qué son el “major” y el “minor” number? ¿Qué referencian cada uno?

El `major` indica qué controlador del kernel, el driver, maneja el dispositivo.

El `minor` distingue entre múltiples dispositivos gestionados por el mismo driver.

5. Ahora escribimos a nuestro dispositivo:

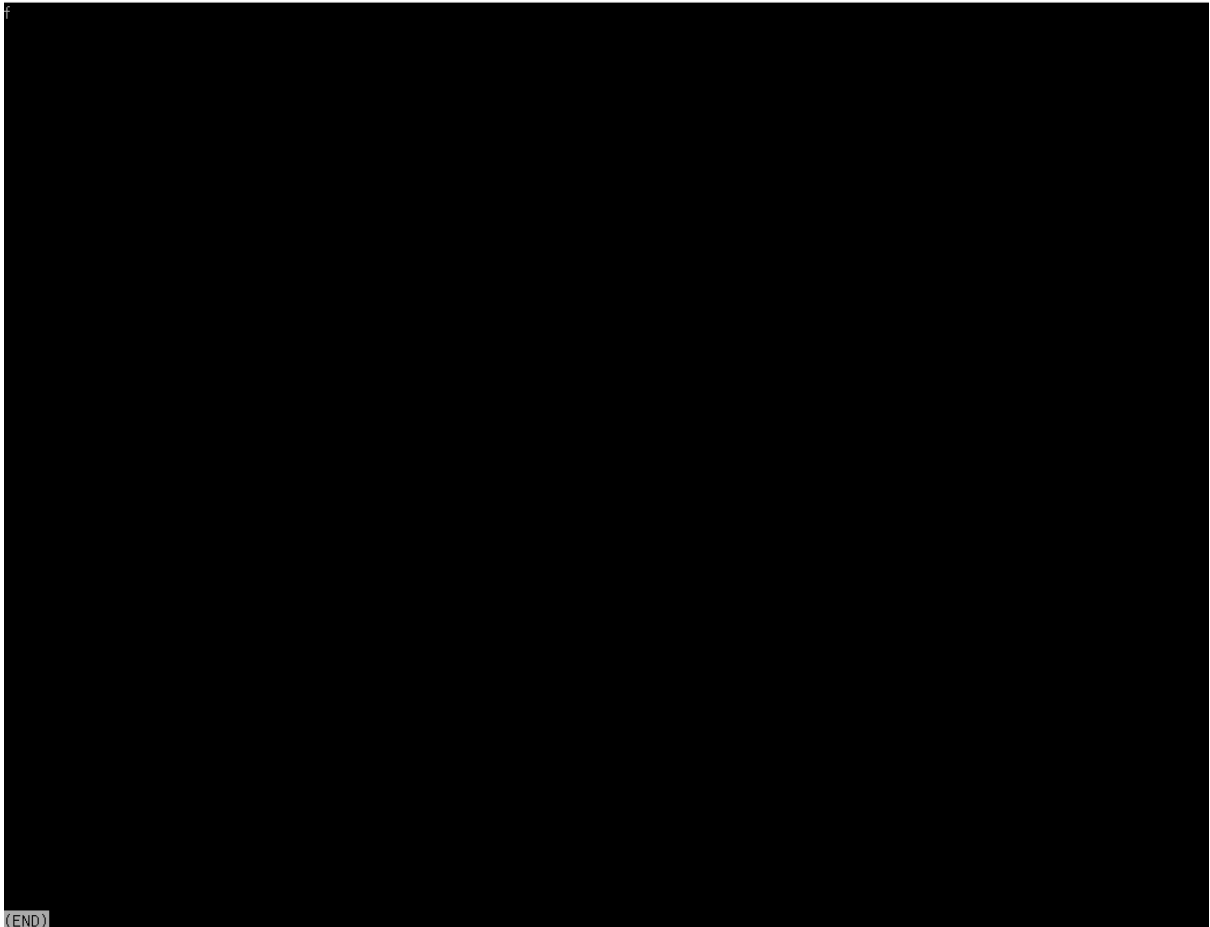
`echo -n abcdef > /dev/memory`

6. Ahora leemos desde nuestro dispositivo:

`more /dev/memory`

7. Responda lo siguiente:

a. ¿Qué salida tiene el anterior comando?, ¿Porque? (ayuda: siga la ejecución de las funciones `memory_read` y `memory_write` y verifique con `dmesg`)



Imprime solo f porque `memory_write` solo guarda 1 byte.

b. ¿Cuántas invocaciones a `memory_write` se realizaron?

Solo una vez por ejecución del echo.

Aunque echo parece que manda todo el string, el kernel lo envía en una sola llamada a write, por lo que `memory_write()` se invoca una vez.

c. ¿Cuál es el efecto del comando anterior? ¿Por qué?

se guarda solo el carácter 'f' en el buffer del driver.

d. 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.

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`.

Sincronización:

Para evitar conflictos si varios procesos acceden al mismo tiempo (se usan semáforos o mutexes).

`IOCTL`:

Para permitir comandos especiales desde el usuario (ej: cambiar modo del dispositivo).

Acceso a hardware:

Se usan funciones como `inb()` y `outb()` para leer/escribir en puertos de entrada/salida.



Interrupciones:

Para reaccionar a eventos del hardware (como conexión/desconexión de dispositivos).

Manejo de múltiples dispositivos:

Usando minor numbers para distinguir entre varios dispositivos controlados por el mismo driver.