Práctica 2 - Módulos, drivers y syscalls

System Calls

Conceptos generales

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

Es un mecanismo para que un programa de usuario solicite servicios al sistema operativo. Actúa como interfaz entre los procesos en modo usuario y el SO que se ejecuta en modo supervisor. Esta separación protege el sistema de programas maliciosos.

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

La macro syscall permite que se ejecuten llamadas al sistema desde programas escritos con C o ensamblador sin usar wrappers.

La llamada tiene 7 parámetros:

1. Number: Numero de la syscall, definidos en un archivo y son unívocos, por ejemplo 1 para write.
2. arg1-arg7: Coinciden con los registros de una arquitectura x86-64, acá va la información necesaria para hacer la syscall, cada uno representa algo distinto en una syscall especifica, de necesitar mas parámetros se puede pasar un puntero a un arreglo o cosas así.

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

`ls -lh /boot | grep vmlinuz`

```
root@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
```

Son kernels de linux compilados para distintas versiones, esto sirve por si falla algún kernel nuevo o algo así siempre se puede volver a funciones anteriores.

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`

`._.xd`:

```
root@so:/home/so# git clone https://github.com/torvalds/linux.git
-bash: git: orden no encontrada
```

5. ¿Para qué sirve el siguiente archivo?

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

El archivo `syscall_64.tbl` en la ruta `arch/x86/entry/syscalls/` del código fuente del kernel de Linux es una **tabla de llamadas al sistema (system calls)** para arquitecturas **x86_64** (64 bits). Su propósito es fundamental en el sistema operativo, ya que mapea números de syscall con sus implementaciones.
Formato:
`60 common exit sys_exit`

- `60`: Número de syscall.
- `common`: ABI (Application Binary Interface) compatible (en este caso, común a 32 y 64 bits).
- `exit`: Nombre de la syscall (usado por programas en espacio de usuario).
- `sys_exit`: Función en el kernel que la implementa.

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

`strace` es una herramienta de diagnóstico y depuración en Linux que **rastrea las llamadas al sistema (system calls)** y las **señales** que un proceso recibe o envía durante su ejecución. Es esencial para:

- **Depurar programas**: Ver qué operaciones fallan (ej: errores al abrir archivos).
- **Analizar el comportamiento** de aplicaciones o scripts.
- **Entender cómo interactúa un proceso** con el kernel (archivos, red, memoria, etc.).
  `strace [opciones] [comando]`

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

`ausyscall` es una herramienta incluida en el paquete `auditd` de Linux que **mapea números de llamadas al sistema (syscalls) con sus nombres** y viceversa. Está diseñada para trabajar con el sistema de auditoría **auditd**, que registra eventos de seguridad en el kernel.
Sirve para:

1. **Traducir números de syscall a nombres legibles**:
   - Útil al analizar logs de `auditd` o `strace`, donde las syscalls aparecen como números (ej: `59` → `execve`).
2. **Resolver diferencias entre arquitecturas**:
   - Los números de syscall varían según la arquitectura (x86, x86_64, arm, etc.). `ausyscall` ajusta la salida según la arquitectura del sistema.
3. **Facilitar la depuración y análisis forense**:
   - Ayuda a interpretar registros de auditoría o rastreos de syscalls sin memorizar números.
   `ausyscall [opciones] [número|nombre]`

Práctica guiada

Simplemente seguí la guía, a tener en cuenta que la carpeta kernel/ es la que esta en home/so/kernel/linux-6.13/kernel.
Además hay que cambiar los números de las syscalls.
Pregunta teóricas random que aparecian:
Mirando el código anterior, investigue y responda lo siguiente?

● ¿Para qué sirven los macros SYS_CALL_DEFINE?

Los macros `SYSCALL_DEFINE` (como `SYSCALL_DEFINE1`, `SYSCALL_DEFINE2`, etc.) son utilizados en el **kernel de Linux** para **declarar e implementar nuevas llamadas al sistema (syscalls)**

- El número indica la cantidad de argumentos que acepta la syscall. Por ejemplo:
  - `SYSCALL_DEFINE1(my_sys_call, int, arg)` define una syscall llamada `my_sys_call` que recibe un argumento de tipo `int`.
  - `SYSCALL_DEFINE2(get_task_info, char __user *, buffer, size_t, length)` define una syscall con dos argumentos.
  Este macro se encarga de registrar correctamente la función dentro del sistema, asignarle un nombre y habilitar el paso de argumentos desde el espacio de usuario al kernel.

● ¿Para que se utilizan la macros for_each_process y for_each_thread?

- `for_each_process(task)`:
  - Recorre **todos los procesos** del sistema.
  - Por cada iteración, `task` apunta a una estructura `task_struct` que representa un proceso.
- `for_each_thread(task, thread)`:
  - Recorre **todos los hilos** (threads) del proceso representado por `task`.
  - `thread` apunta a cada hilo individual del proceso.

● ¿Para que se utiliza la función copy_to_user?

- `copy_to_user(dest_user_ptr, src_kernel_ptr, size)` copia datos **desde el espacio de memoria del kernel al espacio de usuario**.
- Es crucial para devolver resultados a programas que invocan la syscall desde user-space.
- En el kernel, **acceder directamente a punteros de usuario está prohibido** (por motivos de seguridad), así que se debe usar funciones como `copy_to_user` o `copy_from_user`.

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

- `printk` es la función del kernel para imprimir mensajes al **log del sistema** (usualmente accesible vía `dmesg`).

- **No se puede usar** `printf` en el kernel porque:
  - `printf` es parte de la biblioteca estándar de C (`libc`), la cual **no está disponible en el kernel**.
  - El kernel no tiene acceso a funciones de I/O de alto nivel.
  `printk` permite imprimir mensajes con niveles de prioridad como `KERN_INFO`, `KERN_WARNING`, `KERN_ERR`, etc., para ayudar a la depuración.

• Podría explicar que hacen las sytem call que hemos incluido?

- `my_sys_call(int arg)`
  - Syscall básica de ejemplo.
  - Imprime el argumento recibido con `printk` y devuelve 0.
  - Es útil para aprender cómo declarar e invocar syscalls.
- `get_task_info(char __user *buffer, size_t length)`
  - Recorre todos los procesos del sistema.
  - Para cada proceso, obtiene el PID, nombre (`comm`) y estado (`task_state_index`).
  - Los datos se escriben en un buffer del kernel y luego se copian al espacio de usuario.
  - Ideal para obtener un resumen del estado actual del sistema.
- `get_threads_info(char __user *buffer, size_t length)`
  - Recorre todos los procesos y sus hilos.
  - Para cada proceso, lista todos sus hilos (nombre + TID).
  - Usa `kmalloc` para crear un buffer dinámico (más grande que en la syscall anterior).
  - También devuelve la información al usuario mediante `copy_to_user`.

Que imprime `sudo dmesg`:

Muestra los mensajes del **buffer del anillo del kernel (kernel ring buffer)**, es decir, todos los mensajes generados por el kernel desde el arranque del sistema. Esto incluye

- Mensajes de **inicialización del hardware**
- **Logs del sistema** (drivers, módulos, errores, etc.)
- Salidas de `printk()` desde el código del kernel (como las que usás en tu syscall)

Que imprime `strace get_task_info`:

Intenta rastrear (trace) las **llamadas al sistema** realizadas por el programa ejecutable `get_task_info`, el cual hace uso de la syscall anteriormente implementada que consigue la información de los procesos.

Módulos y Drivers

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?
2. ¿Qué es un driver? ¿Para qué se utiliza?
3. ¿Por qué es necesario escribir drivers?
4. ¿Cuál es la relación entre módulo y driver en GNU/Linux?
5. ¿Qué implicancias puede tener un bug en un driver o módulo?
6. ¿Qué tipos de drivers existen en GNU/Linux?
7. ¿Qué hay en el directorio /dev? ¿Qué tipos de archivo encontramos en esa ubicación?
8. ¿Para qué sirven el archivo /lib/modules//modules.dep utilizado por el comando modprobe?
9. ¿En qué momento/s se genera o actualiza un initramfs?
10. ¿Qué módulos y drivers deberá tener un initramfs mínimamente para cumplir su objetivo?

---

1. **Módulo del kernel**. No, no requiere reinicio. Sin módulos: recompilar el kernel con la funcionalidad integrada.
2. **Driver**: Software que permite al sistema operativo interactuar con hardware específico.
3. **Porque el kernel no puede conocer todo el hardware posible**; cada dispositivo necesita su código de control.
4. **Todo driver es un módulo (si es cargable), pero no todo módulo es un driver**.
5. **Un bug puede causar cuelgues, corrupción de datos o vulnerabilidades críticas**.
6. **De dispositivo (char, block), de red, de sistema de archivos, pseudo-dispositivos, etc.**
7. **Interfaces a dispositivos**. Contiene archivos especiales: **carácter, bloque, simbólicos, pipes, etc.**
8. **Define dependencias entre módulos** para que `modprobe` cargue los necesarios automáticamente.
9. **Al instalar kernel nuevo o actualizar módulos**. También con `update-initramfs` o `mkinitcpio`.
10. **Drivers esenciales para montar el sistema raíz**: discos, controladores SATA/NVMe, sistemas de archivos.

Práctica guiada:

2

a. Utilidad del archivo `Makefile`:
Indica a `make` cómo compilar el módulo del kernel (`memory.c`), generando el archivo `.ko`.
Específicamente, `obj-m := memory.o` declara que `memory.o` es un módulo que debe ser compilado.

b. ¿Para qué sirve `MODULE_LICENSE`? ¿Es obligatoria?:
Declara la licencia del módulo.
Permite al kernel saber si puede cargarlo sin marcarlo como "**taint**" (manchado).
No es estrictamente obligatoria, **pero si falta, el kernel lo marca como propietario** y puede restringir el acceso a funciones internas.

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
```

¿Que quiere decir exactamente?:
Yo escribí:
`make -C ../linux-6.13/ M=$(pwd) modules`

`..` Porque mi codigo estaba en kernel/program/ entonces tenía que salir un nivel para llegar al codigo del kernel, cuando termino me creo 300 archivos .o, .mod, .order, .cmd, . memory.
Rarísimo.

a. ¿Cuál es la salida del comando anterior?

- `-C <KERNEL_CODE>`: le dice a `make` que cambie al directorio con el código fuente del kernel (`<KERNEL_CODE>` debería reemplazarse por el path real, como `/usr/src/linux-headers-$(uname -r)`).
- `M=$(pwd)`: le dice que compile el módulo que se encuentra en el directorio actual (el que contiene `memory.c` y el `Makefile`).
- `modules`: indica que se quieren construir módulos del kernel.

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

Archivo	Descripción
`memory.o`	Código objeto compilado del archivo fuente (`memory.c`).
`memory.mod.o`	Código objeto extendido con metainformación del módulo.
`memory.ko`	**Kernel Object**, el archivo del **módulo final que se puede cargar en el kernel** con `insmod`.
`Module.symvers`	Contiene símbolos exportados por el módulo y sus dependencias. Útil si otros módulos dependen de este.
`modules.order`	Indica el orden en el que deben cargarse los módulos.

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

i. make, nuestro módulo se compila
ii. make clean, limpia el módulo y el código objeto generado
iii. make run, ejecuta el programa
Lo hizo gpt:

```
# Nombre del módulo
obj-m := memory.o
```

```
# Ruta al código fuente del kernel
KDIR := /lib/modules/$(shell uname -r)/build

# Directorio actual
PWD := $(shell pwd)

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

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

# Carga el módulo al kernel
run: all
    sudo insmod memory.ko
    dmesg | tail -n 10
```

4

a. Responda lo siguiente: *Ya voy amigo, no sé porque es una pregunta esto*
b. ¿Para qué sirven el comando insmod y el comando modprobe? ¿En qué se diferencian?
El mismo carga un modulo de kernel o driver, pero no resuelve dependencias como modprobe.

5

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

La salida fue `memory 8192 0`, muestra los modulos cargados

---

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

Este archivo contiene información en tiempo real sobre los **módulos cargados en el kernel**, y es lo que `lsmod` usa internamente. Cada línea representa un módulo y contiene datos como:
`nombre_tamaño número_usuarios otros_módulos estado dirección_memoria`

---

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

```
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
```


```
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
```


¿Qué significa cada columna?

1. **Nombre del módulo**
2. **Tamaño (en bytes)**
3. **Número de instancias que lo están usando**
4. **Dependencias (otros módulos que dependen de él)**
5. **Estado** (Live = activo)
6. **Dirección en memoria**
7. *(Opcional)* **Marcas especiales**, como `(OE)`

Análisis línea por línea:

- `memory 8192 0 - Live 0x0000000000000000 (OE)`

- `memory`: nombre del módulo (probablemente un módulo personalizado o académico, ya que no es estándar).
- `8192`: ocupa 8 KB.
- `0`: nadie lo está usando.
- `-`: no tiene dependencias de otros módulos.
- `Live`: está activo.
- `0x...`: dirección de memoria, aquí se muestra como `0x0000000000000000`, lo cual podría ser indicativo de un módulo de prueba o no mapeado.
- `(OE)`: indica que **fue cargado desde fuera del árbol oficial del kernel** (Out-of-tree + External). Es decir, no forma parte de los módulos que vienen por defecto en el kernel.

- `binfmt_misc 24576 1 - Live 0x0000000000000000`

- Permite que el sistema Linux ejecute archivos que no son binarios ELF tradicionales (como scripts de otras plataformas).
- Tiene 1 usuario.
- Sin dependencias.
- Está activo.

- `intel_rapl_msr 16384 0 - Live 0x0000000000000000`

- Módulo para manejo de energía de procesadores Intel (RAPL = Running Average Power Limit).
- No está siendo usado directamente.
- Activo

- `intel_rapl_common 32768 1 intel_rapl_msr, Live 0x0000000000000000`

- Módulo base compartido por otros de la familia RAPL.
- Está siendo usado por `intel_rapl_msr`.
- Activo.

---

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

`rmmod memory.ko`

6

No sale más.

7

Las 2 ultimas líneas son Hello world y Bye Cruel world.

---

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?.
Las funciones se llaman cuando se cargan o descargan modules.

---

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?.
Toda la funcionalidad especifica y una interfaz con el usuario.

---

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

Tipo	Acceso	Ejemplos	Uso principal	Interfaz
Carácter	Byte a byte	`/dev/tty`, teclado	Datos en tiempo real	`/dev`, `read`, `write`
Bloque	Por bloques	`/dev/sda`, USB	Almacenamiento	`/dev`, permite `seek`
Red	Por sockets	`eth0`, `wlan0`	Comunicación en red	Pila TCP/IP (no `/dev`)

Driver

2. Preguntas sobre drivers en Linux

a. ¿Para qué sirve `ssize_t` y `memory_fops`?

- `ssize_t`: tipo de dato usado para devolver la **cantidad de bytes leídos o escritos** (o error negativo) en funciones como `read` y `write` del driver.
- `memory_fops`: estructura `file_operations` con los **punteros a funciones del driver** (open, read, write, etc.) asociadas al dispositivo.

¿Y `register_chrdev` y `unregister_chrdev`?

- `register_chrdev`: **registra un dispositivo de carácter** en el sistema y lo vincula con las operaciones (`file_operations`).
- `unregister_chrdev`: **elimina** ese registro, liberando el número de dispositivo.

---

b. ¿Cómo sabe el kernel qué funciones del driver invocar para leer/escribir?

El kernel usa la estructura `file_operations` (ej: `memory_fops`) registrada al cargar el módulo. Cuando se accede al dispositivo (`/dev/xxx`), el kernel consulta esa estructura para saber qué función ejecutar (`read`, `write`, etc.).

---

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

A través de **archivos en** `/dev`, usando llamadas estándar como:

- `open("/dev/xxx")`
- `read()`, `write()`, `ioctl()`

---

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

Mediante:

1. Registro con `register_chrdev` (o `cdev_add`).
2. Asignación de número mayor (`major`) y menor (`minor`).
3. Creación del archivo en `/dev/` (con `mknod` o `udev`).
4. Asociación de la estructura `file_operations`.

---

e. ¿Qué hacen `copy_to_user` y `copy_from_user`?

- `copy_to_user`: copia **desde el kernel al espacio de usuario**.
- `copy_from_user`: copia **desde el espacio de usuario al kernel**.

Se usan en `read` y `write` para mover datos entre procesos de usuario y el driver, de forma segura.

4

i. ¿Para qué sirve el comando mknod? ¿qué especifican cada uno de sus parámetros?.
El comando `mknod` se usa para **crear un archivo de dispositivo en** `/dev/`, que representa un driver cargado en el kernel.

- `/dev/memory`: nombre del archivo de dispositivo.
- `c`: tipo de dispositivo → **carácter** (`b` sería bloque).
- `60`: **número mayor (major)** → identifica al **driver** asociado.
- `0`: **número menor (minor)** → identifica al **dispositivo específico** dentro del driver (si hay más de uno).
  ii:
- **Major number**: número que indica **qué módulo o driver** manejará las operaciones sobre el dispositivo. Lo asocia con las funciones en `file_operations`.
- **Minor number**: identifica **un dispositivo particular** manejado por ese driver. Permite que un mismo driver maneje varios dispositivos distintos.

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)

f

b. ¿Cuántas invocaciones a memory_write se realizaron?

Imprimió al menos 6.

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

La función `memory_write()` **copia solo el último byte** del buffer de usuario (`buf`) a un buffer del kernel (`memory_buffer`).

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.

e. 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.

Aspectos Críticos Omitidos en el Driver (Versión Ultra-Sintética)

1. **Sincronización**: Usar *mutex/spinlocks* para evitar race conditions.
2. **Errores**: Retornar `-EFAULT` (no `0`) si falla `copy_from_user`.
3. **Hardware**:
   - Acceso a puertos con `inb()` / `outb()`.
   - DMA para transferencias rápidas.
4. **ioctl**: Para operaciones especiales (ej: reiniciar dispositivo).
5. **Bloqueo**: Verificar `O_NONBLOCK` en `filp->f_flags`.
6. **Interrupciones**: Registrar ISR con `request_irq()`.
7. **Poll/Select**: Implementar `.poll` para notificar eventos.

Conclusión: El ejemplo es demasiado básico; un driver real necesita manejar concurrencia, hardware y eventos.