UNIVERSIDAD NACIONAL DE LA PLATA

Sistemas Operativos

Práctica 2 (2025)

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

https://www.kernel.org/doc/html/latest/process/adding-syscalls.html

Linux Kernel Hacking: A Crash Course - Speaker Deck

System Calls

Conceptos generales

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

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

   ```
   ls -lh /boot | grep vmlinuz
   ```

4. Acceda al codigo 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?
   a. `arch/x86/entry/syscalls/syscall_64.tbl`
6. ¿Para qué sirve la herramienta `strace`? ¿Cómo se usa?
7. ¿Para qué sirve la herramienta `ausyscall`? ¿Cómo se usa?

Práctica guiada

La System Calls que vamos a implementar accederán a la estructura `task_struct` 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 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

```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);
}

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?

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   commonlandlock_create_ruleset sys_landlock_create_ruleset
445   commonlandlock_add_rule sys_landlock_add_rule
446   commonlandlock_restrict_self  sys_landlock_restrict_self
447   commonmemfd_secret      sys_memfd_secret
448   commonprocess_mrelease  sys_process_mrelease
449   commonfutex_waitv       sys_futex_waitv
450   commonset_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
```

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
```

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.
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

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
```

- **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

Monitoreando System Calls

1. Ejecute el programa anteriormente compilado

```
$ ./get_task_info
```

Cual es el output del programa?

2. Luego de ejecutar el programa ahora ejecute

```
$ sudo dmesg
```

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

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, 0x400, 0xaaaabe110740, 0xffff9cc790c0, 0xbd2cc5d5aef6ff14, 0xffff9cc22078) = 0x400

Si luego ejecuto

```
# echo $((0x1C4))
```

- ¿Qué valor obtengo? porque?

Módulos y Drivers

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?
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/<version>/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?

Práctica guiada

Desarrollando un módulo simple para Linux

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:

   a. Explique brevemente cual es la utilidad del archivo Makefile.
   b. ¿Para qué sirve la macro `MODULE_LICENSE`? ¿Es obligatoria?

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:

a. ¿Cuál es la salida del comando anterior?
b. ¿Qué tipos de archivo se generan? Explique para qué sirve cada uno.
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

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

   Ejecutamos:

```
# insmod memory.ko
```

   a. Responda lo siguiente:
   b. ¿Para qué sirven el comando insmod y el comando modprobe? ¿En qué se diferencian?

5. Ahora ejecutamos:

```
$ lsmod | grep memory
```

   Responda lo siguiente:

   a. ¿Cuál es la salida del comando? Explique cuál es la utilidad del comando lsmod.
   b. ¿Qué información encuentra en el archivo /proc/modules?
   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
```

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

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

```
lsmod | grep memory
```

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

```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);
```

   a. Compile y cargue en memoria el módulo.
   b. Invoque al comando dmesg
   c. Descargue el módulo de memoria y vuelva a invocar a dmesg
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?.
   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?.
   c. Clasifique los tipos de dispositivos en Linux. Explique las características de cada uno.

Desarrollando un Driver

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
2. Responda lo siguiente:
   a. ¿Para qué sirve la estructura ssize_t y memory_fops? ¿Y las funciones register_chrdev y unregister_chrdev?
   b. ¿Cómo sabe el kernel que funciones del driver invocar para leer y escribir al dispositivo?
   c. ¿Cómo se accede desde el espacio de usuario a los dispositivos en Linux?
   d. ¿Cómo se asocia el módulo que implementa nuestro driver con el dispositivo?
   e. ¿Qué hacen las funciones copy_to_user y copy_from_user? (https://developer.ibm.com/technologies/linux/articles/l-kernel-memory-access/).
3. Ahora ejecutamos lo siguiente:

```
# mknod /dev/memory c 60 0
```

4. Y luego:

```
# insmod memory.ko
```

   a. Responda lo siguiente:
      i. ¿Para qué sirve el comando mknod? ¿qué especifican cada uno de sus parámetros?.

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

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)
   b. ¿Cuántas invocaciones a memory_write se realizaron?
   c. ¿Cuál es el efecto del comando anterior? ¿Por qué?
   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.