UNIVERSIDAD NACIONAL DE LA PLATA

Sistemas Operativos

Práctica 2 (2024)

Requisitos

Para realizar esta práctica será necesario 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).

En esta práctica la cadena <kernel_code> hace referencia al directorio donde se encuentra el código fuente del kernel, si se usa la misma VM de la práctica 1 este directorio es /home/so/kernel/linux-<version>/.

Importante: En esta práctica hay varios fragmentos de código para copiar. No se recomienda copiar y pegar directamente desde este documento ya que muchas veces el formato introduce caracteres que hacen fallar la compilación con errores del estilo: “error: stray ‘\302’ in program”. Puede copiar los ejemplos más largos desde el siguiente repositorio:

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

Materiales de referencia

https://speakerdeck.com/georgiknox/linux-kernel-hacking-a-crash-course

https://www.kernel.org/doc/html/v4.10/process/adding-syscalls.html

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. ¿Para qué sirven los siguientes archivos?
   a. <kernel_code>/arch/x86/entry/syscalls/syscall_32.tbl
   b. <kernel_code>/arch/x86/entry/syscalls/syscall_64.tbl
4. ¿Para qué sirve la macro asmlinkage?
5. ¿Para qué sirve la herramienta strace? ¿Cómo se usa?

Práctica guiada

La System Call que vamos a implementar accederá a la estructura rq (runqueue), que es la estructura de datos básica del planificador de procesos (scheduler). Esta estructura es definida en

<kernel_code>/kernel/sched/sched.h. Hay una rq por cada procesador y cada proceso del sistema estará sólo en una rq. Esta estructura mantiene información adicional por cada procesador, la cual será el objetivo de la syscall.

Fragmento de la definición de la estructura runqueue rq:

```
struct rq {
    /* runqueue lock: */
    raw_spinlock_t                  __lock;
    /*
     * nr_running and cpu_load should be in the same cacheline because
     * remote CPUs use both these fields when doing load calculation.
     */
    unsigned int                    nr_running;
/* ... */
    /*
     * This is part of a global counter where only the total sum
     * over all CPUs matters. A task can increase this counter on
     * one CPU and if it got migrated afterwards it may decrease
     * it on another CPU. Always updated under the runqueue lock:
     */
     unsigned int                    nr_uninterruptible;
/* ... */
```

Intentaremos acceder con nuestra llamada al sistema a dos datos almacenados en los campos de la estructura runqueue:

- nr_running: contiene el número de tareas en estado de ejecución para este procesador, es decir, tareas en estado TASK_RUNNING.
- nr_uninterruptible: contiene el número de tareas en estado no interrumpible, TASK UNINTERRUPTIBLE.

Es decir, tareas esperando por un evento de entrada/salida no activables por la llegada de una señal. Por tanto al utilizar nuestra llamada al sistema, que denominaremos a partir de este momento rqinfo, obtendremos los datos antes comentados del espacio kernel en el espacio de usuario de nuestra aplicación.

Agregamos una nueva System Call

1. 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>
   - Llamaremos a nuestra system call “rqinfo”. Nuestro entry point será sys_rqinfo
   - 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.

Agregar rqinfo a <kernel_code>arch/x86/entry/syscalls/syscall_64.tbl:

```
458     common      listmount           sys_listmount
459     common      lsm_get_self_attr   sys_lsm_get_self_attr
460     common      lsm_set_self_attr   sys_lsm_set_self_attr
461     common      lsm_list_modules    sys_lsm_list_modules
462     common      rqinfo              sys_rqinfo
```

2. Ahora incluiremos la declaración de nuestra system call (sólo la línea que dice sys_rqinfo, el resto se provee como contexto) en los headers del kernel junto a las otras system calls. Usaremos como referencia la declaración de "sys_ni_syscall" y agregaremos la nuestra justo debajo de ella.

   <kernel_code>/include/linux/syscalls.h:

```
asmlinkage long sys_old_mmap(struct mmap_arg_struct __user *arg);
/*
* Not a real system call, but a placeholder for syscalls which are
* not implemented -- see kernel/sys_ni.c
*/
asmlinkage long sys_ni_syscall(void);
asmlinkage long sys_rqinfo(unsigned long *ubuff, long len); // Agregar
esta declaración
#endif /* CONFIG_ARCH_HAS_SYSCALL_WRAPPER */
```

3. El próximo paso es incluir el código de nuestra syscall en algún punto del árbol de fuentes ya sea añadiendo un nuevo archivo al conjunto del kernel o incluyendo nuestra implementación en algún archivo ya existente. La primera opción nos obligaría a modificar los makefiles del kernel para incluir el nuevo archivo fuente. Por simplicidad añadiremos el código de nuestra syscall en algún punto del código ya existente. Pero ¿dónde colocamos nuestro código?. En nuestro ejemplo, un buen sitio para implementar la syscall inforq en el archivo kernel/sched/core.c, donde podemos acceder a la estructura que nos interesa con facilidad, en este mismo archivo se implementan otra syscalls relacionadas con el planificador de CPU y el scheduler del sistema operativo. Otro motivo por el cual colocar aquí nuestra system call es porque muchas funciones que necesitamos están en este archivo y no son exportadas.

   Agregaremos la siguiente implementación para la system call al final del archivo <kernel_code>/kernel/sched/core.c (después del último #endif si lo hubiera):

```
SYSCALL_DEFINE2 (rqinfo, unsigned long *, ubuff, long, len)
{
    struct rq *rqs;
    struct rq_flags flags;
    unsigned long kbuff[2];
    /*
     * Si el buffer size del usuario
```

```
     * es distinto al nuestro devolvemos error.
     */
    if (len != sizeof (kbuff))
        return -EINVAL;
    /*
     * Delimitamos la región crítica para
     * acceder al recurso compartido.
     */
    rqs = task_rq_lock (current, &flags);
    kbuff[0] = rqs->nr_running;
    kbuff[1] = rqs->nr_uninterruptible;
    task_rq_unlock (rqs, current, &flags);
    if (copy_to_user (ubuff, &kbuff, len))
        return -EFAULT;
    return len;
}
```

Notas:

- El valor “current” utilizado es una macro definida en el Kernel la cual retorna una estructura que representa el proceso actual (el que ejecutó el llamado a la System Call). Esta estructura, llamada task_struct, se encuentra definida en <kernel_code>/include/linux/sched.h.
- Bloqueo del recurso: es importante destacar esto!!, ya que en el código hemos bloqueado la estructura rq antes de acceder a ella, para ello hemos usado funciones que nos proporciona el propio código del planificador. Es común y necesario conseguir acceso exclusivo a este tipo de recursos, pues son compartidos por muchas partes del sistema y tenemos que mantenerlos consistentes. En especial, puesto que las syscall son interrumpibles, tenemos que tener mucho cuidado a la hora de acceder a este tipo de recursos.
- Notar que nuestra system call debe exportar de alguna manera los datos obtenidos al espacio de usuario, es decir al programa o librería que utilizará nuestra system call, es por eso que tenemos la función copy_to_user para llevar a cabo esta tarea.

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. Nuestro último paso es realizar un programa que llame a la System Call. Para ello crearemos un archivo, por ejemplo prueba_inforq.c, con el siguiente contenido:

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

#define NR_rqinfo 462

int main(int argc, char **argv)
{
    long ret;
    unsigned long buf[2];
```

```
    printf("invocando syscall ..\n");
    if ((ret = syscall(NR_rqinfo, buf, 2* sizeof(long))) < 0){
        perror("ERROR");
        return -1;
    }
    printf("runnables: %lu\n", buf[0]);
    printf("uninterruptibles: %lu\n", buf[1]);
    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 prueba prueba_inforq.c
```

Por último nos queda ejecutar nuestro programa y ver el resultado.

```
./prueba
```

Aclaración: Para más opciones de compilación de programas en C se sugiere la lectura del manual de compilador de c.

Monitoreando System Calls

1. Implemente y compile el siguiente programa:

```
#include <stdio.h>
int main() {
    printf("Hola, mundo!\n");
    return 0;
}
```

Ejecute el programa utilizando el comando strace:

```
# strace ./nombre_mi_programa
```

Analice que System Calls son invocadas.

Aclaración: Si el programa strace no está instalado, puede instalarlo en distribuciones basadas en Debian con:

```
# apt-get install strace
```

2. Compile y ejecute los siguientes programas. Realice un trace de los mismos utilizando la herramienta strace. ¿Existe alguna diferencia?

Invocando getpid a través de libc:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    int p_id= (int) getpid();
    printf("El pid es %d\n",p_id);
    return 0;
}
```

Invocando getpid a través de syscall:

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

int main(void) {
    int p_id= syscall(SYS_getpid);
    printf("El pid es %d\n",p_id);
    return 0;
}
```

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.

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. Descarge 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 serc 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.