UNIVERSIDAD NACIONAL DE LA PLATA

Sistemas Operativos

Práctica 3

Requisisitos

Para poder agregar una nueva System Call al sistema es necesario contar con el código fuente del núcleo del sistema operativo. Cuando utilicemos el termino <kernel_code>haremos referencia al directorio donde se encuentra el código fuente del Kernel. Por convención el mismo es colocado en /usr/src/linux. Para realizar la practica debe utilizarse el código utilizado en la practica 1.

Conceptos Generales

1. ¿Qué es una *System Call?*, ¿para que 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 que sirven los siguientes archivos?
   - <kernel_code>/arch/x86/syscalls/syscall_32.tbl
   - <kernel_code>/arch/x86/syscalls/syscall_64.tbl
4. ¿Para qué sirve la macro asmlinkage?
5. ¿Para qué sirve la herramienta strace?, ¿Cómo se usa?

System Calls

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.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.
Definición de la estructura runqueue rq:

```
#include <linux/cpu.h>
/*
 * This is the main, per-CPU runqueue data structure.
 */
struct rq {
 unsigned long nr_uninterruptible;
 task_struct *curr, *idle, *stop;
         unsigned long next_balance;
 .
 .
 struct task_struct *curr;
```

```
unsigned long nr_uninterruptible;
task_struct *curr, *idle, *stop;
long next_balance;
```

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_UNINTERRUMPIBL 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 sólo a la arquitectura x86. Atención:

- Usamos la nomenclatura oficial para denominar la syscall, esto es, el nombre de la syscall precedido con el prefijo sys_
- Debemos asignar un número único a nuestra system call, de modo que aumentaremos en 1 el número de la última existente.

<kernel_code>/arch/x86/syscalls/syscall_32.tbl

```
343     i386    clock_adjtime           sys_clock_adjtime
344     i386    syncfs                  sys_syncfs
345     i386    sendmmsg                sys_sendmmsg
346     i386    setns                   sys_setns
347     i386    process_vm_readv        sys_process_vm_readv
348     i386    process_vm_writev       sys_process_vm_writev
349     i386    kcmp                    sys_kcmp
350     i386    finit_module            sys_finit_module
351     i386    rqinfo                  sys_rqinfo
```

2°)Ahora incluir la declaración de nuestra system call
<kernel_code>/include/linux/syscalls.h

```
asmlinkage long sys_rqinfo(unsigned long *ubuff, long len);
#endif
```

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 obligará 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. Esta segunda opción es la que utilizaremos, 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(forzoso) por el cual colocar aquí nuestra system call es porque muchas funciones que necesitamos están en este archivo y no son reexportadas.
<kernel_code>/kernel/sched/core.c

```
asmlinkage long sys_rqinfo(unsigned long *ubuff, long len){
        struct rq *rqs;
        unsigned long 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 region critica 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 ejecuto 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 nuesta system call debe exportar de alguna manera los datos obtenidos al espacio de usuario, es decir al programa o librería que utlizará 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 syscall.c, con el siguiente contenido:

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

#define NR_rqinfo 351

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;
}
```

Notas:

- 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 prueba_inforq.c
```

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

```
./a.out
```

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 el siguiente programa.

```
#include <stdio.h>

 int main() {
    printf("¡Hola, mundo!\n");
    return 0;
 }
```

Ejecútelo utilizando el comando strace.

```
# strace mi_programa
```

Analice que System Calls son invocadas. Aclaración: Es posible que el programa strace no esté instalado en su equipo. Para ello deberá instalarlo. Si estamos en debian podemos hacerlo ejecutando lo siguiente:

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

O bien bajando el paquete e instalándolo manualmente:

```
#wget http://ftp.us.debian.org/debian/pool/main/s/strace/strace_4.5.20-2_i386.deb
```

```
# dpkg -i strace_4.5.20-2_i386.deb
```

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:

```
#include <stdio.h>
#include <stdlib.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:

```
#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 de lectura obligatoria: http://tldp.org/LDP/lkmpg/2.6/html/c38.html

Conceptos Generales

1. a. ¿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. b. ¿Qué es un driver? ¿para que se utiliza?
3. c. ¿Porque es necesario escribir drivers?
4. d. ¿Cuál es la relación entre modulo y driver en Gnu/Linux?
5. e. ¿Qué implicancias puede tener un bug en un driver o módulo?
6. f. ¿Qué tipos de drivers existen en Gnu/Linux?
7. g) ¿Que hay en el directorio /dev? ¿qué tipos de archivo encontramos en esa ubicación?
8. h) ¿Para qué sirven el archivos /lib/modules/<version>/modules.dep utilizado por el comando modprobe

System Calls

Agregando un módulo a nuestro kernel

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

```
#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:

- Explique brevemente cual es la utilidad del archivo Makefile en la organización de un proyecto de linux.
- ¿Para qué sirve la macro MODULE_LICENSE? ¿Es obligatoria?

3°) Ahora es necesario compilar nuestro modulo 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:

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

4°) El paso que resta el agregar y eventualmente quitar nuestro modulo al kernel en tiempo de ejecución. Ejecutamos:

```
sudo insmod memory.ko
```

Responda lo siguiente:

- ¿Para qué sirve comando insmod., y el comando modprobe?, ¿en qué se diferencian?

5°) Ahora ejecutamos

```
lsmod | grep memory
```

Responda lo siguiente:

- ¿Cuál es la salida del comando? Explique cuál es la utilidad del comando lsmod.
- ¿Qué información encuentra en el archivo /proc/modules?
- Si ejecutamos more /proc/modules encontramos los siguientes fragmentos

```
parport_pc 37412 0 - Live 0xf8b02000
lp 12580 0 - Live 0xf8ae1000
parport 37448 3 ppdev,parport_pc,lp, Live 0xf8ae9000
.memory 3844 0 - Live 0xf89fe000
```

- ¿Qué información obtenemos de aquí?
- ¿Con qué comando eliminamos el módulo de nuestro kernel?

6°) Elimine el modulo recién agregado al kernel. Para corroborar que efectivamente el mismo ha sido eliminado del kernel ejecute lo siguiente

```
lsmod | grep memory
```

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("<1> Hello world!\n");
return 0;
}

static void hello_exit(void) {
printk("<1> Bye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Responda lo siguiente:

- ¿Para qué sirven las funciones module_init y module_exit?. ¿Cómo haría para ver la información del log que arrojan las mismas?.
- 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?.
- Clasifique los tipos de dispositivos en Linux. Explique las características de cada uno.

Desarrollando un Driver

Ahora completaremos nuestro modulo 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. Pasos.
1°) Modifique el archivo memory.c de la siguiente manera

```
#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 */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <asm/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, 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) {

/* Transfering data to user space */
copy_to_user(buf,memory_buffer,1);

/* 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, char *buf,
size_t count, loff_t *f_pos) {
char *tmp;
tmp=buf+count-1;
copy_from_user(memory_buffer,tmp,1);
return 1;
}
```

Responda lo siguiente:

- ¿Para qué sirve la estructura ssize_t y memory_fops? ¿Y las funciones register_chrdev y unregister_chrdev?
- ¿Cómo sabe el kernel que funciones del driver invocar para leer y escribir al dispositivo?
- ¿Cómo se accede desde el espacio de usuario a los dispositivos en Linux?
- ¿Cómo se asocia el modulo que implementa nuestro driver con el dispositivo?

2) Ahora ejecutamos lo siguiente:

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

3) y luego:

```
sudo insmod memory.ko
```

Responda lo siguiente:

- ¿Para qué sirve el comando mknod? ¿qué especifican cada uno de sus parámetros?. ¿Qué son el "major" y el "minor" number? ¿Qué referencian cada uno?

4) Ahora escribimos a nuestro dispositivo

```
echo -n abcdef > /dev/memory
```

5) Ahora leemos desde nuestro dispositivo

```
more /dev/memory
```

Responda lo siguiente:

- ¿Qué salida tiene el anterior comando?, ¿Porque? (ayuda: siga la ejecución de las funciones memory_read y memory_write)
- ¿Cuantas invocaciones a memory_write se realizaron?
- ¿Cuál es el efecto del comando anterior?,¿porque?
- 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.

Crashing

Atención: guarde cualquier información y cierre todos los programas antes hacer el siguiente ejercicio.
1) Compile y cargue el siguiente módulo.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE( "GPL" );

static int initP(void) {
    printk("SO2015!\n");
    for(;;);
    return 0;
}

static void exitP(void) {
    printk("SO2015!\n");
}
```

Responda lo siguiente:

- ¿Cuál es el resultado?, ¿puede hacer algo al respecto?(ej.: matar el proceso, logearse en otra terminal para llevar a cabo alguna acción,etc.), ¿porque ocurre esto?

2) Ahora escribimos el siguiente programa en C llamado user_process.c, con casi el mismo código que el anterior pero en un programa que se ejecutará como un proceso de usuario.

```
int main(){
  for(;;);
  return 0;
}
```

Responda lo siguiente:

- ¿Cuál es el resultado en este caso?, ¿puede hacer algo al respecto?(ej.: matar el proceso, logearse en otra terminal para llevar a cabo alguna acción,etc.), ¿porque ocurre esto?