Practica 2

System Calls

Conceptos generales

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

   Una System Call es un al kernel para ejecutar una función específica que controla un dispositivo o ejecuta una instrucción privilegiada. Estas llamadas permiten que los programas realicen tareas como acceso a archivos, gestión de memoria, creación de procesos y comunicación entre procesos

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

   syscall es una función proporcionada por glibc para hacer system calls de forma explícita. Esta misma es más difícil de usar y menos portátil que utilizar los wrappers específicos de glibc pero aun así es más fácil y portátil que codificar instrucciones en assembler. syscall es más útil cuando se trabaja con system calls que son especiales o más nuevas que la glibc que se está usando. syscall se implementa de forma totalmente genérica; la función no sabe nada sobre lo que hace una system call en particular o si la misma e s valida.

   Parámetros:

   - Número de la system call: indica qué system call específica se desea realizar. Cada llamada al sistema tiene un número único asociado.
   - Los parámetros restantes son parámetros para la system call, en orden, y sus significados dependen de la system call realizada. Cada tipo de system call tiene un número definido de parámetros, de cero a cinco. Si se codifican más de los que toma la system call, los que están de más se ignoran.

3. ¿Para qué sirven los siguientes archivos?
   a. /arch/x86/entry/syscalls/syscall_32.tbl

      Es la syscall table para la arquitectura x86 de 32 bits.

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

      Es la syscall table para la arquitectura x86 de 64 bits.

      Estas tablas mapean los números de las syscalls a sus respectivas funciones en el kernel.

4. ¿Para qué sirve la macro asmlinkage?

   La macro asmlinkage sirve para decirle al compilador que busque los parámetros de una función en el stack de la CPU en lugar de los registros. Se utiliza para indicar al compilador que una función debe ser compilada utilizando la convención de llamada específica del kernel, lo que garantiza que la función se llame correctamente desde el kernel y que los argumentos se pasen de la manera adecuada, generalmente a través del stack en lugar de registro.

   https://www.quora.com/Linux-Kernel-What-does-asmlinkage-mean-in-the-definition-of-system-calls (leer esto)

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

   Sirve para monitorear las system calls realizadas por un proceso. Para utilizarlo se debe ejecutar lo siguiente:
   strace ./programa
   La salida mostrará todas las llamadas al sistema realizadas por el programa, junto con los argumentos de esas llamadas y cualquier valor de retorno.
   Strace tiene muchas opciones adicionales, por ejemplo, strace -f tiene en cuenta además lo que hacen los subprocesos e hilos hijos.

Práctica guiada

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

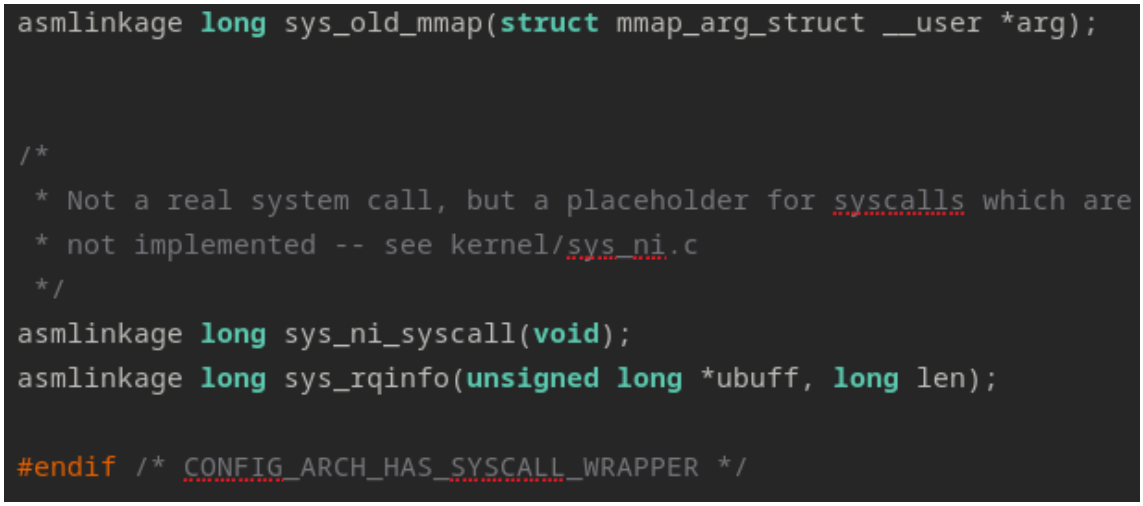
```
458     common      listmount           sys_listmount
459     common      lsm_get_self_attr   sys_lsm_get_self_attr
460     common      lsm_set_self_attr   sys_lsm_set_self_attr
461     common      lsm_list_modules    sys_lsm_list_modules
462     common      rqinfo              sys_rqinfo
```

```
452     common  fchmodat2              sys_fchmodat2
453     64      map_shadow_stack       sys_map_shadow_stack
454     common  futex_wake             sys_futex_wake
455     common  futex_wait             sys_futex_wait
456     common  futex_requeue          sys_futex_requeue
457     common  statmount              sys_statmount
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/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 /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

```
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;
}
```

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:

```c
#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];
```

```c
    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

```
agusnfr@SistemasOperativos:~/pruebas$ gcc -o prueba prueba_inforq.c
agusnfr@SistemasOperativos:~/pruebas$ ./prueba
invocando syscall ..
runnables: 1
uninterruptibles: 4294967231
agusnfr@SistemasOperativos:~/pruebas$
```

Funciono :D

Monitoreando System Calls

1. Implemente y compile el siguiente programa:

```c
#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.

execve("./prueba1", ["./prueba1"], 0x7ffecf2ef1f0 / 44 vars */) = 0*

brk(NULL) = 0x55e2a8cf1000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5fd8cca000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=70970, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 70970, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5fd8cb8000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784

newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...}, AT_EMPTY_PATH) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784

mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5fd8ad7000

mmap(0x7f5fd8afd000, 1396736, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f5fd8afd000

mmap(0x7f5fd8c52000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17b000) = 0x7f5fd8c52000

mmap(0x7f5fd8ca5000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7f5fd8ca5000

mmap(0x7f5fd8cab000, 53072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f5fd8cab000

close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5fd8ad4000

arch_prctl(ARCH_SET_FS, 0x7f5fd8ad4740) = 0

set_tid_address(0x7f5fd8ad4a10) = 3137

set_robust_list(0x7f5fd8ad4a20, 24) = 0

rseq(0x7f5fd8ad5060, 0x20, 0, 0x53053053) = 0

mprotect(0x7f5fd8ca5000, 16384, PROT_READ) = 0

mprotect(0x55e2a715a000, 4096, PROT_READ) = 0

mprotect(0x7f5fd8cfc000, 8192, PROT_READ) = 0

*prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0*

munmap(0x7f5fd8cb8000, 70970) = 0

newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0

getrandom("\x4a\xeb\xe6\xab\x71\x3e\x6f\xc0", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x55e2a8cf1000

brk(0x55e2a8d12000) = 0x55e2a8d12000

write(1, "Hola, mundo!\n", 13Hola, mundo!

) = 13

exit_group(0) = ?

+++ exited with 0 +++

SYSTEM CALL	USO DE SYSCALL
EXECVE	Ejecuta un nuevo programa. Reemplaza la imagen del proceso actual con una nueva imagen del proceso especificada por la ruta del archivo proporcionada
BRK(NULL)	Le dice al proceso donde termina su memoria heap
MMAP	Asigna o desasigna espacio de memoria para el proceso que lo invoca
ACCESS	Comprueba los permisos del usuario para un archivo
OPENAT	Abre un archivo
READ/PREAD64	Leen datos desde un descriptor de archivo abierto previamente
CLOSE	Cierra un descriptor de archivo abierto previamente.
ARCH_PRCTL	Establece el valor del registro FS en la arquitectura x86_64.
SET_TID_ADDRESS/ SET_ROBUST_LIST	Establecen direcciones para las estructuras de datos relacionadas con hilos y manejo de señales
RSEQ	Se relaciona con la manipulación de secuencias de lectura-escritura atómicas
MPROTECT	Cambia los permisos de protección de memoria para ciertas regiones de memoria
PRLIMIT64	Establece u obtiene los límites de recursos para un proceso
MUNMAP	Desasigna regiones previamente asignadas de memoria.
NEWFSTATAT/GETRANDOM	Están relacionadas con la obtención de información sobre un descriptor de archivo y la generación de números aleatorios.

WRITE(1, "HOLA, MUNDO!\N", 13)	Escribe datos en un descriptor de archivo (1 se refiere a la salida estándar)
EXIT_GROUP	Termina el proceso actual con un código de salida proporcionado

https://www.quora.com/What-is-the-execve-system-call-in-the-C-programming-language

https://unix.stackexchange.com/questions/464974/why-execve-and-brknull-are-always-the-first-two-system-calls

https://man7.org/linux/man-pages/man2/

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;
}
```

execve("./prueba2", ["./prueba2"], 0x7fffff08e5a0 / 44 vars */) = 0*

brk(NULL) = 0x55f3368a5000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc1fe7b0000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=70970, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 70970, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fc1fe79e000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...}, AT_EMPTY_PATH) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fc1fe5bd000

mmap(0x7fc1fe5e3000, 1396736, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7fc1fe5e3000

mmap(0x7fc1fe738000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17b000) = 0x7fc1fe738000

mmap(0x7fc1fe78b000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7fc1fe78b000

mmap(0x7fc1fe791000, 53072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fc1fe791000

close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc1fe5ba000

arch_prctl(ARCH_SET_FS, 0x7fc1fe5ba740) = 0

set_tid_address(0x7fc1fe5baa10) = 5025

set_robust_list(0x7fc1fe5baa20, 24) = 0

rseq(0x7fc1fe5bb060, 0x20, 0, 0x53053053) = 0

mprotect(0x7fc1fe78b000, 16384, PROT_READ) = 0

mprotect(0x55f3359d4000, 4096, PROT_READ) = 0

mprotect(0x7fc1fe7e2000, 8192, PROT_READ) = 0

*prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0*

munmap(0x7fc1fe79e000, 70970) = 0

getpid() = 5025

newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0

getrandom("\x81\x90\x48\x44\x0c\xad\x4e\x5a", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x55f3368a5000

brk(0x55f3368c6000) = 0x55f3368c6000

write(1, "El pid es 5025\n", 15El pid es 5025

) = 15

exit_group(0) = ?

+++ exited with 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;
}
```

execve("./prueba3", ["./prueba3"], 0x7ffe4a6f88b0 / 44 vars */) = 0*

brk(NULL) = 0x56349f5bb000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f45c3410000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=70970, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 70970, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f45c33fe000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1922136, ...}, AT_EMPTY_PATH) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

```
mmap(NULL, 1970000, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f45c321d000
mmap(0x7f45c3243000, 1396736, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0x7f45c3243000
mmap(0x7f45c3398000, 339968, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17b000) = 0x7f45c3398000
mmap(0x7f45c33eb000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ce000) = 0x7f45c33eb000
mmap(0x7f45c33f1000, 53072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f45c33f1000
close(3)                        = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f45c321a000
arch_prctl(ARCH_SET_FS, 0x7f45c321a740) = 0
set_tid_address(0x7f45c321aa10)       = 5032
set_robust_list(0x7f45c321aa20, 24)    = 0
rseq(0x7f45c321b060, 0x20, 0, 0x53053053) = 0
mprotect(0x7f45c33eb000, 16384, PROT_READ) = 0
mprotect(0x56349e492000, 4096, PROT_READ) = 0
mprotect(0x7f45c3442000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f45c33fe000, 70970)         = 0
getpid()                        = 5032
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...},
AT_EMPTY_PATH) = 0
getrandom("\xc8\xab\x06\x48\x6c\x63\x45\xfd", 8, GRND_NONBLOCK) = 8
brk(NULL)                       = 0x56349f5bb000
brk(0x56349f5dc000)                 = 0x56349f5dc000
write(1, "El pid es 5032\n", 15El pid es 5032
)      = 15
exit_group(0)                    = ?
+++ exited with 0 +++
```

No, no existe ninguna diferencia. Esto sucede dado que getpid() es un wrapper ofrecido por glibc que internamente realiza una invocación a la macro syscall pasandole SYS_getpid tal como se hace en el segundo programa.

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?

   La porción de codigo que se agrega al kernel en tiempo de ejecución se denomina “modulo”. No, no es necesario reiniciar el cargar el sistema al cargarlo. Si no se hiciera esto el kernel debería ser 100% monolito, es decir, los módulos en si serian parte del kernel, por lo que, si se desea agregar una funcionalidad, se debería re-compilar el kernel, instalarlo y rebotear la PC.

2. ¿Qué es un driver? ¿Para qué se utiliza?
   Un drive es un software que actúa como intermediario entre el SO y el hardware. Su función principal es permitir que el sistema operativo se comunique con los componentes físicos

3. ¿Por qué es necesario escribir drivers?

   Es necesario que se escriban drivers ya que son los que permiten que se pueda comunicar el SO con el hardware, ofreciéndole al SO abstracción respecto a los distintos dispositivos físicos, simplificando su desarrollo y permitiendo una mayor compatibilidad entre diferentes dispositivos y sistemas operativos.

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

Los drivers son implementados como módulos del SO.

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

   Todos los módulos están en el mismo espacio de memoria en modo kernel por lo que un bug en alguno de ellos podría implicar modificaciones indebidas o incluso la caída de todo el sistema, haciendo que haya un BSD (blue screen of death), en Windows, o Kernel Panic, en Unix.

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

   - Drivers de bloques: son un grupo de bloques de datos persistentes. Leemos y escribimos de a bloques, generalmente de 1024 bytes.
   - Drivers de carácter: Se accede de a 1 byte a la vez y 1 byte sólo puede ser leído por única vez.
   - Drivers de red: tarjetas ethernet, WIFI, etc.

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

   En /dev están las representaciones de los dispositivos.

   En /dev hay archivos de:
   - Dispositivos de caracteres: interfaces simples para dispositivos de caracteres. Los dispositivos de caracteres no requieren almacenamiento en búfer cuando se comunican con un controlador
   - Dispositivos de bloques: interfaces simples para bloquear dispositivos. Un controlador accede a los datos de los dispositivos de bloque a través de un caché
   - Pseudo-Dispositivos: no se corresponden con dispositivos de hardware, si no que proporcionan varias funciones útiles, como por ejemplo /dev/random o /dev/null. Los pseudodispositivos también son dispositivos de caracteres, se puede distinguir entre pseudodispositivos y otros dispositivos de caracteres utilizando el número major de un dispositivo.

- Enlaces simbólicos: algunos archivos de dispositivos en el directorio /dev/ no aparecen como dispositivos de bloques o caracteres. En su lugar, se usan enlaces simbólicos.

  https://www.baeldung.com/linux/dev-directory

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

   Es un archivo de dependencias de módulos. Contiene información sobre las dependencias entre diferentes módulos del kernel cargados en el SO.
   Cuando se carga un módulo del kernel, el SO necesita saber si ese módulo depende de otros módulos para funcionar correctamente. El archivo modules.dep especifica estas dependencias, lo que permite al SO cargar automáticamente los módulos necesarios cuando se carga un módulo específico.

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

   initramfs e genera o actualiza:
   - Durante la instalación del sistema operativo.
   - Después de instalar un nuevo kernel.
   - Cuando se realizan cambios significativos en la configuración del sistema.

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

    Para cumplir su objetivo básico, un initramfs debe incluir los módulos y drivers necesarios para:

    - Montar el sistema de archivos raíz (root filesystem).
    - Detectar y montar dispositivos de almacenamiento como discos duros, unidades flash USB o sistemas de archivos en red.

- Detectar y cargar los controladores necesarios para el hardware crítico del sistema, como controladores de disco, controladores de red y controladores de gráficos, dependiendo de la configuración específica del sistema.

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 <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
```

```
#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.

Es un archivo que se incluye en la carpeta raíz de un proyecto para que le digan a un programa, make, que se ejecuta desde la terminal, qué hacer con cada uno de los archivos de código para compilarlos

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

Es una macro que permite que los módulos cargables del kernel declaren su licencia. No, la declaración MODULE_LICENSE() no es obligatoria; un módulo del kernel se puede cargar sin ella, pero el kernel será marcado como

“contaminado”. La licencia declarada del módulo se utiliza para decidir si un módulo determinado puede acceder al pequeño número de símbolos "solo GPL" en el kernel.

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?

```
agusnfr@SistemasOperativos:~/pruebas$ make -C $HOME/kernel/linux-6.7 M=$(pwd) mo
dules
make: Entering directory '/home/agusnfr/kernel/linux-6.7'
  CC [M]  /home/agusnfr/pruebas/memory.o
  MODPOST /home/agusnfr/pruebas/Module.symvers
  CC [M]  /home/agusnfr/pruebas/memory.mod.o
  LD [M]  /home/agusnfr/pruebas/memory.ko
make: Leaving directory '/home/agusnfr/kernel/linux-6.7'
```

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

memory.c: es el archivo fuente en C del módulo del kernel. Contiene el código fuente del módulo que se compiló para producir los archivos objeto y el archivo del módulo.

memory.ko: es el archivo del módulo del kernel compilado. Como mencionamos antes, es el código objeto del módulo de kernel que puede cargarse en el núcleo en tiempo de ejecución.

memory.mod: es el archivo que contiene información sobre el módulo, como su nombre y versión. Es generado como parte del proceso de compilación.

memory.mod.c: es generado por el sistema de compilación del kernel y contiene código fuente en C que describe el módulo y su versión. Se utiliza en el proceso de compilación del módulo.

memory.mod.o: es un archivo objeto generado durante el proceso de compilación del módulo. Sin embargo, está relacionado específicamente con la información del módulo.

memory.o: es otro archivo objeto generado durante el proceso de compilación del módulo. Contiene el código objeto del módulo que puede ser cargado en el núcleo en tiempo de ejecución.

Module.symvers: contiene información sobre los símbolos exportados por los módulos cargables del kernel. Ayuda a garantizar la compatibilidad entre módulos de kernel y el kernel principal.

modules.order: especifica el orden en el que se deben cargar los módulos del kernel. Es utilizado por el sistema de construcción del kernel para determinar el orden de carga de los módulos.

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

   Ejecutamos:

   ```
   # insmod memory.ko
   ```

   Responda lo siguiente:

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

      Insmod trata de cargar el módulo especificado y modprobe carga el módulo especificado y sus dependencias

5. Ahora ejecutamos:

   ```
   $ lsmod | grep memory
   ```

   Responda lo siguiente:

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

```
agusnfr@SistemasOperativos:~/pruebas$ lsmod | grep memory
memory                  8192  0
```

muestra los módulos cargables en el kernel que están cargados actualmente

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

Este archivo muestra una lista de todos los módulos cargados en el kernel.

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
```

- La primera columna contiene el nombre del módulo.
- La segunda columna se refiere al tamaño de la memoria del módulo, en bytes.
- La tercera columna enumera cuántas instancias del módulo están cargadas actualmente. Un valor de cero representa un módulo descargado.
- La cuarta columna indica si el módulo depende de la presencia de otro módulo para funcionar y enumera esos otros módulos.
- La quinta columna enumera en qué estado de carga se encuentra el módulo: Live, Loading o Unloading son los únicos valores posibles.
- La sexta columna enumera el desplazamiento de memoria del kernel actual para el módulo cargado. Esta información puede resultar útil para fines de depuración o para herramientas de creación de perfiles como oprofile.

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

Con el comando rmmod

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

```
lsmod | grep memory
```

```
root@SistemasOperativos:~# cd /home/agusnfr/pruebas
root@SistemasOperativos:/home/agusnfr/pruebas# rmmod memory.ko
root@SistemasOperativos:/home/agusnfr/pruebas# exit
logout
agusnfr@SistemasOperativos:~/pruebas$ lsmod | grep memory
agusnfr@SistemasOperativos:~/pruebas$
```

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);
```

```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.

```
agusnfr@SistemasOperativos:~/pruebas$ make -C $HOME/kernel/linux-6.7/ M=$(pwd) modules
make: Entering directory '/home/agusnfr/kernel/linux-6.7'
  CC [M]  /home/agusnfr/pruebas/memory.o
  MODPOST /home/agusnfr/pruebas/Module.symvers
  CC [M]  /home/agusnfr/pruebas/memory.mod.o
  LD [M]  /home/agusnfr/pruebas/memory.ko
make: Leaving directory '/home/agusnfr/kernel/linux-6.7'
agusnfr@SistemasOperativos:~/pruebas$ su -
Password:
root@SistemasOperativos:~# cd /home/agusnfr/pruebas/
root@SistemasOperativos:/home/agusnfr/pruebas# insmod memory.ko
```

b. Invoque al comando dmesg

```
[ 3269.551698] memory: loading out-of-tree module taints kernel.
[ 3269.551705] memory: module verification failed: signature and/or required key missing -
 tainting kernel
[ 4164.688691] Hello world!
```

c. Descargue el módulo de memoria y vuelva a invocar a dmesg

```
[ 3269.551705] memory: module verification failed: signature and/or required key missing -
tainting kernel
[ 4164.688691] Hello world!
[ 4299.684014] 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?.

Sirven para personalizar los nombres de las funciones de inicialización y cleanup. Estas funciones realizan la vinculación entre los nombres personalizados e init_module() y cleanup_module().
Para ver la información del log que arrojan estas funciones se pueden usar herramientas de registro del kernel, como dmesg.

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?.

Para tener un driver completo nos faltaría la interacción con un dispositivo físico o virtual.

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

- Dispositivos de acceso aleatorio:
    - Almacenan y recuperan datos en bloques de tamaño fijo.
    - Permiten un acceso no secuencial a los datos, lo que significa que se puede leer o escribir en cualquier ubicación de manera aleatoria.
    - Son dispositivos de almacenamiento de datos que mantienen la integridad de los datos independientemente del orden en que se accede a ellos.
    - Ejemplos: discos duros (HDD), unidades de estado sólido (SSD), unidades USB y tarjetas de memoria.
    - Acceso:
        - El acceso se realiza mediante operaciones de lectura y escritura en bloques de datos.
        - Se pueden formatear con sistemas de archivos como ext4, NTFS, FAT32, etc.
        - Los sistemas de archivos proporcionan una abstracción para organizar y administrar los datos almacenados en estos dispositivos.

- Dispositivos seriales (por ejemplo, mouse, sonido, etc.):

- Transmiten datos secuenciales, uno tras otro, en forma de caracteres.
- Son dispositivos de entrada/salida que manejan datos de manera secuencial, caracter por caracter.
- Los datos se transmiten y reciben en serie, uno detrás del otro, en lugar de en bloques.
- Ejemplos: mouse, teclado, dispositivos de sonido (altavoces, micrófonos), GPS, etc.
- Acceso:
  - Los datos se leen o escriben secuencialmente, carácter por carácter, sin estructura de bloques.
  - Los controladores de dispositivos proporcionan una interfaz para que el kernel del sistema operativo interactúe con estos dispositivos.
  - Los eventos generados por estos dispositivos (como movimientos del mouse, pulsaciones de teclas, datos de audio) se procesan y utilizan para diversas funciones y aplicaciones en el sistema.

Desarrollando un Driver

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
   Recordar recompilar el módulo antes de cargarlo

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: Se utiliza para contar bytes o indicar errores. Es un tipo de dato entero firmado capaz de almacenar valores al menos en el rango [-1, SSIZE_MAX].

      memory_fops: es el struct donde se encuentran definidas las funciones que se deben llamar cuando ocurren ciertas operaciones que el dispositivo es capaz de hacer.

register_chrdev: registra el driver en el kernel. Se le debe pasar el major number, nombre del driver y la file_operation

unregister_chrdev: desregistra el driver en el kernel. Se le debe pasar el major number y el nombre del driver.

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

Lo sabe gracias a la struct file_operation mencionada en el punto a.

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

- A través de archivos en /dev
- System Calls
- Interfaz de archivo /proc y /sys
- ioctl()

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

La asociación entre un módulo de kernel (que implementa un driver) y un dispositivo se realiza principalmente mediante el registro del driver con el subsistema correspondiente del kernel.

- Identificación del dispositivo: Se debe identificar el dispositivo para el cual se desea asociar el driver.
- Registro del driver: El driver se registra con el subsistema relevante del kernel utilizando funciones proporcionadas por el kernel, como register_chrdev() para dispositivos de caracteres o register_blkdev() para dispositivos de bloque.
- Indicación de las funciones del driver: Se definen e indican al kernel las funciones del driver que manejarán las operaciones específicas del dispositivo mediante una estructura file_operations y la función de registro del driver.
- Manejo de detección de dispositivos: Si el dispositivo es detectado dinámicamente (por ejemplo, al conectar un dispositivo USB), el kernel lo

detecta y busca un driver adecuado para manejarlo. Si el driver registrado coincide con el dispositivo detectado, el kernel lo asociará automáticamente.

- Carga del módulo del driver: Finalmente, se asegura que el módulo del driver esté cargado en el kernel, ya sea manualmente utilizando herramientas como insmod, o automáticamente durante el arranque del sistema mediante la configuración en archivos como /etc/modules.

e. ¿Qué hacen las funciones copy_to_user y copy_from_user?

copy_to_user: copia un bloque de datos en el espacio del usuario.
copy_from_user: copia un bloque de datos desde el espacio del usuario

3. Ahora ejecutamos lo siguiente:

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

```
root@SistemasOperativos:/home/agusnfr/pruebas# mknod /dev/memory c 60 0
```

4. Y luego:

```
# insmod memory.ko
```

```
root@SistemasOperativos:/home/agusnfr/pruebas# insmod memory.ko
```

a. Responda lo siguiente:

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

El comando mknod crea una entrada de directorio y el correspondiente i-nodo para un archivo especial. El primer parámetro es el nombre del dispositivo de entrada. Mknod tiene dos formas con diferentes flags.

En la primer forma el segundo parámetro puede ser b o c según se trate de dispositivos de carácter o de bloque. Los últimos dos parámetros son números que especifican el major y minor device number. El major device number ayuda al sistema a encontrar el código del driver del dispositivo . El minor device number es el número de unidad o línea que puede ser decimal u octal.

mknod Name { b | c } Major Minor

En la segunda forma del comando mknod, se utiliza la flag p para crear pipelines FIFO

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

Major: Identifica el tipo de dispositivo, como disco duro, impresora o tarjeta de red.
Minor: Identifica un dispositivo específico dentro de un tipo, como una partición de disco duro o un puerto específico en una tarjeta de red.

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)
   La salida del comando anterior es f. El dispositivo solo almacena el ultimo carácter escrito.

   b. ¿Cuántas invocaciones a memory_write se realizaron?

   Se hicieron 6

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

```
write(1, "abcdef", 6)                   = 1
write(1, "bcdef", 5)                    = 1
write(1, "cdef", 4)                     = 1
write(1, "def", 3)                      = 1
write(1, "ef", 2)                       = 1
write(1, "f", 1)                        = 1
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
```

Es así porque al ser un dispositivo de carácter, se va escribiendo de a caracteres, no puede escribir el string completo de una.

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.

Algunos aspectos importantes que se deberían considerar son:

- Sincronización de acceso: Al leer un dispositivo, especialmente en un entorno multi-hilo o multi-proceso, es importante garantizar la sincronización adecuada del acceso al dispositivo para evitar interferencias. Esto puede lograrse mediante el uso de semáforos, mutexes u otros mecanismos de sincronización proporcionados por el kernel de Linux.

- Operaciones de control: Es posible que el driver necesite admitir operaciones de control adicionales más allá de la simple lectura y escritura de datos. Las llamadas ioctl (Input/Output Control) pueden ser utilizadas para implementar estas operaciones de control. Las ioctl permiten que los programas de usuario envíen comandos al driver para realizar diversas funciones, como configurar parámetros del dispositivo, obtener información adicional o realizar acciones específicas del dispositivo.

- Acceso a registros de hardware: En el caso de dispositivos de hardware específicos puede ser necesario acceder directamente a los registros de hardware para configurar el dispositivo o leer datos. Esto puede implicar el uso de funciones como inb() y outb() para leer y escribir bytes de los puertos de E/S del hardware.