Práctica 2

lunes, 12 de abril de 2021 10:38

System Calls

1. ¿Qué es una *System Call?*, ¿para que se utiliza?

Son llamados al kernel para ejecutar una función específica que controla un dispositivo o ejecuta una instrucción privilegiada. Su funcionalidad se ejecuta en modo Kernel pero en contexto del proceso.

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

La macro syscall realiza una system call genérica.
Recibe varios parámetros, y el primero se llama sysno.
Sysno es el número identificador de la system call a ejecutar. Los demás argumentos son los argumentos que se le van a pasar a la system call invocada. A la hora de pasar los argumentos a la syscall, se va a respetar el orden.
Cualquier system call tiene una cantidad de argumentos desde 0 a 5. Si se le pasan más argumentos de los necesarios, los que sobren van a ser ignorados.

3. ¿Para que sirven los siguientes archivos?

- <kernel_code>/arch/x86/syscalls/syscall_32.tbl
- <kernel_code>/arch/x86/syscalls/syscall_64.tbl

Se les llama syscall table. Contienen un listado de las syscalls, en donde las identifican con un número único, además de guardar también el "entry point" y "abi".

4. ¿Para qué sirve la macro asmlinkage?

asmlinkage instruye al compilador a pasar parámetros por stack y no, por ejemplo, en registros.

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

La herramienta strace nos dice las system calls que realiza cualquier programa. Para utilizarla hay que escribir "strace " y el comando que se quiere analizar. También se puede usar "-p " seguido del pid de un proceso para analizarlo.

Módulos y Drivers

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?

Se los denomina módulos. No es necesario reiniciar el sistema al cargarlo.
Si no tuvieramos los módulos, deberíamos utilizar un kernel monolítico, tener un kernel más grande e incluir esos módulos de base.
A monolithic kernel is one where there is no access protection between the various kernel subsystems and where public functions can be directly called between various subsystems.

2. ¿Qué es un driver? ¿para que se utiliza?

Un driver es una porción de código, el cuál se implementa como módulo, que le permite al SO interactuar con un dispositivo específico.

3. ¿Porque es necesario escribir drivers?

Porque cada dispositivo del mercado funciona de forma diferente, por lo que el SO no puede tener registrado como comuinicarse con cada uno de ellos. Para solucionar este inconveniente aparecen los drivers, los cuales se encargan de "traducir" todo el conjunto de funciones que tiene el dispositivo a algunas reducidas, una especie de interfaz o API, que define el SO. Para agregar nuevo HW sólo basta indicar el driver correspondiente sin necesidad de cambios en el Kernel. El SO se comunica con los drivers escribiendo o leyendo de registros que se encuentran dentro de los dispositivos.

4. ¿Cuál es la relación entre modulo y driver en Gnu/Linux?

Cada driver es implementado como un módulo.

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

Como el Kernel de Linux es monolítico híbrido, lo que hace es cargar los módulos por demanda, pero si un módulo o driver cargado tiene un bug, se deberá reiniciar el sistema.

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 caracter: se accede de a 1 byte a la vez y 1 byte solo puede ser leído por única vez.
Drivers de red: tarjetas ethernet, WIFI, etc.

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

Se encuentran los archivos que representan los dispositivos (dev = devices).
Se pueden encontrar archivos tipo "hda1" representando al disco.

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

Es un archivo que contiene las dependencias de cada módulo.

- ¿Para qué sirve comando insmod y el comando modprobe? ¿En qué se diferencian?

- ¿Con qué comando eliminamos el módulo de nuestro kernel?

Desarrollando un Driver

- ¿Para qué sirve la estructura ssize_t y memory_fops? ¿Y las funciones register_chrdev y unregister_chrdev?

ssizze_t: para definir las funciones que utilizan los drivers
memory_fops: determina las funciones que debe utilizar el kernel para utilizar la memoria

La función register_chrdev sirve para registrar un driver
La función unregister_chrdev sirve para desregistrar un driver

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

Mediante la struct file operations, creo que se encuentra en "kernel code/linux/Documentation/devices.txt".

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

Mediante la utilización (lectura o escritura) de los archivos dentro de /dev

- ¿Cómo se asocia el modulo que implementa nuestro driver con el dispositivo?

Cuando se registra un driver, en la función module_init se hace:
register_chrdev(major_number, "myDriver", &my_driver_fops);
Esta línea asocia un driver "myDriver" a un majer_number que le corresponde al dispositivo.

- ¿Qué hacen las funciones `copy_to_user` y `copy_from_user`? (https://developer.ibm.com/technologies/linux/articles/l-kernel-memory-access/).

copy_to_user: copies a block of data from the kernel to user space
copy_from_user: copies a block of data from user space to the kernel

Monitoreando Systems Call
1)

```
root@so2020:/home/so# strace ./mi_programa
execve("./mi_programa", ["./mi_programa"], 0x7fff4188b760 /* 13 vars */) = 0
brk(NULL)                               = 0x560a44039000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=21695, ...}) = 0
mmap(NULL, 21695, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fab4f92c000
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\260A\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fab4f92a000
mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fab4f769000
mprotect(0x7fab4f78b000, 1658880, PROT_NONE) = 0
mmap(0x7fab4f78b000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000)
= 0x7fab4f78b000
mmap(0x7fab4f8d3000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16a000) = 0x7fab4f
8d3000
mmap(0x7fab4f920000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b6000)
= 0x7fab4f920000
mmap(0x7fab4f926000, 14336, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f
ab4f926000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7fab4f92b500) = 0
mprotect(0x7fab4f920000, 16384, PROT_READ) = 0
mprotect(0x560a4370a000, 4096, PROT_READ) = 0
mprotect(0x7fab4f959000, 4096, PROT_READ) = 0
munmap(0x7fab4f92c000, 21695)           = 0
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x4, 0x1), ...}) = 0
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
brk(NULL)                               = 0x560a44039000
brk(0x560a4405a000)                     = 0x560a4405a000
write(1, "Hola, mundo!\n", 13Hola, mundo!
)           = 13
exit_group(0)                           = ?
+++ exited with 0 +++
root@so2020:/home/so# _
```

2)
Invocación con libc

```
execve("./invocacionlibc", ["./invocacionlibc"], 0x7ffe0be501b0 /* 13 vars */) = 0
brk(NULL)                               = 0x557884122000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=21695, ...}) = 0
mmap(NULL, 21695, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f665bfda000
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\260A\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f665bfd8000
mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f665be17000
mprotect(0x7f665be39000, 1658880, PROT_NONE) = 0
mmap(0x7f665be39000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000)
= 0x7f665be39000
mmap(0x7f665bf81000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16a000) = 0x7f665b
f81000
mmap(0x7f665bfce000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b6000)
= 0x7f665bfce000
mmap(0x7f665bfd4000, 14336, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f
665bfd4000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f665bfd9500) = 0
mprotect(0x7f665bfce000, 16384, PROT_READ) = 0
mprotect(0x557883fde000, 4096, PROT_READ) = 0
mprotect(0x7f665c007000, 4096, PROT_READ) = 0
munmap(0x7f665bfda000, 21695)           = 0
getpid()                                = 9822
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x4, 0x1), ...}) = 0
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
brk(NULL)                               = 0x557884122000
brk(0x557884143000)                     = 0x557884143000
write(1, "El pid es 9822\n", 15El pid es 9822
)        = 15
exit_group(0)                           = ?
+++ exited with 0 +++
root@so2020:/home/so# _
```

Invocación con system call:

```
execve("./invocacionsyscall", ["./invocacionsyscall"], 0x7fff0db7eab0 /* 13 vars */) = 0
brk(NULL)                               = 0x5614fb49e000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No existe el fichero o el directorio)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=21695, ...}) = 0
mmap(NULL, 21695, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa83e3cf000
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\260A\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fa83e3cd000
mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fa83e20c000
mprotect(0x7fa83e22e000, 1658880, PROT_NONE) = 0
mmap(0x7fa83e22e000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000)
= 0x7fa83e22e000
mmap(0x7fa83e376000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16a000) = 0x7fa83e
376000
mmap(0x7fa83e3c3000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b6000)
= 0x7fa83e3c3000
mmap(0x7fa83e3c9000, 14336, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f
a83e3c9000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7fa83e3ce500) = 0
mprotect(0x7fa83e3c3000, 16384, PROT_READ) = 0
mprotect(0x5614fa5c6000, 4096, PROT_READ) = 0
mprotect(0x7fa83e3fc000, 4096, PROT_READ) = 0
munmap(0x7fa83e3cf000, 21695)           = 0
getpid()                                = 9826
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x4, 0x1), ...}) = 0
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
brk(NULL)                               = 0x5614fb49e000
brk(0x5614fb4bf000)                     = 0x5614fb4bf000
write(1, "El pid es 9826\n", 15El pid es 9826
)        = 15
exit_group(0)                           = ?
+++ exited with 0 +++
root@so2020:/home/so# _
```

No existe diferencia en el uso de las system calls.

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

La utilidad del archivo Makefile es un archivo con un listado de tareas, este archivo es utilizado por el comando make.

La macro MODULE_LICENSE sirve para definir sobre que licencia está definido el código. No 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.

La salida es:

```
make: se entra en el directorio '/home/so/kernel/linux-5.6'
  CC [M]  /home/so/memory.o
  MODPOST 1 modules
  CC [M]  /home/so/memory.mod.o
  LD [M]  /home/so/memory.ko
make: se sale del directorio '/home/so/kernel/linux-5.6'
```

El archivo memory.ko es un archivo u objeto que representa un módulo utilizado por el kernel de linux
El archivo memory.o es un objeto que representa el módulo
El archivo memory.mod.o is a type of program unit that contains specifications of such entities as data objects, parameters, structures, procedures, and operators. These precompiled specifications and definitions can be used by one or more program units.

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

```
# insmod memory.ko
```

Responda lo siguiente:

- ¿Para qué sirve comando insmod y el comando modprobe? ¿En qué se diferencian?

insmod: trata de cargar el modulo especificado
modprobe: emplea la informacion generada por depmod e informacion de /etc/modules.conf para cargar el modulo especificado.

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?

lsmod sirve para listar los módulos cargados

```
root@so2020:/home/so# lsmod | grep memory
memory                 12288  0
root@so2020:/home/so#
```

/proc/modules contiene los modulos cargados (cat /proc/modules es lo mismo que hacer lsmod)

- 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?

```
nfs      170109  0 -          Live 0x129b0000
lockd    51593   1 nfs,       Live 0x128b0000
nls_utf8 1729    0 -          Live 0x12830000
vfat     12097   0 -          Live 0x12823000
fat      38881   1 vfat,      Live 0x1287b000
autofs4  20293   2 -          Live 0x1284f000
sunrpc   140453  3 nfs,lockd, Live 0x12954000
3c59x    33257   0 -          Live 0x12871000
uhci_hcd 28377   0 -          Live 0x12869000
md5      3777    1 -          Live 0x1282c000
ipv6     211845 16 -          Live 0x128de000
ext3     92585   2 -          Live 0x12886000
jbd      65625   1 ext3,      Live 0x12857000
dm_mod   46677   3 -          Live 0x12833000
```

The first column contains the name of the module.

The second column refers to the memory size of the module, in bytes.

The third column lists how many instances of the module are currently loaded. A value of zero represents an unloaded module.

The fourth column states if the module depends upon another module to be present in order to function, and lists those other modules.

The fifth column lists what load state the module is in: **`Live`**, **`Loading`**, or **`Unloading`** are the only possible values.

The sixth column lists the current kernel memory offset for the loaded module. This information can be useful for debugging purposes, or for profiling tools such as **`oprofile`**.

Basicamente, sabemos que nuestro modulo pesa 3844 bytes, que no tiene instancias cargadas, que no depende de otros módulos, que está en estado "Live" y la posición de memoria del kernel donde se aloja.

Para eliminar el módulo del kernel se utiliza rmmod

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?.

module_init determina la función que se va a ejecutar cuando se ejecute insmod sobre el módulo seleccionado, y module_exit determina cual se va a ejecutar cuando se haga rmmod. Para conocer la información del log podemos poner, dentro de la función a la que apuntan, un printk de KERN_INFO

- 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?.

Asignar el módulo a un dispositivo

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

Se dividen en 2 tipos (según su forma de acceso):
Dispositivos de acceso aleatorio(ej. discos).
Dispositivos seriales(ej. Mouse, sonido,etc).
También hay dispositivos fifo (ej. pipe?xd)

Una clasificación más completa sería:

Algunas formas de clasificar los dispositivos de E/S son:

- Unidad de transferencia
  - Por bloques
    - Discos
  - Por caracteres
    - Teclado, mouse
- Formas de acceso
  - Secuencial
    - Cintas magnéticas
  - Aleatorio
    - Cualquier otra cosa
- Tipo de acceso
  - Compartido
    - Discos
  - Exclusivo
    - Impresora
- Otro tipo de acceso
  - Solo lectura
    - CDROM
  - Solo escritura
    - Monitores
  - Las dos
    - Los discos
- Legible para el usuario
  - Comunicación con el usuario: impresoras, pantallas, etc
- Legibles para la maquina
  - Comunicación con componentes: discos, cintas, sensores
- Comunicación
  - Interface de red, modem

Desarrollando un Driver

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?

mknod is creating a device file, usually to be located in the /dev branch
The first parameter is telling which kind of device to create, here c for character device. Other choices might be b for block devices, p for fifo (pipe). The second parameter is the major number, it identifies the driver for the kernel to use. The third parameter is the minor number, it is passed to the driver for its internal usage.

The major number identifies the driver associated with the device
The minor number is used only by the driver specified by the major number. It is common for a driver to control several devices (as shown in the listing); the minor number provides a way for the driver to differentiate among them.

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)

La salida es f, ya que nosotros le mandamos abcdef como mensaje, y como guarda un solo carácter quedó con f.

- ¿Cuantas invocaciones a memory_write se realizaron?

6, una por cada carácter

- ¿Cuál es el efecto del comando anterior?,¿porque?

Escribe un carácter

- 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.

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?

Se colgó el módulo. No se puede hacer nada al respecto, ya que como el módulo se carga en el código del kernel no se puede dar de baja, se muere todo.

```
root@so2020:/home/so# make -C /home/so/kernel/linux-5.6 M=`pwd` modules
make: se entra en el directorio '/home/so/kernel/linux-5.6'
  CC [M]  /home/so/modulo_crashing.o
  MODPOST 1 modules
  CC [M]  /home/so/modulo_crashing.mod.o
  LD [M]  /home/so/modulo_crashing.ko
make: se sale del directorio '/home/so/kernel/linux-5.6'
root@so2020:/home/so# ls
kernel            memory.mod     modules.order        modulo_crashing.mod.c  prueba_inforq.c
linux-5.6.tar.xz  memory.mod.c   Module.symvers       modulo_crashing.mod.o
Makefile          memory.mod.o   modulo_crashing.c    modulo_crashing.o
memory.c          memory.o       modulo_crashing.ko   patch-5.6.2.xz
memory.ko         mi_programa.c  modulo_crashing.mod  prueba
root@so2020:/home/so# sudo insmod modulo_crashing.ko
```

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.

```c
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?

Preguntar:
1) No tenemos ni idea que es un memory_fops (desarrollando un driver)
2) La segunda pregunta de como sabe el so ni idea tampcoo (desarrollando un driver)
-) Ultima antes de crushing

Se colgó el programa. En este caso si lo podemos dar de baja, ya que corre a nivel de usuario y no con el código del kernel.

```
root@so2020:/home/so# gcc -o user_process user_process.c
root@so2020:/home/so# ./user.process
-bash: ./user.process: No existe el fichero o el directorio
root@so2020:/home/so# ./user_process
^C
root@so2020:/home/so#
```