

Sistemas Operativos - Práctica 2 [2025]

System Calls

Conceptos generales

1. ¿Qué es una System Call? ¿Para qué se utiliza? Una llamada al sistema es una rutina que permite a una aplicación de usuario solicitar acciones que requieren privilegios especiales. Las funciones del sistema operativo se ponen a disposición del programa de aplicación en forma de bibliotecas de programación. Un conjunto de funciones de biblioteca que se encuentran en una biblioteca como, por ejemplo, `libc.a` puede tener funciones que realizan algún proceso en modalidad de usuario y, a continuación, inician internamente una llamada al sistema. Las funciones de sistema operativo disponibles para los programas de aplicación se pueden dividir o mover entre funciones de modalidad de usuario y funciones de modalidad de kernel según sea necesario para diferentes releases o plataformas de máquina. ^1
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. Es más difícil de usar y menos portátil que usar las librerías (`libc`) pero aún 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 librería que se está usando.

Parámetros:

- **Número de la system call:** indica qué system call específica se desea realizar. Es único.
- Los parámetros restantes son parámetros de para la system call, en orden, y sus significados dependen de la system call realizada. Se permiten hasta 5. Los que están de más se ignoran.

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

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

```
root@so:/boot# ls -lh /boot | grep vmlinuz
-rw-r--r-- 1 root root 7,9M ene  2 10:31 vmlinuz-6.1.0-29-amd64
-rw-r--r-- 1 root root 7,9M feb  7 06:43 vmlinuz-6.1.0-31-amd64
-rw-r--r-- 1 root root 8,3M mar 24 15:37 vmlinuz-6.13.7
```

Lo que se muestra son las imágenes del núcleo del kernel de Linux que están instaladas en el sistema. Su nombre indica la versión del kernel. Estos archivos son usados por el bootloader (GRUB) al arrancar el sistema donde carga uno de estos núcleos en memoria y lo ejecuta.

4. Acceda al código fuente de GNU Linux, sea visitando <https://kernel.org/> o bien trayendo el código del kernel(cuidado, como todo software monolítico son unos cuantos gigas)

```
git clone https://github.com/torvalds/linux.git
```

5. ¿Para qué sirve el siguiente archivo `arch/x86/entry/syscalls/syscall_64.tbl`? Se encarga de definir la tabla de llamadas al sistema para la arquitectura `x86_64`. Asocia cada número de **syscall** con:
- el nombre del syscall
 - su función implementada en el kernel
 - y el ABI (application binary interface): common, 64, x32, etc. Este archivo es usado por el kernel durante la compilación para generar el código que mapea llamadas al sistema (por ejemplo, cuando un programa hace syscall 0 se invoca `__x64_sys_read`).

```

≡ syscall_64.tbl X
arch > x86 > entry > syscalls > ≡ syscall_64.tbl
1  # SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
2  #
3  # 64-bit system call numbers and entry vectors
4  #
5  # The format is:
6  # <number> <abi> <name> <entry point> [<compat entry point> [noreturn]]
7  #
8  # The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
9  #
10 # The abi is "common", "64" or "x32" for this file.
11 #
12 0   common  read      sys_read
13 1   common  write     sys_write
14 2   common  open      sys_open
15 3   common  close     sys_close
16 4   common  stat      sys_newstat
17 5   common  fstat     sys_newfstat
18 6   common  lstat     sys_newlstat

```

6. ¿Para qué sirve la herramienta `strace`? ¿Cómo se usa? Es una herramienta para depurar, analizar y entender lo que hace un programa en Linux a nivel del sistema operativo. Permite ver en tiempo real las **system calls* que realiza un programa. Se usa de la siguiente forma:
- `strace ./mi_programa`: se verán todas las llamadas que realiza el programa `mi_programa`.
 - `strace -e openat ./mi_programa`: para ver todas las systemcalls `open` que se realizan desde `mi_programa`.
 - `strace -o salida.txt ./mi_programa`: para almacenar la salida en un archivo.
 - `strace -p <PID>`: adjuntarse a un proceso ya en ejecución.
7. ¿Para qué sirve la herramienta `ausyscall`? ¿Cómo se usa? `ausyscall` forma parte del sistema de auditoría de Linux. Permite:
- Consultar el número de syscall a partir de su nombre,
 - consultar el nombre de syscall a partir de número,
 - usarlo para escribir reglas de auditoría con `auditctl` o analizar logs de `auditd`.

Práctica guiada

Agregamos una nueva System Call

1. Código de `my_sys_call.c` • **Para qué sirven los macros `SYS_CALL_DEFINE`?** Se utiliza para definir nuevas llamadas al sistema dentro del kernel. Estos macros generan la función correcta con el nombre esperado por el sistema para poder invocarla desde espacio de usuario.

- **¿Para que se utilizan la macros `for_each_process` y `for_each_thread`?**

- `for_each_process`: se utiliza para recorrer todos los procesos del sistema. Itera sobre cada estructura `task_struct` que representa un proceso en el scheduler del kernel.
- `for_each_thread`: recorre todos los hilos de un proceso específico. Permite iterar sobre los threads que comparten el mismo `thread group`.

- **¿Para que se utiliza la función `copy_to_user`?** Se usa para copiar datos desde el espacio del kernel hacia el espacio de usuario dado que ambos ocupan manejan espacios de memoria separados entonces no se puede acceder directamente a punteros del usuario desde el kernel.

- **¿Para qué se utiliza la función `printk`?, ¿porque no la típica `printf`?** `printk` es la función que se usa en el kernel de Linux para imprimir mensajes de depuración o información al log del sistema. No se puede usar `printf` en el kernel porque `printf` pertenece al espacio de usuario (por ejemplo, lo usa `glibc`).

- **¿Podría explicar que hacen las `system call` que hemos incluido?**

- `my_sys_call(int arg)`: Es una syscall de prueba que simplemente recibe un número y lo imprime en el log del kernel usando `printk`.
- `get_task_info(char __user *buffer, size_t length)`: Recorre todos los procesos activos del sistema y guarda su información (PID, nombre, estado) en un buffer del kernel. Luego, copia ese contenido al espacio de usuario para que pueda ser leído por una aplicación. Es útil para obtener una lista de procesos en ejecución desde el usuario, de forma similar a `ps`.
- `get_threads_info(char __user *buffer, size_t length)`: Hace algo similar a la anterior, pero además de los procesos, también lista todos los hilos (threads) de cada proceso. Esto proporciona una vista más completa del estado del sistema, incluyendo los hilos de cada proceso.

2. Modificaremos uno de los archivos Makefile del código del Kernel para indicar la compilación de nuestro código agregado en el paso anterior: `kernel/Makefile`

```
obj-y = fork.o exec_domain.o panic.o \
      cpu.o exit.o softirq.o resource.o \
      sysctl.o capability.o ptrace.o user.o \
      signal.o sys.o umh.o workqueue.o pid.o task_work.o \
      extable.o params.o \
      kthread.o sys_ni.o nsproxy.o \
      notifier.o ksfs.o cred.o reboot.o \
      async.o range.o smpboot.o ucount.o regset.o \
      my_sys_call.o
```

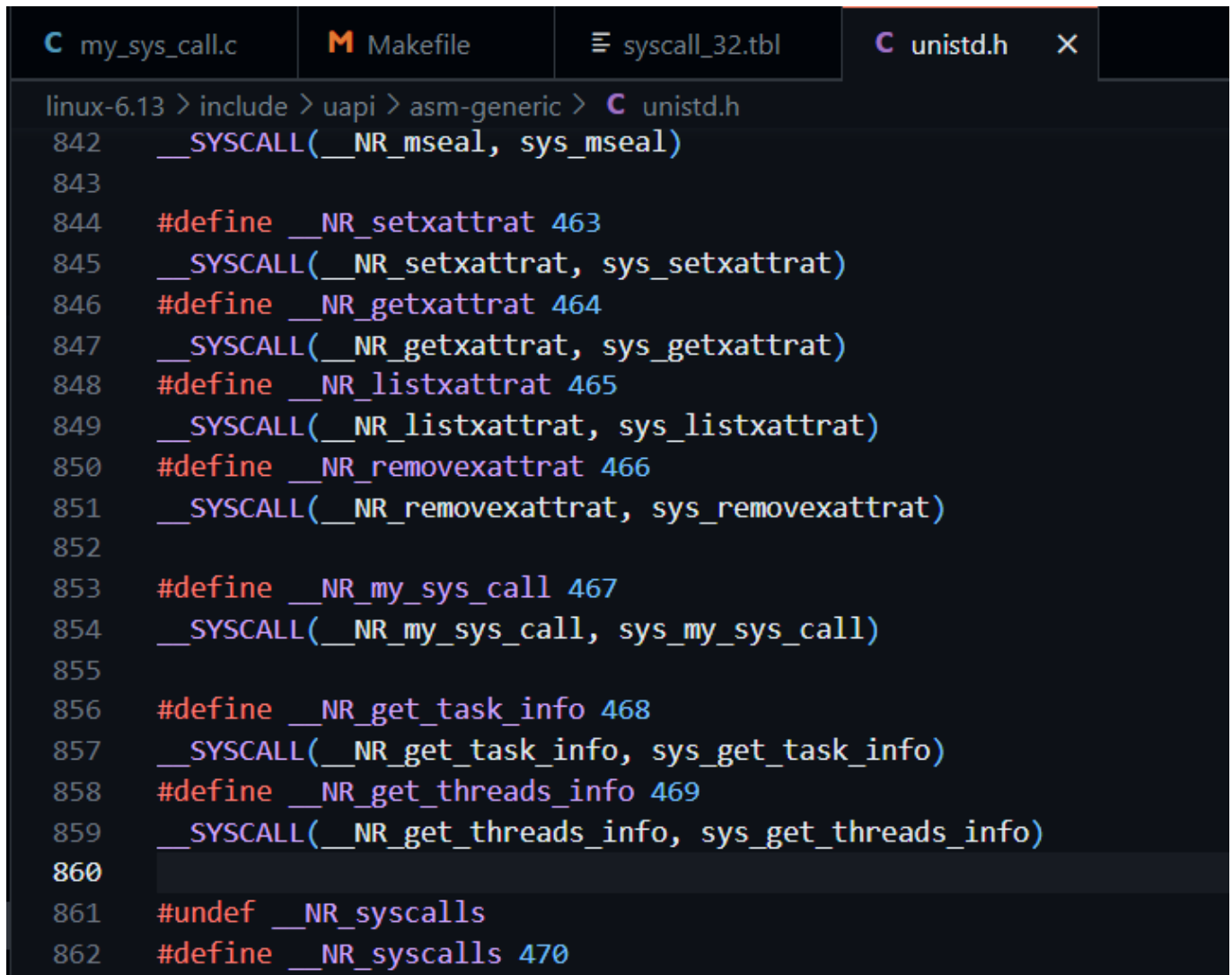
3. Añadir una entrada al final de la tabla que contiene todas las Syscalls.

```

C my_sys_call.c  M Makefile  syscall_32.tbl  syscall_64.tbl X
linux-6.13 > arch > x86 > entry > syscalls > syscall_64.tbl
375 449 common futex_waitv sys_futex_waitv
376 450 common set_mempolicy_home_node sys_set_mempolicy_home_node
377 451 common cachestat sys_cachestat
378 452 common fchmodat2 sys_fchmodat2
379 453 common map_shadow_stack sys_map_shadow_stack
380 454 common futex_wake sys_futex_wake
381 455 common futex_wait sys_futex_wait
382 456 common futex_requeue sys_futex_requeue
383 457 common statmount sys_statmount
384 458 common listmount sys_listmount
385 459 common lsm_get_self_attr sys_lsm_get_self_attr
386 460 common lsm_set_self_attr sys_lsm_set_self_attr
387 461 common lsm_list_modules sys_lsm_list_modules
388 462 common mseal sys_mseal
389 463 common setxattrat sys_setxattrat
390 464 common getxattrat sys_getxattrat
391 465 common listxattrat sys_listxattrat
392 466 common removexattrat sys_removexattrat
393 467 common my_sys_call sys_my_sys_call
394 468 common get_task_ingo sys_get_task_info
395 469 common get_thread_info sys_get_threads_info

```

Agregar los headers al vector manejador de System calls:



```
linux-6.13 > include > uapi > asm-generic > C unistd.h
842  __SYSCALL(__NR_mseal, sys_mseal)
843
844  #define __NR_setxattrat 463
845  __SYSCALL(__NR_setxattrat, sys_setxattrat)
846  #define __NR_getxattrat 464
847  __SYSCALL(__NR_getxattrat, sys_getxattrat)
848  #define __NR_listxattrat 465
849  __SYSCALL(__NR_listxattrat, sys_listxattrat)
850  #define __NR_removexattrat 466
851  __SYSCALL(__NR_removexattrat, sys_removexattrat)
852
853  #define __NR_my_sys_call 467
854  __SYSCALL(__NR_my_sys_call, sys_my_sys_call)
855
856  #define __NR_get_task_info 468
857  __SYSCALL(__NR_get_task_info, sys_get_task_info)
858  #define __NR_get_threads_info 469
859  __SYSCALL(__NR_get_threads_info, sys_get_threads_info)
860
861  #undef __NR_syscalls
862  #define __NR_syscalls 470
```

4. Compilar el kernel de nuevo. Se ejecutó mediante

```
make -jX
make modules_install
make install
```

5. Verificación de las system calls que ya sean parte del kernel

```
grep get_task_info "/boot/System.map-$(uname -r)"
```

```
so@so:~$ grep get_task_info "/boot/System.map-$(uname -r)"
ffffffff810fc310 t __pfx___do_sys_get_task_info
ffffffff810fc320 t __do_sys_get_task_info
ffffffff810fc530 T __pfx___x64_sys_get_task_info
ffffffff810fc540 T __x64_sys_get_task_info
ffffffff810fc560 T __pfx___ia32_sys_get_task_info
ffffffff810fc570 T __ia32_sys_get_task_info
ffffffff82653dc0 d event_exit__get_task_info
ffffffff82653e40 d event_enter__get_task_info
ffffffff82653ec0 d __syscall_meta__get_task_info
ffffffff82653f00 d args__get_task_info
ffffffff82653f10 d types__get_task_info
ffffffff82f3aac0 d __event_exit__get_task_info
ffffffff82f3aac8 d __event_enter__get_task_info
ffffffff82f3f188 d __p_syscall_meta__get_task_info
```

6. Creación del script que pone a prueba la ejecución de la System Call

```
so@so:~/practica2$ ls
prueba_syscall.c
so@so:~/practica2$ gcc -o ./prueba_obtencion_info prueba_syscall.c
so@so:~/practica2$ ./prueba_obtencion_info
```

Información de los procesos en ejecución :

```
-----
PID: 1 | Nombre: systemd | Estado: 1
PID: 2 | Nombre: kthreadd | Estado: 1
PID: 3 | Nombre: pool_workqueue_ | Estado: 1
PID: 4 | Nombre: kworker/R-rcu_g | Estado: 8
PID: 5 | Nombre: kworker/R-sync_ | Estado: 8
PID: 6 | Nombre: kworker/R-slub_ | Estado: 8
PID: 7 | Nombre: kworker/R-netns | Estado: 8
PID: 9 | Nombre: kworker/0:1 | Estado: 0
PID: 10 | Nombre: kworker/0:0H | Estado: 8
PID: 12 | Nombre: kworker/R-mm_pe | Estado: 8
PID: 13 | Nombre: rcu_tasks_kthre | Estado: 8
PID: 14 | Nombre: rcu_tasks_rude_ | Estado: 8
PID: 15 | Nombre: rcu_tasks_trace | Estado: 8
PID: 16 | Nombre: ksoftirqd/0 | Estado: 1
PID: 17 | Nombre: rcu_preempt | Estado: 8
PID: 18 | Nombre: rcu_exp_par_gp_ | Estado: 1
PID: 19 | Nombre: rcu_exp_gp_kthr | Estado: 1
PID: 20 | Nombre: migration/0 | Estado: 1
PID: 21 | Nombre: idle_inject/0 | Estado: 1
PID: 22 | Nombre: cpuhp/0 | Estado: 1
PID: 23 | Nombre: cpuhp/1 | Estado: 1
PID: 24 | Nombre: idle_inject/1 | Estado: 1
PID: 25 | Nombre: migration/1 | Estado: 1
PID: 26 | Nombre: ksoftirqd/1 |
```

Makefile:

```
init: prueba_syscall.c
    gcc -o prueba_syscall prueba_syscall.c
run: prueba_syscall
    ./prueba_syscall
clean:
    rm -f prueba_syscall
```

Monitoreando System Calls

1. Ejecuta el programa anteriormente compilado

```
./get_task_info
```

2. Ejecute

```
sudo dmesg
```

```
root@so:/home/so/kernel/practica2# dmesg
[ 0.000000] Linux version 6.13.7+ (root@so) (gcc (Debian 12.2.0-14) 12.2.0, GNU ld (GNU Binutils for Debian) 2.40) #4 SMP PREEMPT_DYNAMIC Sun Apr  6 16:06:37 -03 2025
[ 0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-6.13.7+ root=UUID=86ce1580-7b2a-4a7e-a89b-8b8e030dda5a ro quiet
[ 0.000000] [Firmware Bug]: TSC doesn't count with P0 frequency!
[ 0.000000] BIOS-provided physical RAM map:
[ 0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[ 0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
[ 0.000000] BIOS-e820: [mem 0x00000000000f0000-0x000000000000ffffff] reserved
[ 0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000007fffff] usable
[ 0.000000] BIOS-e820: [mem 0x00000000007fff0000-0x00000000007fffff] ACPI data
[ 0.000000] BIOS-e820: [mem 0x00000000fec00000-0x00000000fec0ffff] reserved
[ 0.000000] BIOS-e820: [mem 0x00000000fee00000-0x00000000fee0ffff] reserved
[ 0.000000] BIOS-e820: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
[ 0.000000] NX (Execute Disable) protection: active
[ 0.000000] APIC: Static calls initialized
[ 0.000000] SMBIOS 2.5 present.
[ 0.000000] DMI: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[ 0.000000] DMI: Memory slots populated: 0/0
[ 0.000000] Hypervisor detected: KVM
[ 0.000000] kvm-clock: Using msrs 4b564d01 and 4b564d00
[ 0.000011] kvm-clock: using sched offset of 3332897190333 cycles
[ 0.000022] clocksource: kvm-clock: mask: 0xffffffffffffffff max_cycles: 0x1cd42e4dffb, max_idle_ns: 881590591483 ns
[ 0.000033] tsc: Detected 2095.994 MHz processor
```

```
[ 9.222543] Console: switching to colour frame buffer device 160x50
[ 9.232255] vmwgfx 0000:00:02.0: [drm] fb0: vmwgfxdrmfb frame buffer device
[ 1240.169345] PID: 1, Nombre: systemd
[ 1240.169987] PID: 2, Nombre: kthreadd
[ 1240.169991] PID: 3, Nombre: pool_workqueue_
[ 1240.169994] PID: 4, Nombre: kworker/R-rcu_g
[ 1240.169997] PID: 5, Nombre: kworker/R-sync_
[ 1240.170000] PID: 6, Nombre: kworker/R-slub_
[ 1240.170003] PID: 7, Nombre: kworker/R-netns
[ 1240.170006] PID: 9, Nombre: kworker/0:1
[ 1240.170009] PID: 10, Nombre: kworker/0:0H
[ 1240.170012] PID: 12, Nombre: kworker/R-mm_pe
[ 1240.170015] PID: 13, Nombre: rcu_tasks_kthre
[ 1240.170018] PID: 14, Nombre: rcu_tasks_rude_
[ 1240.170021] PID: 15, Nombre: rcu_tasks_trace
[ 1240.170024] PID: 16, Nombre: ksoftirqd/0
[ 1240.170027] PID: 17, Nombre: rcu_preempt
[ 1240.170030] PID: 18, Nombre: rcu_exp_par_gp_
[ 1240.170032] PID: 19, Nombre: rcu_exp_gp_kthr
[ 1240.170035] PID: 20, Nombre: migration/0
[ 1240.170038] PID: 21, Nombre: idle_inject/0
[ 1240.170042] PID: 22, Nombre: cpuhp/0
[ 1240.170044] PID: 23, Nombre: cpuhp/1
```

`dmesg` muestra mensajes del kernel de Linux, del ring buffer (buffer circular: estructura de datos que tiene un tamaño fijo, cuando llega al final, vuelve al inicio y empieza a sobrescribir los datos más antiguos). El output son mensajes generados por el kernel. Cada línea normalmente tiene este formato:

```
[ 123.456789] usb 1-1: new high-speed USB device number 4 using ehci-pci
```

Explicación:

- `[123.456789]` → Tiempo desde que se encendió el sistema (uptime en segundos).
- `usb 1-1:` → El componente del sistema que generó el mensaje (en este caso, el controlador USB).
- `new high-speed USB device...` → Descripción del evento (acá, detectó un nuevo dispositivo USB).

3. Ejecute el programa anterior con la siguiente herramienta:

```
#En mi caso el programa se llama de esta forma
strace ./prueba_syscall
```

La salida se encuentra en [salida.txt](#), línea 30. Al ejecutar el echo nos sale el número de system call que se ejecutó

```
syscall_0x1d4(0x7ffcf974cfb0, 0x400, 0x563119415dd8, 0, 0x7f5087916680, 0x7f5087944ad0) = 0x400
newfstatat(1, "", {st_mode=S_IFIFO|0600, st_size=0, ...}, AT_EMPTY_PATH) = 0
getrandom("\x0a\xc7\x30\xda\x36\x8e\x7c\xe", 8, GRND_NONBLOCK) = 8
brk(NULL) = 0x563154595000
brk(0x5631545b6000) = 0x5631545b6000
write(1, "\nInformaci303\263n de los procesos en"... , 1168) = -1 EPIPE (Tubería rota)
--- SIGPIPE {si_signo=SIGPIPE, si_code=SI_USER, si_pid=7685, si_uid=1000} ---
+++ killed by SIGPIPE +++
so@so:~/kernel/practica2$ strace -o salida.txt ./prueba_syscall

Información de los procesos en ejecución :
-----
PID: 1 | Nombre: systemd | Estado: 1
PID: 2 | Nombre: kthreadd | Estado: 1
PID: 3 | Nombre: pool_workqueue_ | Estado: 1
PID: 4 | Nombre: kworker/R-rcu_g | Estado: 8
PID: 5 | Nombre: kworker/R-sync_ | Estado: 8
PID: 6 | Nombre: kworker/R-slub_ | Estado: 8
PID: 7 | Nombre: kworker/R-netns | Estado: 8
PID: 10 | Nombre: kworker/0:0H | Estado: 8
PID: 12 | Nombre: kworker/R-mm_pe | Estado: 8
PID: 13 | Nombre: rcu_tasks_kthre | Estado: 8
PID: 14 | Nombre: rcu_tasks_rude_ | Estado: 8
PID: 15 | Nombre: rcu_tasks_trace | Estado: 8
PID: 16 | Nombre: ksoftirqd/0 | Estado: 1
PID: 17 | Nombre: rcu_preempt | Estado: 8
PID: 18 | Nombre: rcu_exp_par_gp_ | Estado: 1
PID: 19 | Nombre: rcu_exp_gp_kthr | Estado: 1
PID: 20 | Nombre: migration/0 | Estado: 1
PID: 21 | Nombre: idle_inject/0 | Estado: 1
PID: 22 | Nombre: cpuhp/0 | Estado: 1
PID: 23 | Nombre: cpuhp/1 | Estado: 1
PID: 24 | Nombre: idle_inject/1 | Estado: 1
PID: 25 | Nombre: migration/1 | Estado: 1
PID: 26 | Nombre: ksoftirqd/1 | Estado: 1
PID: 29 | Nombre: cpuhp/2 | Es
-----
so@so:~/kernel/practica2$ echo $((0x1d4))
468
```

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? Se denomina **módulo del kernel** o **kernel Module**. Suelen tener extensión **.ko** (kernel object). Pueden cargarse y descargarse en tiempo de ejecución.

```
sudo insmod mi_modulo.ko      # Carga el módulo
sudo rmmod mi_modulo          # Lo descarga
```

En caso de que el kernel no admitiera módulos (monolítico sin soporte de carga dinámica) la forma de agregar más funcionalidad sería así: 1. Agregar el código directamente al kernel fuente. 2. Recompilar el kernel completo. 3. Reiniciar el sistema.

2. ¿Qué es un driver? ¿Para qué se utiliza? Un **driver** es un módulo o programa que le dice al sistema operativo cómo interactuar con un hardware específico. Sirve para que el sistema pueda usar hardware de forma genérica, sin tener que saber cómo funciona internamente. Se pueden cargar con **modprobe**. Aparecen en **/lib/modules\$(uname -r)/**
3. ¿Por qué es necesario escribir drivers? Dado que cada dispositivo es diferente y tiene su propia forma de funcionar el sistema operativo no puede incluir soporte nativo para todos los dispositivos. A parte, favorece al aislamiento y modularidad ya que separan el código del sistema operativo del manejo específico del hardware. Un drive generalmente está implementado como módulo del kernel pero también puede formar parte del núcleo desde el arranque (incluido directamente en el archivo **vmlinuz**)
4. ¿Cuál es la relación entre módulo y driver en GNU/Linux? Un driver es el software que permite al sistema operativo comunicarse con un dispositivo (como una impresora, tarjeta de red, USB, etc). Un módulo es una forma de cargar ese driver en el kernel dinámicamente, en tiempo de ejecución. Entonces: Un driver puede estar implementado como un módulo del kernel.
5. ¿Qué implicancias puede tener un bug en un driver o módulo? Debe tenerse en cuenta un driver corre en el espacio del **modo privilegiado** o **modo kernel** por lo tanto no hay protección de memoria, tienen acceso total a todos los recursos del sistema. Puede implicar:
 - Bloqueo del sistema (kernel panic)
 - Caída del rendimiento
 - Corrupción de memoria o datos
 - Vulnerabilidades de seguridad
6. ¿Qué tipos de drivers existen en GNU/Linux?

Tipos de drivers	Detalle
de dispositivos (device drivers)	controlan dispositivos físicos

Tipos de drivers	Detalle
de pseudo-dispositivos (virtuales)	emulan dispositivos. Ej: <code>/dev/null</code> descarta todo lo que se escribe en él; <code>/dev/zero</code> devuelve una secuencia infinita de ceros; <code>ramdisk</code> usa memoria RAM como si fuera un disco.
de sistema de archivos	manejan como se leen/escriben datos en distintos sistemas de archivos
de bus	controlan como se comunican los dispositivos conectados mediante buses. Ej: <code>usbcore</code> para USB; <code>pci_generic</code> para PCI.
de virtualización / hypervisores	permiten que Linux corra sobre o dentro de máquinas virtuales
de energía / sensores	controlan funciones como la administración de energía, de ventiladores, de temperatura.

7. ¿Qué hay en el directorio `/dev`? ¿Qué tipos de archivo encontramos en esa ubicación? Contiene archivos de dispositivos. Son interfaces entre el sistema operativo y los dispositivos. Hay dos tipos principales de archivos de dispositivo:

- **Archivos de dispositivo de carácter (chardevice) - c:** transfieren datos byte a byte (como un flujo). Ej: teclados, mouse, puertos serie.
- **Archivos de dispositivo de bloque (block device) - b:** transfieren datos por bloques con acceso aleatorio. Ej: discos, pendrives.

8. ¿Para qué sirven el archivo `/lib/modules/<version>/modules.dep` utilizado por el comando `modprobe`? Es un archivo de dependencias que mantiene una lista de qué módulos del kernel dependen de otros. Cuando se quiere cargar un módulo con:

```
modprobe nombre_modulo
```

`modprobe`: consulta `modules.dep`, ve si ese módulo necesita otros módulos para funcionar y en caso de que así sea, carga todas las dependencias en el orden correcto.

9. ¿En qué momento/s se genera o actualiza un `initramfs`? `initramfs` es un archivo comprimido que contiene un pequeño sistema de archivos usado temporalmente durante el arranque del sistema y se almacena en `/boot/initrd-img-<version>`. Se genera o actualiza:

- al instalar o actualizar el kernel
- al instalar nuevos módulos (necesarios para el arranque) del kernel
- al modificar configuraciones relacionadas al arranque

10. ¿Qué módulos y drivers deberá tener un `initramfs` mínimamente para cumplir su objetivo? Como el rol del `initramfs` es permitir que se prepare lo mínimo necesario para que el sistema pueda montar y

pasarle el control al sistema operativo entonces mínimamente el arranque del sistema deberá incluir los siguientes módulos:

- drivers de almacenamiento: permitir acceder al disco físico donde está el sistema.
- drivers de sistema de archivos: permiten montar el sistema de archivos raíz (/) como ext4, btrfs, vfat.
- módulos del bus del sistema: para reconocer el hardware al que están conectados los discos.

```
root@so:/# lsinitramfs /boot/initrd.img-$(uname -r) | grep -E '\.ko$'
usr/lib/modules/6.13.7+/kernel/arch/x86/crypto/crc32c-intel.ko
usr/lib/modules/6.13.7+/kernel/drivers/ata/acard-ahci.ko
usr/lib/modules/6.13.7+/kernel/drivers/ata/ahci.ko
usr/lib/modules/6.13.7+/kernel/drivers/ata/ata_generic.ko
usr/lib/modules/6.13.7+/kernel/drivers/ata/ata_piix.ko
usr/lib/modules/6.13.7+/kernel/drivers/ata/libahci.ko
usr/lib/modules/6.13.7+/kernel/drivers/ata/libata.ko
usr/lib/modules/6.13.7+/kernel/drivers/cdrom/cdrom.ko
usr/lib/modules/6.13.7+/kernel/drivers/hid/hid-generic.ko
usr/lib/modules/6.13.7+/kernel/drivers/hid/hid.ko
usr/lib/modules/6.13.7+/kernel/drivers/hid/usbhid/usbhid.ko
usr/lib/modules/6.13.7+/kernel/drivers/i2c/busses/i2c-piix4.ko
usr/lib/modules/6.13.7+/kernel/drivers/i2c/i2c-smbus.ko
usr/lib/modules/6.13.7+/kernel/drivers/input/mouse/psmouse.ko
usr/lib/modules/6.13.7+/kernel/drivers/md/dm-mod.ko
usr/lib/modules/6.13.7+/kernel/drivers/net/ethernet/intel/e1000/e1000.ko
usr/lib/modules/6.13.7+/kernel/drivers/scsi/scsi_common.ko
usr/lib/modules/6.13.7+/kernel/drivers/scsi/scsi_mod.ko
usr/lib/modules/6.13.7+/kernel/drivers/scsi/sd_mod.ko
usr/lib/modules/6.13.7+/kernel/drivers/scsi/sg.ko
usr/lib/modules/6.13.7+/kernel/drivers/scsi/sr_mod.ko
usr/lib/modules/6.13.7+/kernel/drivers/usb/common/usb-common.ko
usr/lib/modules/6.13.7+/kernel/drivers/usb/core/usbcore.ko
usr/lib/modules/6.13.7+/kernel/drivers/usb/host/ehci-hcd.ko
usr/lib/modules/6.13.7+/kernel/drivers/usb/host/ehci-pci.ko
usr/lib/modules/6.13.7+/kernel/drivers/usb/host/ohci-hcd.ko
usr/lib/modules/6.13.7+/kernel/drivers/usb/host/ohci-pci.ko
usr/lib/modules/6.13.7+/kernel/fs/ext4/ext4.ko
usr/lib/modules/6.13.7+/kernel/fs/jbd2/jbd2.ko
usr/lib/modules/6.13.7+/kernel/fs/mbcache.ko
usr/lib/modules/6.13.7+/kernel/lib/crc16.ko
root@so:/#
```

Módulos actuales del sistema

Práctica guiada

Desarrollo de un módulo simple

1. Crear el archivo memory.c con el siguiente contenido:

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

MODULE_LICENSE("Dual BSD/GPL");
```

2. Crear el archivo Makefile con el siguiente contenido:

```
obj-m := memory.o //
```

Responder:

- Explique brevemente cual es la utilidad del archivo **Makefile**. Sirve para compilar el módulo.
 - ¿Para qué sirve la macro **MODULE_LICENSE**? ¿Es obligatoria? Se usa para indicar la licencia bajo la cual se distribuye el módulo. No es obligatoria pero es recomendable incluirla para evitar problemas legales y de compatibilidad con el kernel.
3. Compilar el módulo usando el mismo kernel en que correrá el mismo, utilizaremos el que instalaos en el primer paso del ejercicio guiado.

```
make -C <KERNEL_CODE> M=$(pwd) modules //En KERNEL_CODE se debe poner la ruta del kernel que se descargo.
```

a. ¿Cuál es la salida del comando anterior?

```
root@so:/home/so/practica2# make -C /home/so/kernel/linux-6.13 M=$(pwd) modules
make: se entra en el directorio '/home/so/kernel/linux-6.13'
make[1]: se entra en el directorio '/home/so/practica2'
  CC [M]  memory.o
  MODPOST Module.symvers
  CC [M]  memory.mod.o
  CC [M]  .module-common.o
  LD [M]  memory.ko
make[1]: se sale del directorio '/home/so/practica2'
make: se sale del directorio '/home/so/kernel/linux-6.13'
```

b. ¿Qué tipos de archivo se generan? Explique para qué sirve cada uno. Se generaron los siguientes archivos:

- **memory.o**: es el objeto del módulo compilado. Contiene el código máquina del módulo.
- **memory.mod.o**: contiene información adicional sobre el módulo, como su nombre, versión y dependencias.
- **memory.ko**: es el módulo del kernel compilado (kernel object). Es el archivo que se carga en el kernel para usar la funcionalidad del módulo.

c. Con lo visto en la Práctica 1 sobre Makefiles, construya un Makefile de manera que si ejecuto i. **make**, nuestro módulo se compila ii. **make clean**, limpia el módulo y el código objeto generado iii. **make run**, ejecuta el programa

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

```
insmod memory.ko
```

Responda lo siguiente: ¿Para qué sirven el comando **insmod** y el comando **modprobe**? ¿En qué se diferencian? Tanto **insmod** como **modprobe** se utilizan para cargar módulos en el kernel. - **insmod**: carga un

módulo específico en el kernel. No resuelve dependencias automáticamente, por lo que si el módulo requiere otros módulos, deben ser cargados manualmente primero. - **modprobe**: carga un módulo y resuelve automáticamente las dependencias necesarias. Si el módulo requiere otros módulos, **modprobe** los carga en el orden correcto. Es más conveniente para manejar módulos con dependencias.

5. Verificamos que el módulo se haya cargado correctamente:

```
lsmod | grep memory
```

Salida:

```
root@so:/home/so/practica2# lsmod | grep memory
memory                8192    0
```

- a. ¿Cuál es la salida del comando? Explique cuál es la utilidad del comando **lsmod**. Indica que el módulo **memory** se ha cargado correctamente en el kernel. **lsmod** muestra una lista de todos los módulos del kernel que están actualmente cargados en el sistema. Proporciona información sobre el nombre del módulo, su tamaño y cuántas veces está siendo utilizado (número de referencias).
- b. ¿Qué información encuentra en el archivo **/proc/modules**? Es un archivo virtual del sistema, que contiene información en tiempo real sobre los módulos cargados, es la fuente directa de donde **lsmod** obtiene la información y la formatea.
- 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
```

De los fragmentos se puede ver el nombre del módulo, su tamaño, el número de referencias (0 significa que no hay otros módulos que dependan de él), y el estado del módulo (Live indica que está activo). También se puede ver la dirección de memoria donde está cargado el módulo.

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

```
rmmod memory
```

6. Descargue el módulo **memory**. Para corroborar ejecute:

```
lsmod | grep memory
```

7. Modifique el archivo memory.c:

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

static int hello_init(void) {
    printk(KERN_INFO "Hello, world!\n");
    return 0;
}

static void hello_exit(void) {
    printk("Bye, cruel world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

- Cargar el modulo, ejecutar `dmesg`, descargar el modulo y ejecutar `dmesg` nuevamente. ¿Qué diferencias encuentra?

```
[13495.165271] memory: module verification failed: signature and/or required key missing - tainting kernel
[14521.508503] Hello, world!
[14848.161956] Bye, cruel world!
[14921.440519] Hello, world!
[14942.725550] Bye, cruel world!
root@so:/home/so/practica2#
```

8. Responder: a. ¿Para qué sirve la función `module_init`? ¿Y `module_exit`? ¿Cómo haría para ver la información del log que arrojan las mismas? `module_init` se usa para indicar la función que se ejecutará cuando el módulo se cargue en el kernel. `module_exit` indica la función que se ejecutará cuando el módulo se descargue del kernel. Para ver la información del log que arrojan se puede usar `dmesg` o `cat /var/log/kern.log`.

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 faltaría:

- Implementar la lógica específica del hardware que se va a controlar.
- Manejar la comunicación entre el kernel y el hardware (lectura/escritura de datos).

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

- **Dispositivos de bloque:** permiten acceso aleatorio a bloques de datos. Ej: discos duros, pendrives. Se accede a ellos mediante bloques de tamaño fijo.
- **Dispositivos de carácter:** permiten acceso secuencial a datos. Ej: teclados, mouse. Se accede a ellos byte por byte y 1 byte solo puede ser leído por única vez.

Desarrollo de un driver

Ahora completamos nuestro módulo para agregarle la capacidad de escribir y leer un dispositivo. En nuestro caso el dispositivo a leer será la memoria de nuestra CPU, pero podría ser cualquier otro dispositivo.

El mayor number identifica qué driver del kernel maneja un determinado dispositivo. Cuando se accede a un archivo en `/dev`, el kernel usa el número mayor para saber a qué driver llamar.

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. Responder: a. ¿Para qué sirve la estructura `ssize_t` y `memory_fops`? ¿Y las funciones `register_chrdev` y `unregister_chrdev`? `ssize_t` es un tipo de dato entero con signo. Se utiliza para representar el tamaño de un objeto en bytes leídos o escritos exitosamente (valor positivo) o un código de error (valor negativo). En el código se usa: si la función `memory_read()` devuelve 1, significa que leyó 1 byte, en cambio si devuelve `-EFAULT`, hubo un error de acceso a memoria del usuario. `memory_fops` es una estructura del kernel que define las funciones que implementa el driver para operaciones comunes de archivos. En el caso del código se definen las operaciones y se le pasan las funciones que se implementaron.

Las funciones `register_chrdev` y `unregister_chrdev` se utilizan para registrar y anular el registro de un controlador de caracteres (character device driver) en el kernel.

- `register_chrdev`: registra un dispositivo de caracteres con un numero mayor. Le da un nombre y las funciones que se usarán para manejar las operaciones de lectura y escritura.
- `unregister_chrdev`: anula el registro del dispositivo de caracteres. Limpio todo cuando el módulo se remueve (`rmmod`).

b. ¿Cómo sabe el kernel que funciones del driver invocar para leer y escribir al dispositivo? Lo sabe por la estructura `file_operations` que se le pasa a `register_chrdev`. Esta estructura contiene punteros a las funciones que implementan las operaciones.

c. ¿Cómo se accede desde el espacio de usuario a los dispositivos en Linux? Los dispositivos se acceden a través de archivos especiales ubicados en `/dev`. Estos archivos representan los dispositivos y permiten interactuar con ellos como si fueran archivos normales. Por ejemplo, desde el espacio de usuario se puede usar `open()` para abrir un archivo de dispositivo y su equivalente en bash sería `cat`, `echo`, etc.

d. ¿Cómo se asocia el módulo que implementa nuestro driver con el dispositivo? Se da en `register_chrdev`. Se le pasa el número mayor y el nombre del dispositivo. El kernel asocia el número mayor con el módulo que implementa el driver. Cuando se accede al dispositivo, el kernel usa este número para invocar las funciones definidas en `memory_fops`. Luego se crean un archivo en `/dev` con el nombre del dispositivo.

```
mknod /dev/memory c <major_number> 0
```

Con `unregister_chrdev` se desasocia el módulo del dispositivo.

e. ¿Qué hacen las funciones `copy_to_user` y `copy_from_user`? (<https://developer.ibm.com/articles/l-kernel-memory-access/>) [^2].

- `copy_to_user`: se encarga de copiar datos desde el espacio del kernel al espacio de usuario.

- **copy_from_user**: se encarga de copiar datos desde el espacio de usuario al espacio del kernel. Ambas funciones son necesarias porque el kernel y el espacio de usuario tienen diferentes espacios de memoria y no pueden acceder directamente a los datos del otro. Estas funciones manejan la transferencia de datos entre ambos espacios de manera segura, verificando permisos y evitando accesos no autorizados.

3. Ejecutar:

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

4. Ejecutar:

```
insmod memory.ko
```

Responder: a. ¿Para qué sirve el comando **mknod**? ¿qué especifican cada uno de sus parámetros? Sirve para crear archivos de dispositivo en **/dev**.

```
mknod [ruta] [tipo] [mayor] [menor]
```

- **ruta**: ruta donde se creará el archivo de dispositivo (ej: **/dev/memory**).
- **tipo**: tipo de dispositivo (c para carácter, b para bloque).
- **mayor**: número mayor del dispositivo (identifica el driver que maneja el dispositivo).
- **menor**: número menor del dispositivo (identifica una instancia específica del dispositivo manejado por el driver).

b. ¿Qué son el "major number" y el "minor number"? El **major number** (número mayor) es un identificador único que asigna el kernel a un driver específico. Indica qué driver manejará el dispositivo. El **minor number** (número menor) es un identificador que permite distinguir entre diferentes instancias de un mismo dispositivo manejado por el mismo driver. Por ejemplo, si hay dos discos duros manejados por el mismo driver, cada uno tendrá un número menor diferente.

5. Ejecutar:

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

6. Ejecutar:

```
more /dev/memory
```



```

root@so:/home/so/practica2# make
make -C /home/so/kernel/linux-6.13 M=/home/so/practica2 modules
make[1]: se entra en el directorio '/home/so/kernel/linux-6.13'
make[2]: se entra en el directorio '/home/so/practica2'
  CC [M]  memory.o
  MODPOST Module.symvers
  CC [M]  memory.mod.o
  LD [M]  memory.ko
make[2]: se sale del directorio '/home/so/practica2'
make[1]: se sale del directorio '/home/so/kernel/linux-6.13'
root@so:/home/so/practica2# mknod /dev/memory c 60 0
root@so:/home/so/practica2# /sbin/insmod memory.ko
root@so:/home/so/practica2# echo -n absdef > /dev/memory
root@so:/home/so/practica2# more /dev/memory
f

```

Ejecutamos el Makefile, seguimos los pasos indicados. Se puede ver que al ser declarados como dispositivo por caracter lo que queda en el buffer es el último char escrito. En caso de ser bloque, quedaría el bloque completo.

7. Responder: a. ¿Qué salida tiene el anterior comando?, ¿Porque? (ayuda: siga la ejecución de las funciones `memory_read` y `memory_write` y verifique con `dmesg`)

```

[ 6086.806185] memory_write()
[ 6086.806195] memory_write()
[ 6086.806197] memory_write()
[ 6086.806200] memory_write()
[ 6086.806203] memory_write()
[ 6086.806205] memory_write()
[ 6096.584641] memory_read()
[ 6096.584918] memory_read()

```

La salida es `f` porque el driver está implementado para que solo se guarde el último byte escrito en el buffer. Notar que la función `memory_write` se invoca 6 veces, una por cada byte escrito.

b. ¿Cuántas invocaciones a `memory_write` se realizaron? Se realizaron 6. c. ¿Cuál es el efecto del comando anterior? ¿Por qué? El efecto es que se escriben los bytes `abcdef` en el buffer del driver. Pero como el driver está implementado para guardar solo el último byte, al final solo queda `f` en el buffer.

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. 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`.

Se deben considerar aspectos como:

- Manejo de concurrencia: usar semáforos o mutex para evitar condiciones de carrera al acceder a recursos compartidos.

- Implementar funciones `ioctl` para manejar operaciones de entrada/salida específicas del dispositivo.
- Manejo de interrupciones: usar `inb` y `outb` para leer y escribir directamente en puertos de hardware.
- Manejo de errores: verificar errores en las operaciones de lectura/escritura y devolver códigos de error apropiados.

[^2]: Modifiqué el link original. El contenido se encuentra archivado. Deprecado.