



UNIVERSITAS
Miguel Hernández

Investigación y documentación de la inclusión de una nueva llamada al sistema en Linux

Diseño de Sistemas Operativos - 4º Ingeniería Informática en
Tecnologías de la Información

Curso 2024/2025

Joan Amorós Ramírez
Miguel Soria Zaragoza
Héctor Calvo Alfosea

ÍNDICE

1. Resumen	4
2. Introducción	5
2.1. ¿Qué es una llamada al sistema?	5
2.2. Contexto y relevancia de las llamadas al sistema en Linux	5
2.3. Importancia de las llamadas al sistema en el funcionamiento del sistema operativo	5
2.4. Motivación del proyecto	6
2.5. Roles y contribuciones del equipo	6
3. Fundamentos teóricos	7
3.1. Comparativa entre llamadas al sistema y funciones de biblioteca	7
3.2. Tipos de llamadas al sistema	7
4. Arquitectura de llamadas al sistema en Linux	8
4.1. Identificación del kernel	8
4.2. Descripción del kernel	8
4.3. Estructura del kernel	9
4.4. Mecanismo de llamadas al sistema en Linux	9
4.5. Ejemplos de llamadas al sistema comunes	10
4.5.1. fork	10
4.5.2. exec	10
4.5.3. open	10
4.5.4. read	10
4.5.5. write	11
5. Implementación de llamadas al sistema	11
5.1. Preparación del entorno	11
5.2. Proceso de implementación de la llamada al sistema	11
5.3. Modificación de archivos necesarios	13
5.4. Registro de la nueva syscall	15
5.5. Interacción entre el espacio de usuario y el espacio del kernel	17
6. Pruebas y validaciones	18
6.1. Compilación del kernel con la nueva syscall	18
6.2. Pruebas funcionales de la nueva syscall	19
7. Ejemplos prácticos	20
7.1. Análisis de casos de uso comunes	20
7.2. Comparación con otras formas de interacción con el sistema operativo	21
8. Optimización y mejores prácticas	22
8.1. Técnicas para optimizar el uso de llamadas al sistema	22
8.2. Mejores prácticas para la programación de llamadas al sistema en Linux	23
8.3. Consideraciones de seguridad y rendimiento	24
8.3.1. Seguridad	24
8.3.2. Rendimiento	24
9. Herramientas y recursos	25
9.1. Herramientas de depuración y análisis de llamadas al sistema	25

9.1.1. strace	25
9.1.2. ltrace	26
9.2. Recursos adicionales para aprender más sobre llamadas al sistema en Linux	27
10. Publicación y presentación de resultados	27
11. Conclusiones	29
11.1. Importancia de las llamadas al sistema en el desarrollo de software	29
11.2. Futuras investigaciones y desarrollos	29
12. Referencias	30

1. Resumen

El propósito de este informe es analizar las llamadas al sistema en el entorno de Linux, explicando su función esencial entre el espacio de usuario y el núcleo (kernel) del sistema operativo. El documento trata sobre los fundamentos teóricos de las llamadas al sistema, que incluyen las llamadas a funciones de biblioteca.

Además, se analiza la estructura que soporta las llamadas al sistema en Linux, analizando cómo se implementan y gestionan en el núcleo. Esto incluye una explicación detallada de los procedimientos empleados, así como de llamadas comunes (como `read()`, `write()` y `fork()`) y la estructura interna del kernel que permite su funcionamiento.

El informe también describe el proceso completo para implementar una nueva llamada al sistema, desde la preparación del entorno hasta las modificaciones en archivos clave del kernel, como `syscalls.h` y `syscall_64.tbl`. Se presentan estrategias para optimizar el uso y la compilación del núcleo, junto con herramientas de depuración y análisis como `strace` y `ltrace`, que son esenciales para comprender y mejorar el rendimiento de las `syscalls`.

Finalmente, se analizan ejemplos específicos y casos de uso, ofreciendo una visión clara de cómo se emplean las llamadas al sistema en situaciones reales. Este estudio no solo busca profundizar en los aspectos técnicos de las `syscalls`, sino también fomentar una mejor comprensión de su importancia en el desarrollo de software y su impacto en el rendimiento y la seguridad del sistema operativo.

2. Introducción

2.1. ¿Qué es una llamada al sistema?

Una **llamada al sistema** es una parte fundamental del funcionamiento de cualquier sistema operativo, incluyendo Linux. Las llamadas permiten que los programas de usuario interactúen con el kernel (el núcleo del sistema operativo), facilitando tareas esenciales como la gestión de procesos, memoria y archivos. El informe tiene como objetivo proporcionar una visión detallada de las llamadas al sistema en Linux, desde sus fundamentos teóricos hasta su implementación práctica y optimización.

2.2. Contexto y relevancia de las llamadas al sistema en Linux

Las llamadas al sistema son una parte fundamental en el sistema Linux y en cualquier sistema operativo. Por ejemplo, las llamadas al sistema permiten la comunicación entre el núcleo del sistema y el espacio del usuario, permitiendo mejorar la seguridad del sistema así como utilizar mejor los recursos disponibles. Las llamadas se dividen en diferentes funcionalidades para tener aglutinadas de una forma más coherente y fácil de buscar las llamadas que necesitamos (llamadas sobre el control de procesos, gestión de archivos, etc)

2.3. Importancia de las llamadas al sistema en el funcionamiento del sistema operativo

Como se ha remarcado anteriormente, las llamadas al sistema permiten comunicar las necesidades del usuario con el núcleo del sistema. Su origen se debe a que los sistemas se hacen cada vez más complejos, y por tanto se decide hacer esta división para tener una manera de mejorar la estabilidad y seguridad del sistema, así como gestionar eficientemente los recursos.

Algunas de las consecuencias de no tener llamadas al sistema sería que los programas tendrían acceso directo a la CPU y a la memoria RAM, lo que podría dar casos de vulnerabilidad de la seguridad. No habría una manera de compartir recursos de manera eficiente entre múltiples procesos y cualquier fallo podría generar un fallo en el núcleo del sistema, cosa que desencadenaría un error muy grave en el sistema completo.

2.4. Motivación del proyecto

Este informe se ha realizado para estudiar en profundidad el funcionamiento del núcleo de Linux y su naturaleza para ser expandido con nuevas llamadas al sistema. Los objetivos son:

- Comprender mejor los conceptos teóricos de las llamadas al sistema.
- Practicar habilidades técnicas avanzadas desarrollando una llamada al sistema personalizada.
- Trabajar colaborativamente utilizando herramientas modernas como Mistral, Perplexity y Notion IA para mejorar la producción.

2.5. Roles y contribuciones del equipo

Joan Amorós Ramírez

- Creación de los agentes en Mistral
- Documentación del proyecto, organizando y redactando los contenidos clave.

Miguel Soria Zaragoza

- Implementación de la llamada al sistema, desarrollando el código fuente y realizando las pruebas necesarias.
- Búsqueda e investigación en Perplexity
- Creación de las instrucciones con la inteligencia artificial de Notion

Héctor Calvo Alfosea

- Búsqueda e investigación con Perplexity
- Creación de los agentes en Mistral junto con Joan

3. Fundamentos teóricos

Para entender el funcionamiento y la implementación de las llamadas al sistema, es importante definir los fundamentos teóricos que las respaldan. Este apartado analiza las diferencias entre las llamadas al sistema y las funciones de biblioteca, y explica los diferentes tipos de *syscalls*.

3.1. Comparativa entre llamadas al sistema y funciones de biblioteca

La diferencia principal entre una **llamada al sistema** y una **llamada a la biblioteca** se basa en el grado de ejecución y la interacción con el sistema operativo.

Las llamadas al sistema se ejecutan en modo **núcleo**, interactuando directamente con el sistema operativo para operaciones críticas como acceso a hardware, archivos o memoria. Algunos ejemplos de *syscalls* (explicados más adelante) son `fork()`, `read()` o `write()`.

Por otra parte, las funciones o llamadas a la biblioteca se ejecutan en modo **usuario**, proporcionando funciones de alto nivel para simplificar tareas comunes del programador. En muchos casos, hay funciones de biblioteca que encapsulan llamadas al sistema (por ejemplo `fopen()` internamente utiliza `open()`)

3.2. Tipos de llamadas al sistema

Las *syscalls* en Linux se pueden agrupar según su propósito y funcionalidad. Algunos de los tipos relevantes de llamadas al sistema son:

- a) **Manejo de ficheros:** permiten interactuar con el sistema de archivos (crear, leer, abrir, cerrar ficheros...)
- b) **Gestión de procesos:** sirven para controlar la creación, ejecución y terminación de procesos en el sistema operativo.
- c) **Gestión de memoria:** son útiles para asignar, liberar o manipular la memoria del sistema.
- d) **Comunicación entre procesos:** su función principal es facilitar la transferencia de datos entre procesos.
- e) **Gestión de dispositivos:** suelen interactuar con dispositivos de hardware.
- f) **De redes:** son importantes para la comunicación a través de redes.
- g) **Gestión de información del sistema:** proporcionan detalles sobre el estado y la configuración del sistema operativo.
- h) **Gestión de seguridad y usuarios:** sirven para controlar permisos y credenciales de usuarios y procesos.

4. Arquitectura de llamadas al sistema en Linux

4.1. Identificación del kernel

En el proyecto se ha optado por utilizar la versión 4.x del kernel de Linux por los siguientes motivos:

- **Estabilidad y soporte LTS:** Versiones como 4.4 o 4.14 cuentan con soporte extendido, lo que las hace ideales para entornos críticos donde la estabilidad es prioritaria.
- **Compatibilidad con hardware antiguo:** Las versiones 4.x son más adecuadas para dispositivos antiguos o sistemas integrados, que a menudo no requieren las características avanzadas de los kernels más nuevos.
- **Simplicidad:** El kernel 4.x es más ligero y menos complejo en comparación con el 5.x, lo que facilita su manejo en proyectos con recursos limitados.
- **Requisitos de software:** Algunos entornos de desarrollo o aplicaciones están optimizados específicamente para trabajar con núcleos de la serie 4.x.

4.2. Descripción del kernel

La serie 4.x del núcleo de Linux (desarrollada desde 2015) representa un avance importante en cuanto a la funcionalidad y al rendimiento. Algunas de sus características más relevantes son:

- **Actualizaciones:** Una de las mejoras más destacadas que presenta la versión 4.0 es la posibilidad de aplicar parches en caliente al kernel sin necesidad de reiniciar el sistema (Xataka, 2015). Esto es crucial en entornos críticos donde la disponibilidad continua del sistema es prioritaria.
- **Eficiente en entrada/salida:** El kernel 4.x. mejora significativamente la gestión de las operaciones de entrada y salida, permitiendo un rendimiento más consistente. (Wikipedia, s.f.)
- **Soporte de hardware:** Como se ha explicado en el apartado anterior, se amplía la compatibilidad, lo que le hace adecuado para proyectos que priorizan la compatibilidad sin comprometer el rendimiento.

4.3. Estructura del kernel

En la versión 4.0, el núcleo de Linux adopta un diseño monolítico modular, lo que significa que la mayor parte de sus funciones se encuentran integradas en un único programa que opera en el espacio del kernel. No obstante, este diseño también facilita la carga y descarga de módulos de manera dinámica, proporcionando versatilidad para incrementar sus capacidades.

La estructura del kernel está organizada en seis bloques principales (Villela, s.f.):

1. **Planificador de tareas (o scheduler):** se encarga de gestionar la ejecución de procesos y garantizar una distribución eficiente de los recursos.
2. **Gestor de memoria:** supervisa la memoria física y virtual para optimizar el rendimiento.
3. **Comunicación entre procesos (IPC):** ofrece mecanismos que permiten la comunicación entre los procesos.
4. **Sistema de archivos:** administra el acceso y la organización de los datos.
5. **Interfaz de red:** facilita las operaciones de red y las comunicaciones externas.
6. **Drivers de dispositivos:** funcionan como intermediarios entre el hardware y el software, permitiendo que el sistema interactúe con dispositivos externos.

4.4. Mecanismo de llamadas al sistema en Linux

Las llamadas al sistema se utilizan principalmente para establecer una interacción directa con el núcleo del sistema operativo. Este proceso es necesario, ya que las aplicaciones no pueden entrar directamente al hardware o realizar ciertas operaciones privilegiadas sin pasar por el núcleo.

En primer lugar, se lleva a cabo la invocación desde el espacio de usuario para efectuar la operación. Tras ello, la llamada al sistema genera una interrupción de software que transfiere el control al núcleo, que utiliza una **tabla específica** para identificar qué función del núcleo debe ejecutarse. Esta tabla (también conocida como *syscall table*) asocia el número de la llamada al sistema con la función correspondiente en el espacio del kernel.

Después el núcleo verifica los parámetros que se han proporcionado, realiza la operación solicitada y retorna el resultado (o un código de error) al programa en espacio de usuario.

Por último, una vez completada la operación, el control vuelve al programa de usuario, que recibe el resultado de la operación mediante el registro de retorno.

4.5. Ejemplos de llamadas al sistema comunes

Algunas de las llamadas al sistema más utilizadas son las siguientes:

4.5.1. fork

Una llamada al sistema frecuente es `fork()`, que crea un nuevo proceso llamado **proceso hijo**, siendo éste una copia del proceso que realiza la llamada (**proceso padre**)

```
pid_t pid = fork();
if (pid == 0) {
    // Código del proceso hijo
} else {
    // Código del proceso padre
}
```

4.5.2. exec

Cuando se utiliza `exec()`, se reemplaza el contenido del espacio de memoria del proceso actual con el programa especificado.

```
int main() {
    execl("/bin/ls", "ls", "-l", (char *)NULL);
    return 0;
}
```

4.5.3. open

La llamada al sistema `open()` abre un fichero en el sistema de archivos y devuelve un descriptor de archivo para interactuar con él:

```
int fd = open("archivo.txt", O_RDONLY);
```

4.5.4. read

La llamada al sistema `read` se utiliza para leer datos de un archivo o entrada estándar (como el teclado) en un buffer del espacio de usuario. Es fundamental para interactuar con archivos y dispositivos en Linux.

```
ssize_t bytes_leidos = read(fd, buffer, sizeof(buffer) - 1);
```

4.5.5. write

Esta llamada se utiliza para escribir datos desde un buffer del espacio de usuario a un archivo:

```
ssize_t bytes_escritos = write(fd, mensaje, sizeof(mensaje));
```

5. Implementación de llamadas al sistema

5.1. Preparación del entorno

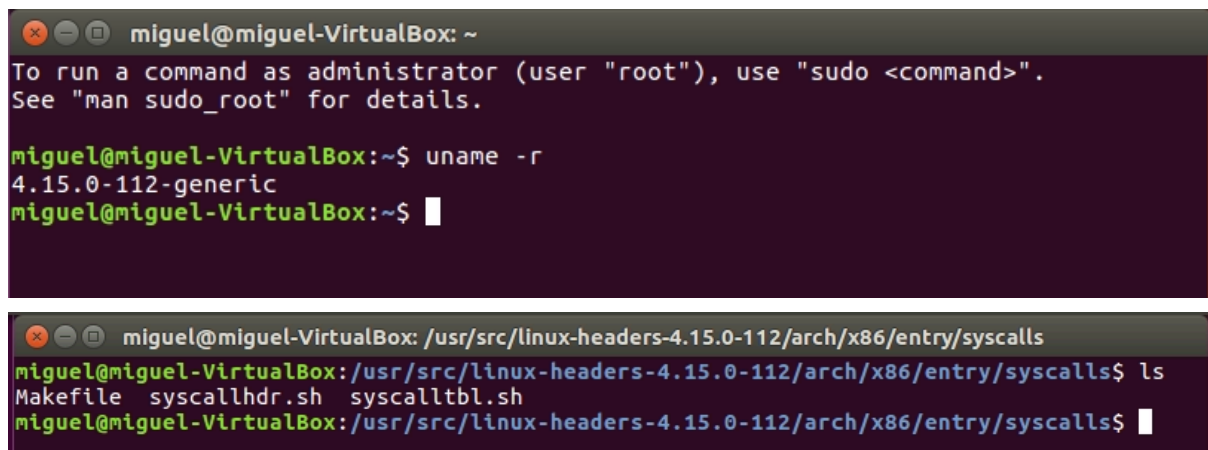
La preparación del entorno ha sido crucial para la investigación técnica y la implementación del proyecto. Esto ha incluido la:

- **Configuración de una máquina virtual** con Ubuntu 16.04 LTS para garantizar un entorno estable y compatible.
- **Instalación de herramientas necesarias** para compilar y trabajar con el kernel, (como `gcc`, `make` y `libncurses5-dev`)
- **Revisión** de documentación técnica y **uso** de recursos avanzados como Perplexity para obtener información precisa.

5.2. Proceso de implementación de la llamada al sistema

Primero de todo se ha de instalar la máquina virtual con **Ubuntu 16.04 LTS**; una vez hecho es necesario utilizar los siguientes comandos para verificar la versión del kernel y acceder como usuario root:

- `uname -r`
- `sudo -s`



```
miguel@miguel-VirtualBox: ~
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

miguel@miguel-VirtualBox:~$ uname -r
4.15.0-112-generic
miguel@miguel-VirtualBox:~$ 

miguel@miguel-VirtualBox: /usr/src/linux-headers-4.15.0-112/arch/x86/entry/syscalls
miguel@miguel-VirtualBox:/usr/src/linux-headers-4.15.0-112/arch/x86/entry/syscalls$ ls
Makefile syscallhdr.sh syscalltbl.sh
miguel@miguel-VirtualBox:/usr/src/linux-headers-4.15.0-112/arch/x86/entry/syscalls$
```

Para poder seguir compilando, se han de instalar las siguientes librerías:

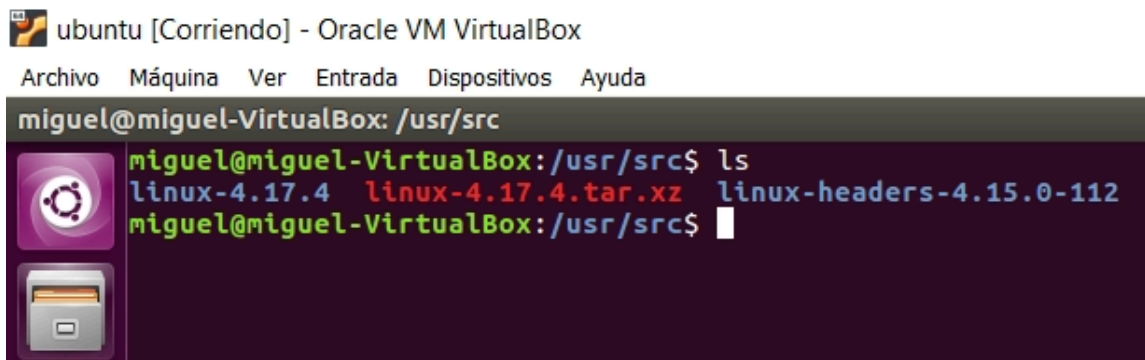
- `sudo apt-get install gcc`
- `sudo apt-get install libncurses5-dev`
- `sudo apt-get install bison`
- `sudo apt-get install flex`
- `sudo apt-get install libssl-dev`
- `sudo apt-get install libelf-dev`
- `sudo apt-get install make`
- `sudo apt-get update`
- `sudo apt-get upgrade`

Con todo esto, hay que ir al directorio **src** con el comando `'cd /usr/src'` y descargar la versión 4.17.4 del kernel:

- `wget`
`https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.17.4.tar.xz`

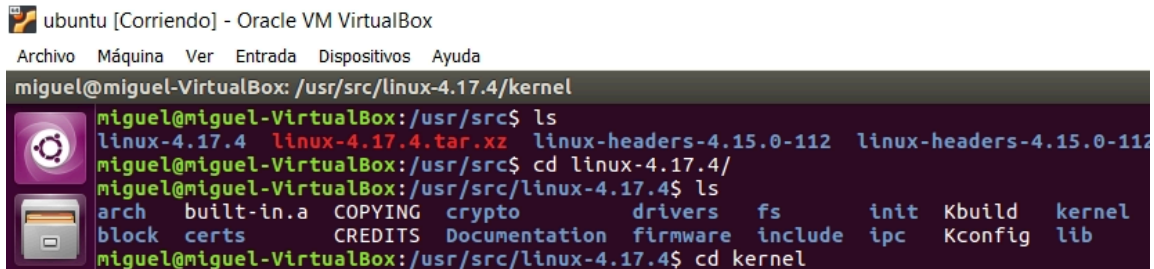
Para poder usarlo, es necesario descomprimirlo con el comando **tar**:

- `tar -xvf linux-4.17.4.tar.xz`



5.3. Modificación de archivos necesarios

A partir de aquí, se puede crear una carpeta a parte o crear directamente la función en la carpeta 'kernel':



```
ubuntu [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/kernel
miguel@miguel-VirtualBox: /usr/src$ ls
linux-4.17.4  linux-4.17.4.tar.xz  linux-headers-4.15.0-112  linux-headers-4.15.0-112
miguel@miguel-VirtualBox: /usr/src$ cd linux-4.17.4/
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4$ ls
arch  built-in.a  COPYING  crypto  drivers  fs  init  Kbuild  kernel
block certs  CREDITS  Documentation  firmware  include  ipc  Kconfig  lib
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4$ cd kernel
```

Se definirá la siguiente función ('suma.c') con el editor de texto **nano**:

- nano suma.c

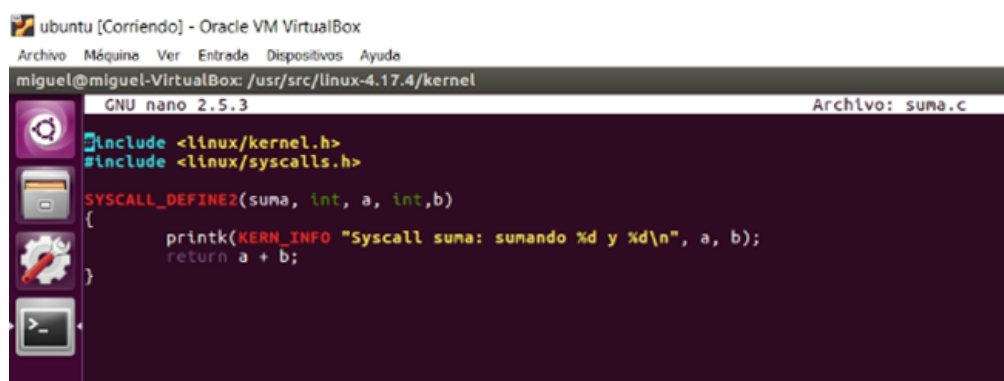
```
#include <linux/kernel.h>
#include <linux/syscalls.h>

SYSCALL_DEFINE2(suma, int, a, int, b) {

    printf(KERN_INFO "Syscall suma: sumando %d y %d\n", a, b);

    return a + b;

}
```



```
ubuntu [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/kernel
GNU nano 2.5.3 Archivo: suma.c
#include <linux/kernel.h>
#include <linux/syscalls.h>

SYSCALL_DEFINE2(suma, int, a, int, b)
{
    printk(KERN_INFO "Syscall suma: sumando %d y %d\n", a, b);
    return a + b;
}
```

Después, habrá que modificar el makefile de la carpeta kernel o crear uno propio con el siguiente código:

```
obj-y += suma.o
```

```
ubuntu [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/kernel
GNU nano 2.5.3 Archivo: Makefile

# These are called from save_stack_trace() on slub debug path,
# and produce insane amounts of uninteresting coverage.
KCOV_INSTRUMENT_module.o := n
KCOV_INSTRUMENT_extable.o := n
# Don't self-instrument.
KCOV_INSTRUMENT_kcov.o := n
KASAN_SANITIZE_kcov.o := n

# cond_syscall is currently not LTO compatible
CFLAGS_sys_ni.o = $(DISABLE_LTO)

obj-y += sched/
obj-y += locking/
obj-y += power/
obj-y += printk/
obj-y += irq/
obj-y += rcu/
obj-y += livepatch/

obj-$(CONFIG_CHECKPOINT_RESTORE) += kcmp.o
obj-$(CONFIG_FREEZER) += freezer.o
obj-$(CONFIG_PROFILING) += profile.o
obj-$(CONFIG_STACKTRACE) += stacktrace.o
obj-y += time/
obj-$(CONFIG_FUTEX) += futex.o
ifeq ($(CONFIG_COMPAT),y)
obj-$(CONFIG_FUTEX) += futex_compat.o
endif
obj-$(CONFIG_GENERIC_ISA_DMA) += dma.o
obj-$(CONFIG_SMP) += smp.o
ifneq ($(CONFIG_SMP),y)
obj-y += up.o
endif
obj-y += suma.o

obj-$(CONFIG_UID16) += uid16.o
obj-$(CONFIG_MODULES) += module.o
obj-$(CONFIG_MODULE_SIG) += module_signing.o
obj-$(CONFIG_KALLSYMS) += kallsyms.o
obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
obj-$(CONFIG_CRASH_CORE) += crash_core.o
obj-$(CONFIG_KEXEC_CORE) += kexec_core.o
obj-$(CONFIG_KEXEC) += kexec.o
obj-$(CONFIG_KEXEC_FILE) += kexec_file.o
obj-$(CONFIG_BACKTRACE_SELF_TEST) += backtracetest.o
obj-$(CONFIG_COMPAT) += compat.o
obj-$(CONFIG_CGROUPS) += cgroup/
obj-$(CONFIG_UTS_NS) += utsname.o
obj-$(CONFIG_USER_NS) += user_namespace.o

^G Ver ayuda ^O Guardar ^M Buscar ^K Cortar Texto ^J Justificar ^C Posición
^X Salir ^R Leer fich. ^N Reemplazar ^U Pegar txt ^T Ortografía ^_ Ir a línea
```

Seguidamente se ejecutará el comando 'cd arch/x86/entry/syscalls/' para acceder a la carpeta 'syscalls'. Para registrar la nueva llamada al sistema habrá que editar el archivo 'syscall_64.tbl' con el comando 'nano syscall_64.tbl'.

```
ubuntu [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/arch/x86/entry/syscalls
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4$ cd arch/x86/entry/syscalls/
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/arch/x86/entry/syscalls$ ls
Makefile syscall_32.tbl syscall_64.tbl syscallhdr.sh syscalltbl.sh
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/arch/x86/entry/syscalls$
```

5.4. Registro de la nueva syscall

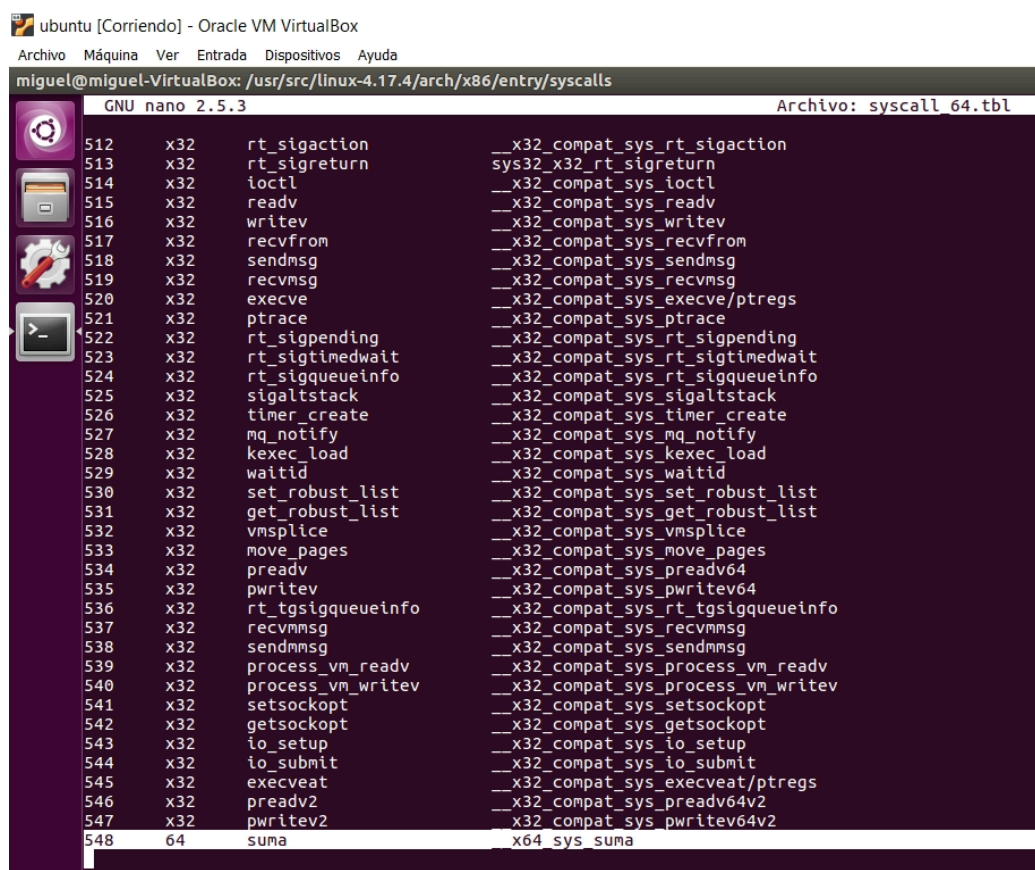
El registro de una nueva syscall en el kernel de Linux implica modificar varios archivos clave. Estos pasos aseguran que la llamada al sistema sea reconocida e integrada correctamente en el mismo.

El archivo **syscall_64.tbl**, ubicado en la carpeta `arch/x86/entry/syscalls/`, se utiliza para asignar un número a cada syscall en el kernel. Este número sirve como identificador de la syscall durante su invocación desde el espacio de usuario.

Para registrar la nueva syscall, hay que editar el archivo y añadir el siguiente código al final:

```
548 64 suma __x64_sys_suma
```

Es importante asegurarse de que el número 548 no esté asignado previamente en el archivo, ya que cada llamada al sistema debe tener un número único.



```
ubuntu [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/arch/x86/entry/syscalls
GNU nano 2.5.3 Archivo: syscall_64.tbl
512 x32 rt_sigaction __x32_compat_sys_rt_sigaction
513 x32 rt_sigreturn sys32_x32_rt_sigreturn
514 x32 ioctl __x32_compat_sys_ioctl
515 x32 readv __x32_compat_sys_readv
516 x32 writev __x32_compat_sys_writev
517 x32 recvfrom __x32_compat_sys_recvfrom
518 x32 sendmsg __x32_compat_sys_sendmsg
519 x32 recvmmsg __x32_compat_sys_recvmmsg
520 x32 execve __x32_compat_sys_execve/ptregs
521 x32 ptrace __x32_compat_sys_ptrace
522 x32 rt_sigpending __x32_compat_sys_rt_sigpending
523 x32 rt_sigtimedwait __x32_compat_sys_rt_sigtimedwait
524 x32 rt_sigqueueinfo __x32_compat_sys_rt_sigqueueinfo
525 x32 sigaltstack __x32_compat_sys_sigaltstack
526 x32 timer_create __x32_compat_sys_timer_create
527 x32 mq_notify __x32_compat_sys_mq_notify
528 x32 kexec_load __x32_compat_sys_kexec_load
529 x32 waitid __x32_compat_sys_waitid
530 x32 set_robust_list __x32_compat_sys_set_robust_list
531 x32 get_robust_list __x32_compat_sys_get_robust_list
532 x32 vmsplice __x32_compat_sys_vmsplice
533 x32 move_pages __x32_compat_sys_move_pages
534 x32 preadv __x32_compat_sys_preadv64
535 x32 pwritev __x32_compat_sys_pwritev64
536 x32 rt_tgsigqueueinfo __x32_compat_sys_rt_tgsigqueueinfo
537 x32 recvmmsg __x32_compat_sys_recvmmsg
538 x32 sendmmsg __x32_compat_sys_sendmmsg
539 x32 process_vm_readv __x32_compat_sys_process_vm_readv
540 x32 process_vm_writev __x32_compat_sys_process_vm_writev
541 x32 setsockopt __x32_compat_sys_setsockopt
542 x32 getsockopt __x32_compat_sys_getsockopt
543 x32 io_setup __x32_compat_sys_io_setup
544 x32 io_submit __x32_compat_sys_io_submit
545 x32 execveat __x32_compat_sys_execveat/ptregs
546 x32 preadv2 __x32_compat_sys_preadv64v2
547 x32 pwritev2 __x32_compat_sys_pwritev64v2
548 64 suma __x64_sys_suma
```

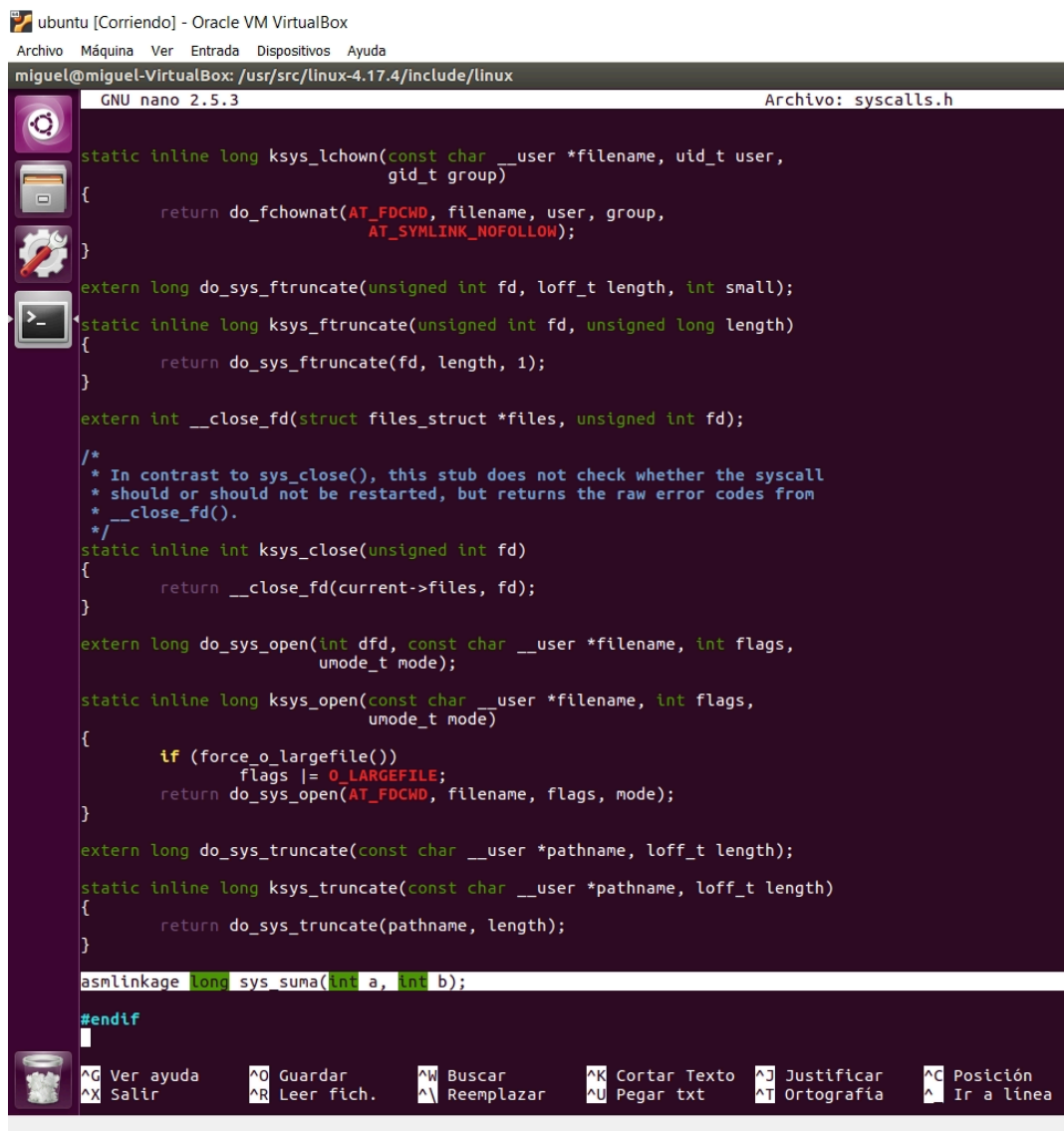
El archivo `syscalls.h`, ubicado en la carpeta `include/linux/`, es otro elemento esencial para registrar la nueva syscall. Este archivo define los prototipos de las funciones de las llamadas, asegurando que puedan ser utilizadas tanto en el kernel como desde el espacio de usuario.

Para declarar la nueva llamada al sistema, habrá que editar el fichero 'syscalls.h' en la carpeta 'include/linux':

- `cd /usr/src/linux-4.17.4/include/linux/`
- `nano syscalls.h`

Se incluirá el siguiente código al final del archivo, antes del último `#endif`:

```
asmlinkage long sys_suma(int a, int b);
```



```
ubuntu [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/include/linux
GNU nano 2.5.3 Archivo: syscalls.h

static inline long ksys_lchown(const char __user *filename, uid_t user,
                              gid_t group)
{
    return do_fchownat(AT_FDCWD, filename, user, group,
                      AT_SYMLINK_NOFOLLOW);
}

extern long do_sys_ftruncate(unsigned int fd, loff_t length, int small);
static inline long ksys_ftruncate(unsigned int fd, unsigned long length)
{
    return do_sys_ftruncate(fd, length, 1);
}

extern int __close_fd(struct files_struct *files, unsigned int fd);
/*
 * In contrast to sys_close(), this stub does not check whether the syscall
 * should or should not be restarted, but returns the raw error codes from
 * __close_fd().
 */
static inline int ksys_close(unsigned int fd)
{
    return __close_fd(current->files, fd);
}

extern long do_sys_open(int dfd, const char __user *filename, int flags,
                       umode_t mode);
static inline long ksys_open(const char __user *filename, int flags,
                             umode_t mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;
    return do_sys_open(AT_FDCWD, filename, flags, mode);
}

extern long do_sys_truncate(const char __user *pathname, loff_t length);
static inline long ksys_truncate(const char __user *pathname, loff_t length)
{
    return do_sys_truncate(pathname, length);
}

asmlinkage long sys_suma(int a, int b);

#endif
```


5.5. Interacción entre el espacio de usuario y el espacio del kernel

Hay que crear un código en C ('prueba.c') para invocar la llamada al sistema:

```
#include <stdio.h>

#include <unistd.h>

#include <sys/syscall.h>

#define SYS_my_syscall 548

int main() {

    int resultado = syscall(SYS_my_syscall, 5, 3);

    printf("El resultado de my_syscall es: %d\n", resultado);

    return 0;

}
```



6. Pruebas y validaciones

6.1. Compilación del kernel con la nueva syscall

Ahora es momento de compilar y probar el kernel con la nueva llamada al sistema implementada. Primero de todo habrá que navegar hasta el directorio raíz del kernel:

- `cd /usr/src/linux-4.17.4`

Antes de compilar, se va a utilizar la herramienta interactiva 'menuconfig' para personalizar las opciones del kernel:

- `make menuconfig`

Esta herramienta permite ajustar el kernel para las necesidades del proyecto, eliminando módulos innecesarios o configurándolo para hardware específico. Hay que guardar la configuración al finalizar y después salir (con 'exit'). Esto creará un archivo `.config` que será utilizado en la compilación. Una vez configurado, se inicia la compilación usando múltiples núcleos del procesador para acelerar el proceso:

- `make -j $(nproc)`

Es necesario esperar a que termine de compilar todo el código donde, tras compilar, se instalarán los módulos y el kernel en el sistema:

- `make modules_install install`

Posteriormente, hay que comprobar que existe la carpeta 'vmlinuz-4.17.4' con el comando '`cd /boot/`', y después se tendrá que reiniciar el sistema operativo:

- `shutdown -r now`

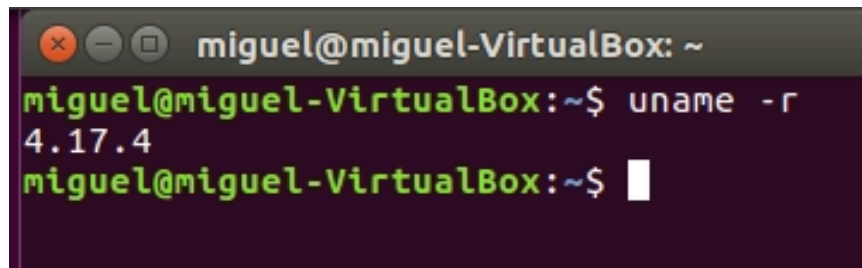
Finalmente, para compilar se usará el comando '`gcc`':

- `gcc prueba.c`

6.2. Pruebas funcionales de la nueva syscall

Una vez iniciado de nuevo el sistema operativo hay que ver si ha cambiado el kernel con:

- `uname -r`

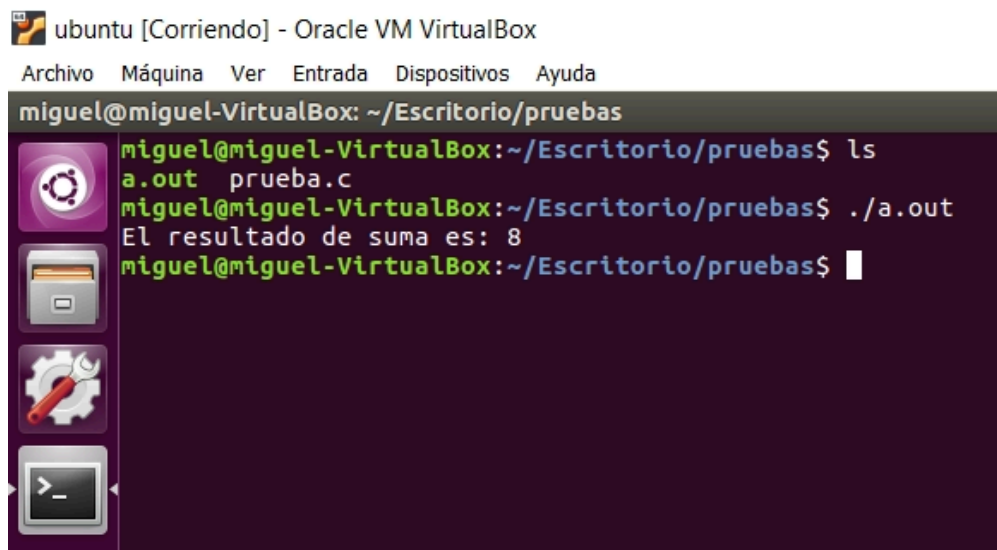


```
miguel@miguel-VirtualBox: ~  
miguel@miguel-VirtualBox:~$ uname -r  
4.17.4  
miguel@miguel-VirtualBox:~$
```

Por último se ejecutará el archivo 'prueba.c' con el comando:

- `./a.out`

Si sale el resultado bien, todo funciona correctamente.



```
ubuntu [Corriendo] - Oracle VM VirtualBox  
Archivo Máquina Ver Entrada Dispositivos Ayuda  
miguel@miguel-VirtualBox: ~/Escritorio/pruebas  
miguel@miguel-VirtualBox:~/Escritorio/pruebas$ ls  
a.out prueba.c  
miguel@miguel-VirtualBox:~/Escritorio/pruebas$ ./a.out  
El resultado de suma es: 8  
miguel@miguel-VirtualBox:~/Escritorio/pruebas$
```

7. Ejemplos prácticos

7.1. Análisis de casos de uso comunes

Las llamadas al sistema (*syscalls*) son fundamentales en los sistemas operativos, ya que permiten a los programas interactuar con el núcleo para realizar tareas esenciales. A continuación, se analizan los casos de uso más comunes y su importancia:

1. **Gestión de procesos:** este caso de uso se refiere a la creación, ejecución, suspensión y finalización de procesos. Es esencial para la multitarea y la gestión eficiente de recursos del sistema. Aquí podemos hablar de llamadas al sistema mencionadas anteriormente en el apartado 4 como por ejemplo `"fork()"` o `"exec()"`.
2. **Gestión de archivos:** nos referimos a la apertura, lectura, escritura o cierre de archivos. Podemos poner como ejemplo un editor de texto que es el que hace uso de estas llamadas al sistema como `"open"`, `"write"` o `"close"`.
3. **Gestión de dispositivos:** nos permite interactuar con hardware como discos duros, impresoras o dispositivos USB (engloba también los dispositivos E/S).
 - Ejemplos de syscalls: `"ioctl"`, `"read"`, `"write"`.
 - Uso práctico: un controlador usa `"ioctl"` para configurar parámetros específicos del dispositivo. O por ejemplo, las aplicaciones leen datos desde un disco o escriben en él mediante las llamadas correspondientes como `"read"` o `"write"`.
4. **Gestión de memoria:** incluye asignación dinámica y mapeo de memoria.
 - Ejemplos de syscalls: `"mmap"`, `"munmap"`, `"brk"`.
 - Uso práctico: las aplicaciones como bases de datos usan `"mmap"` para mapear archivos en memoria, acelerando el acceso a datos. Optimiza el uso eficiente de recursos de memoria, mejorando el rendimiento general del sistema.

Los casos de uso más comunes de llamadas al sistema abarcan aspectos críticos como gestión de procesos, archivos, dispositivos e información del sistema. Estas llamadas al sistema son esenciales para el correcto funcionamiento del sistema operativo y proporcionar una interfaz segura entre las aplicaciones y el hardware del PC.

7.2. Comparación con otras formas de interacción con el sistema operativo

Existen diferentes formas de interacción con el sistema operativo, donde podemos hacer una comparación entre las llamadas al sistema y las principales formas de interacción.

En primer lugar, comparemos las llamadas al sistema con **bibliotecas de alto nivel (APIs)**:

Ventajas sobre las llamadas al sistema	Desventajas sobre las llamadas al sistema
<ul style="list-style-type: none">★ Mayor facilidad de uso y abstracción.★ Portabilidad entre diferentes sistemas operativos.★ Manejo automático de errores y recursos.	<ul style="list-style-type: none">★ Menor control sobre los recursos del sistema.★ Posible sobrecarga de rendimiento.

Ejemplo práctico: La función “`fopen()`” de C abstrae la llamada al sistema “`open()`” simplificando el manejo de archivos.

También podemos comparar la **comunicación directa con dispositivos de hardware**:

Ventajas sobre las llamadas al sistema	Desventajas sobre las llamadas al sistema
<ul style="list-style-type: none">★ Control preciso sobre el hardware.★ Potencial para mayor rendimiento en sistemas especializados.	<ul style="list-style-type: none">★ Requiere privilegios altos y conocimientos específicos del hardware.★ Riesgo de inestabilidad del sistema si se usa incorrectamente.★ Falta de portabilidad.

Finalmente podríamos comparar con los **servicios en modo usuario**, ya que las llamadas al sistema son en modo kernel.

Ventajas sobre las llamadas al sistema	Desventajas sobre las llamadas al sistema
<ul style="list-style-type: none">★ Mayor modularidad y estabilidad del sistema.★ Facilita la actualización y mantenimiento de componentes individuales.	<ul style="list-style-type: none">★ Posible sobrecarga por cambios de contexto adicionales.★ Sigue dependiendo de las llamadas al sistema para operaciones fundamentales.

Como conclusión podemos decir que cada método de interacción tiene sus ventajas y desventajas, sin embargo, las llamadas al sistema siguen siendo fundamentales para el funcionamiento de bajo nivel y el acceso directo a los recursos del sistema y por lo tanto para el correcto funcionamiento del sistema operativo, mientras que las otras formas de interacción nos proporcionan abstracciones útiles para diferentes niveles de desarrollo y uso del sistema operativo.

8. Optimización y mejores prácticas

8.1. Técnicas para optimizar el uso de llamadas al sistema

Es posible acelerar la compilación del kernel de Linux para reducir el tiempo que tarda, por eso existen diversas técnicas para optimizar este proceso:

1. **Compilación en paralelo.** Como se ha explicado anteriormente, se pueden usar múltiples núcleos del procesador con el flag `-j` en el comando `make`, como en:

- `make -j $(nproc)`

siendo `$(nproc)` el número de núcleos disponibles.

Esta instrucción distribuye los trabajos entre los núcleos, acelerando el proceso. (Solo con Linux, 2023)

2. **Uso de la configuración actual del kernel.** Basarse en la configuración del kernel en uso (`.config`) evita tener que configurar todo desde el principio.
3. **Optimización para hardware local.** Es recomendable generar una configuración específica para el hardware de la máquina, con el fin de optimizar la configuración para su hardware específico (Solo con Linux, 2023) con:

- `make localmodconfig`

4. **Eliminación de símbolos de depuración.** Es posible usar el parámetro `INSTALL_MOD_STRIP=1` en el comando `make` para eliminar símbolos de depuración, cosa que reduce el tamaño del paquete generado y acelera la compilación (Evaristo GZ, 2016):

- `make -j $(nproc) INSTALL_MOD_STRIP=1 deb-pkg`

5. **Reducción de módulos innecesarios.** Mediante herramientas como `'menuconfig'` o `'xconfig'`, se elimina soporte para hardware o funcionalidades no requeridas.
6. **Uso de máquinas más potentes o kernels precompilados.** Si los tiempos fueran críticos, se podría optar por compilar en un equipo más potente o utilizar kernels precompilados optimizados (como Xanmod o Liquorix)

Todas estas optimizaciones pueden reducir significativamente el tiempo de compilación, pasando de decenas de minutos a solo unos pocos en un hardware moderno y potente.

8.2. Mejores prácticas para la programación de llamadas al sistema en Linux

Para la programación de llamadas al sistema en Linux debemos tener en cuenta cuáles son las mejores prácticas para su correcto funcionamiento.

- **Manejo de errores:** verificación del **valor de retorno** de las llamadas al sistema para poder detectar errores; podemos usar la variable `"errno"` y la función `"perror()"` para obtener información detallada sobre los errores.
- **Verificación de parámetros:**
 - Validar entradas antes de las llamadas.
 - Comprobar rangos y tipos de datos.
- **Control de recursos:**
 - Liberar memoria asignada.
 - Cerrar descriptores de archivos.
 - Finalizar los procesos correctamente.
- **Seguridad:**
 - Usar permisos mínimos necesarios.
 - Validar entradas de usuario.
- **Rendimiento:**
 - Minimizar llamadas al sistema.
 - Usar syscalls de manera eficiente.
 - Aprovechar el caché del kernel.
- **Portabilidad:**
 - Verificar compatibilidad entre sistemas.
 - Documentar dependencias específicas.

8.3. Consideraciones de seguridad y rendimiento

Al implementar llamadas al sistema en Linux, es crucial considerar tanto la seguridad como el rendimiento.

8.3.1. Seguridad

En cuanto a la seguridad podemos decir que es importante implementar medidas que garanticen la seguridad de la llamada al sistema para poder proteger al kernel de vulnerabilidades conocidas y evitar el acceso no autorizado.

Otra medida importante es mantener el **kernel** y el software actualizados con los últimos parches de seguridad. También podemos utilizar herramientas como SELinux para restringir los permisos y acciones de los procesos.

8.3.2. Rendimiento

Las llamadas al sistema afectan directamente a la eficiencia y velocidad de ejecución de los programas. Cada llamada al sistema implica un cambio de contexto entre el espacio y el kernel, lo que conlleva una sobrecarga de rendimiento. Es importante:

- Minimizar el número de llamadas al sistema para mejorar el rendimiento, ya que cada llamada implica un cambio de contexto entre el espacio de usuario y el kernel.
- Se pueden utilizar herramientas para monitorear el rendimiento del sistema e identificar procesos que consumen muchos recursos (lo cual no es deseable).
- Es importante reducir el número de llamadas al sistema de mayor coste de tiempo de CPU, como `fork()` y `exec()`.
- El ajuste de parámetros como `max-file` (número máximo de archivos abiertos por el sistema) y `ulimit -n` (límite de archivos abiertos por proceso) puede impactar significativamente en el rendimiento del sistema, así que es algo que hay que tener en cuenta.

9. Herramientas y recursos

9.1. Herramientas de depuración y análisis de llamadas al sistema

La **depuración** es el proceso de identificar, analizar y corregir errores o defectos en el código fuente de un programa con el objetivo de mejorar la calidad y la confiabilidad del software. No ayuda a identificar problemas y posibles interacciones entre programa y el kernel pero nos permite analizar el comportamiento del programa.

Nos puede ayudar a entender mejor qué realiza cada paso del programa, aunque no ayuda a detectar elementos necesarios lo que ayuda a optimizar el rendimiento.

Ayuda a detectar vulnerabilidades que no hemos detectado previamente, y puede ser útil para resolver problemas de interoperabilidad (es decir problemas de incompatibilidad o problemas de comunicación)

En los siguientes apartados vamos a explicar varias herramientas para depurar:

9.1.1. strace

Strace es una herramienta de línea de comandos en sistemas Linux utilizada para monitorizar y depurar las interacciones entre un programa kernel del sistema operativo. Intercepta y registra las llamadas al sistema realizadas por un proceso, lo que permite examinar la capa límite entre el espacio de usuario y el espacio kernel.

Algunas funciones principales de strace son la depuración, el análisis de comportamiento y la resolución de problemas. Su sintaxis es:

```
strace [opciones] programa [argumentos]
```

Un ejemplo de strace con el comando `ls`:

```
strace ls
```

Algunas de las opciones más esenciales son:

- `-f`: Rastrea procesos hijos creados por `fork()`.
- `-t`: Agrega marcas de tiempo a cada línea de salida.
- `-e trace=call`: Filtra para mostrar sólo ciertas llamadas al sistema.
- `-o archivo`: Guarda la salida en un archivo en lugar de mostrarla en la terminal.

Ejemplo de uso avanzado:

```
strace -fvttTyy -s 256 -e trace=open,close,read,write programa
```

Este comando rastreará las llamadas `open`, `close`, `read` y `write` del programa especificado, mostrando información detallada y marcas de tiempo.

9.1.2. ltrace

Ltrace es una herramienta para monitorear y depurar las llamadas a funciones de biblioteca realizadas por programas en sistemas Linux. Intercepta y registra las llamadas a funciones de biblioteca dinámica realizadas por un programa, lo que permite examinar cómo un programa interactúa con las bibliotecas del sistema y terceros.

Algunos de sus usos principales son: depuración, análisis de comportamiento y optimización.

En cuanto a la sintaxis, en `ltrace` sería:

```
ltrace [opciones] programa [argumentos]
```

Un ejemplo:

```
ltrace ls
```

Algunas de las opciones más importantes:

- `-f`: Rastrea procesos hijos creados por `fork()`.
- `-t`: Agrega marcas de tiempo a cada línea de salida.
- `-e`: Filtra para mostrar sólo ciertas llamadas a funciones.
- `-o archivo`: Guarda la salida en un archivo en lugar de mostrarla en la terminal.

Ejemplo de uso avanzado:

```
ltrace -f -l libc -e malloc,free programa
```

Este comando rastreará las llamadas a `malloc` y `free` de la biblioteca `libc` en el programa especificado, incluyendo procesos hijos.

9.2. Recursos adicionales para aprender más sobre llamadas al sistema en Linux

Se recomienda el uso de agentes basados en inteligencia artificial para facilitar el aprendizaje y la resolución de problemas técnicos, como **Mistral**. Este es un agente creado para ofrecer soporte interactivo y responder preguntas técnicas relacionadas con Linux y llamadas al sistema, que puede ser consultado en [este enlace](#).

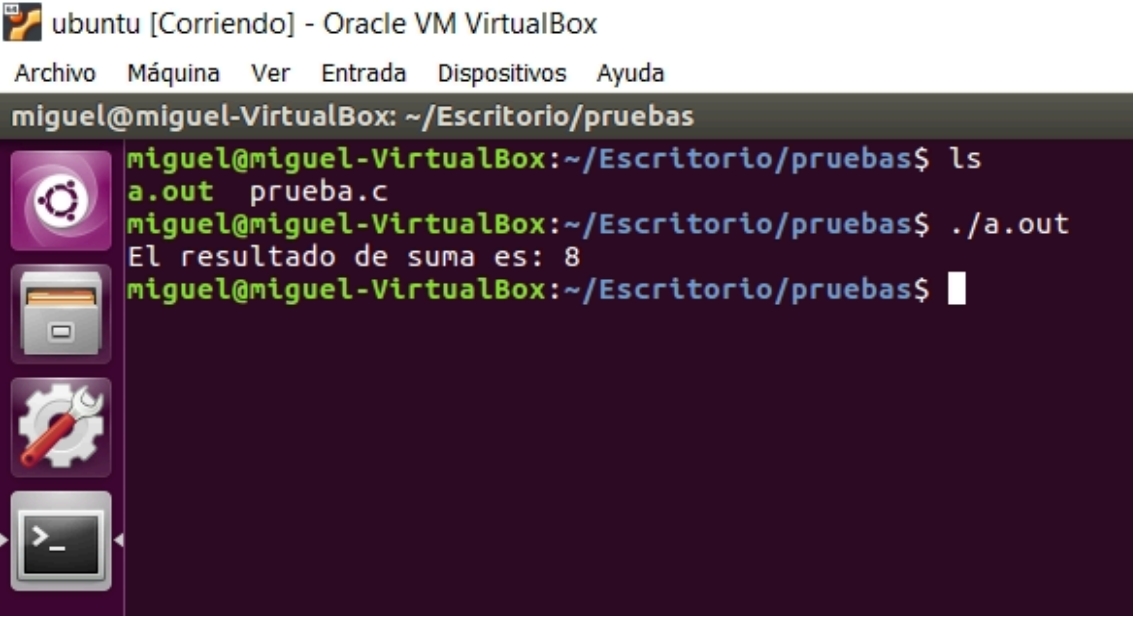
Esta herramienta ha sido útil para obtener respuestas rápidas y adaptadas a dudas específicas del proyecto.

10. Publicación y presentación de resultados

El propósito y funcionalidad de esta llamada al sistema era implementar una *syscall* suma, lo que realizaría la suma de dos parámetros únicamente.

Más adelante se pueden ver los códigos que hemos utilizado para la implementación.

En la siguiente imagen que también hemos mostrado anterior podemos observar el resultado de la llamada al sistema:



The image shows a terminal window titled "ubuntu [Corriendo] - Oracle VM VirtualBox". The window has a menu bar with "Archivo", "Máquina", "Ver", "Entrada", "Dispositivos", and "Ayuda". The terminal prompt is "miguel@miguel-VirtualBox: ~/Escritorio/pruebas". The user enters the command "ls", and the output is "a.out prueba.c". Then, the user enters the command "./a.out", and the output is "El resultado de suma es: 8". The terminal window has a sidebar on the left with icons for a terminal, a folder, a gear, and a window.

```
miguel@miguel-VirtualBox: ~/Escritorio/pruebas
miguel@miguel-VirtualBox:~/Escritorio/pruebas$ ls
a.out  prueba.c
miguel@miguel-VirtualBox:~/Escritorio/pruebas$ ./a.out
El resultado de suma es: 8
miguel@miguel-VirtualBox:~/Escritorio/pruebas$
```

Este sería el código en lenguaje C:



The screenshot shows a terminal window titled 'ubuntu [Corriendo] - Oracle VM VirtualBox'. The menu bar includes 'Archivo', 'Máquina', 'Ver', 'Entrada', 'Dispositivos', and 'Ayuda'. The prompt is 'miguel@miguel-VirtualBox: ~/Escritorio/pruebas'. The nano editor is open to 'prueba.c' with version 'GNU nano 2.5.3'. The code is as follows:

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

#define SYS_my_syscall 548

int main() {
    int resultado = syscall(SYS_my_syscall, 5, 3);
    printf("El resultado de suma es: %d\n", resultado);
    return 0;
}
```

Y esta sería la implementación en el kernel de la llamada al sistema:



The screenshot shows a terminal window titled 'ubuntu [Corriendo] - Oracle VM VirtualBox'. The menu bar is the same as the previous image. The prompt is 'miguel@miguel-VirtualBox: /usr/src/linux-4.17.4/kernel'. The nano editor is open to 'suma.c' with version 'GNU nano 2.5.3'. The code is as follows:

```
#include <linux/kernel.h>
#include <linux/syscalls.h>

SYSCALL_DEFINE2(suma, int, a, int, b)
{
    printk(KERN_INFO "Syscall suma: sumando %d y %d\n", a, b);
    return a + b;
}
```

Al ser una llamada al sistema sencilla como es una suma, no hay gran probabilidad de error, en la acción de llamada, los puntos críticos son la creación y la compilación del kernel.

Algunos desafíos que nos hemos encontrado han sido entender la información sobre llamadas al sistema, ya que hay muchos conceptos que hay que saber muy bien para poder seguir profundizando sobre cómo se estructura el kernel de Linux, etc.

11. Conclusiones

Como conclusiones podemos decir que ha sido interesante la búsqueda de información para la creación de una llamada al sistema, ya que hemos tenido que buscar información desde lo más básico como que es una llamada al sistema a más específico como que hace ciertos archivos o para que funcionan.

La parte de búsqueda ha sido facilitada a la gran cantidad de información que nos ha proporcionado diferentes IAs como Chat GPT o principalmente Perplexity, ya que hemos podido hacer preguntas sobre llamadas al sistema y nos proporcionaba información de forma clara, además de proporcionarnos las fuentes donde pudimos comprobar que la información era correcta.

11.1. Importancia de las llamadas al sistema en el desarrollo de software

La utilización de llamadas al sistema nos puede ayudar a mejorar la comunicación entre aplicaciones, añadir una capa de abstracción para una mejor seguridad y estabilidad del sistema, gestionar mejor los recursos, etc.

Nos puede ayudar a diferenciar que tipo de llamadas al sistema, ya sea de control de procesos, gestión de archivos, gestión de dispositivos, gestión de la información, comunicación entre procesos.

Si usamos adecuadamente las llamadas al sistema podemos mejorar el rendimiento y la productividad de nuestro software, por ejemplo optimizando el uso de recursos, simplificar el número de pasos que necesita hacer una tarea, y añadir una capa de seguridad ya que las llamadas al sistema únicamente se pueden realizar en modo núcleo.

11.2. Futuras investigaciones y desarrollos

Para mejorar el desarrollo de este trabajo, deberíamos quizás introducir la implementación de una llamada al sistema un poco más compleja ya que la operación de suma, para una llamada al sistema es sencilla, pero si queremos mejorarla podríamos cambiar el tipo de dato de int a otro tipo para intentar optimizar recursos, añadir una capa de seguridad para asegurarnos de que si hay cierto número que no se realice la suma.

Otro punto a investigar y desarrollar sería intentar implementar la misma llamada al sistema en diferentes versiones del kernel de linux, obviamente teniendo en cuenta las diferencias que hay entre kernel.

Una mejora que podríamos introducir a la syscall de suma, es averiguar si se puede crear una syscall con varios parámetros y que estos se sumen, tendríamos que tener en cuenta diferentes factores, como si es una buena idea o no, si afectaría al rendimiento, si es seguro tener muchos parámetros posibles en una syscall, etc.

12. Referencias

- [1] Xataka. (14/04/2015). *Llega Linux 4.0, estas son todas sus novedades*. Xataka México. Consultado el 16/01/2025, de <https://www.xataka.com.mx/telecomunicaciones/llega-linux-4-0-estas-son-todas-sus-novedades>
- [2] Isophiagr. (21/03/2021). *Llamada al sistema*. os-system-call. Consultado el 03/01/2025, de <https://github.com/Isophiagr/os-system-call>
- [3] Solo con Linux. (07/11/2023). *Compilar kernel en Linux*. Cómo compilar un Kernel. Consultado el 03/01/2025, de <https://soloconlinux.org.es/compilar-kernel/>
- [4] Evaristo GZ. (30/09/2016). *Personalización, reducción y compilación de un kernel Linux a medida*. Compilación kernel Linux a medida. Consultado el 03/01/2025, de <https://www.evaristogz.com/compilacion-kernel-linux-a-medida/>
- [5] Wikipedia. (s.f.). *Linux kernel version history*. Wikipedia. Consultado el 16/01/2025, de https://en.wikipedia.org/wiki/Linux_kernel_version_history
- [6] Villela. (s.f.). *Estructura del Kernel de Linux*. Consultado el 18/01/2025, de https://issuu.com/l23069/docs/actividad_6._manri._villela/s/30958670#:~:text=La%20estructura%20del%20kernel%20de,de%20red%20y%20los%20drivers%2C