Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computadores
Principios de sistemas operativos (CE-4303)

Tarea 1:

Servicio de Cliente/Servidor en Linux
Atributos

Profesor:
Jason Leiton Jimenez

Estudiantes:
Adriel Sebastián Chaves Salazar
Daniel Cob Beirute

II Semestre 2025

Tabla de Contenidos

1. Desarrollo de las preguntas de teoría:

1.1. Pregunta#1 : ¿Qué es una llamada al sistema?

Respuesta: Una llamada al sistema es una interfaz programática entre las aplicaciones de usuario y el núcleo del sistema operativo que permite a los programas solicitar servicios privilegiados del SO de manera controlada y segura [1]. Cuando un programa en modo usuario necesita realizar operaciones que requieren privilegios del kernel (como acceso a hardware, gestión de memoria o E/S), ejecuta una instrucción especial de trampa (trap) o syscall que transfiere el control al sistema operativo [2]. El kernel procesa la solicitud, verifica los permisos, ejecuta la operación solicitada y devuelve el control al programa usuario junto con el resultado de la operación. Este mecanismo garantiza que ningún proceso en modo usuario pueda acceder directamente al hardware, manteniendo así la seguridad y estabilidad del sistema [3].

1.2. Pregunta#2 : ¿Hay alguna diferencia entre una llamada al sistema y una interrupción?

Respuesta: Sí, existen diferencias fundamentales entre llamadas al sistema e interrupciones, aunque ambas utilizan mecanismos similares de cambio de contexto [4]. Las llamadas al sistema son interrupciones de software **síncronas** iniciadas voluntariamente por el programa mediante instrucciones específicas (como `int 0x80` en x86 o `syscall` en x86-64), mientras que las interrupciones de hardware son eventos **asíncronos** generados por dispositivos externos o condiciones del sistema [5]. Las llamadas al sistema siempre ocurren en puntos predecibles del código cuando el programa las invoca explícitamente, mientras que las interrupciones pueden ocurrir en cualquier momento durante la ejecución. Además, las llamadas al sistema están diseñadas para proporcionar servicios específicos del SO a las aplicaciones (como abrir archivos o crear procesos), mientras que las interrupciones manejan eventos del hardware como entrada de teclado, señales del reloj o errores de hardware [6].

1.3. Pregunta#3 : Busque al menos 10 llamadas al sistema del estándar de POSIX y anote su utilidad.

Respuesta: A continuación se presentan 10 llamadas al sistema fundamentales del estándar POSIX con sus utilidades [7, 8]:

Llamada al Sistema	**Utilidad**
`fork()`	Crea un nuevo proceso hijo duplicando el proceso padre actual
`execve()`	Reemplaza la imagen del proceso actual con un nuevo programa
`waitpid()`	Espera a que un proceso hijo específico termine su ejecución
`exit()`	Termina el proceso actual y devuelve un código de estado al padre
`open()`	Abre un archivo o dispositivo y retorna un descriptor de archivo
`read()`	Lee datos desde un descriptor de archivo hacia un buffer
`write()`	Escribe datos desde un buffer hacia un descriptor de archivo
`close()`	Cierra un descriptor de archivo abierto liberando sus recursos
`stat()`	Obtiene información detallada sobre un archivo (tamaño, permisos, etc.)
`mkdir()`	Crea un nuevo directorio con los permisos especificados

Cuadro 1: Llamadas al sistema POSIX fundamentales y sus funciones

1.4. Pregunta#4 : Cuantas llamadas al sistema (aproximadamente, ya que puede variar de la versión) tiene un entorno basado en Linux.

Respuesta: El número de llamadas al sistema en Linux varía según la versión del kernel y la arquitectura del procesador. En kernels modernos de Linux (versión 5.x y 6.x), la arquitectura x86_64 implementa aproximadamente entre 330 y 450 llamadas al sistema [7]. Por ejemplo, el kernel Linux 5.15 LTS para x86_64 define alrededor de 335 llamadas al sistema, mientras que el kernel 6.0 contiene aproximadamente 450 [9]. Esta variación se debe a que Linux continúa evolucionando, agregando nuevas syscalls para soportar funcionalidades modernas como espacios de nombres, cgroups, y características de seguridad avanzadas, mientras mantiene compatibilidad hacia atrás con las llamadas al sistema tradicionales de UNIX [3]. Es importante notar que no todas las llamadas al sistema están disponibles en todas las arquitecturas; por ejemplo, ARM64 puede tener un conjunto ligeramente diferente comparado con x86_64 [6].

2. Desarrollo de la sección práctica

En esta sección se documenta el procedimiento seguido para **crear** y **ejecutar** las llamadas al sistema (*system calls*). Primero se muestran los **comandos iniciales de preparación del entorno**; luego, la **creación de las dos llamadas al sistema** en el código fuente; y, finalmente, las **pruebas de ejecución y verificación** desde espacio de usuario.

2.1. Creación de las llamadas al sistema

Para iniciar, en la Figura 1 se ilustran los **comandos de preparación** utilizados antes de modificar el código del kernel. Estos pasos aseguran un entorno controlado de desarrollo y de pruebas.

2.2. Descompresión del código fuente del kernel

El comando `sudo tar -xvjf linux-source-*.tar.bz2` se utiliza para descomprimir el código fuente del kernel de Linux. Cada letra en las opciones `-xvjf` tiene un significado específico: **x** (extract) extrae los archivos del archivo tar, **v** (verbose) muestra el progreso detallado de los archivos que se están extrayendo, **j** indica que el archivo está comprimido con bzip2 (extensión .bz2), y **f** (file) especifica que el siguiente argumento es el nombre del archivo a procesar. El asterisco (*) actúa como comodín para tomar cualquier versión del kernel disponible en el directorio.

```
linux-source-6.14.0/net/dccp/ccids/lib/tfrc.c
linux-source-6.14.0/net/dccp/ccids/lib/loss_interval.c
linux-source-6.14.0/net/dccp/ccids/lib/packet_history.h
linux-source-6.14.0/net/dccp/ccids/lib/loss_interval.h
linux-source-6.14.0/net/dccp/ccids/lib/tfrc.h
linux-source-6.14.0/net/dccp/ccids/lib/tfrc_equation.c
linux-source-6.14.0/net/dccp/ccids/lib/packet_history.c
linux-source-6.14.0/net/dccp/ccids/ccid2.c
linux-source-6.14.0/net/dccp/ccids/ccid2.h
linux-source-6.14.0/net/dccp/ccids/ccid3.c
linux-source-6.14.0/net/dccp/ccids/ccid3.h
linux-source-6.14.0/net/dccp/ipv6.h
linux-source-6.14.0/net/dccp/ackvec.h
linux-source-6.14.0/net/dccp/ccid.c
linux-source-6.14.0/net/dccp/feat.c
linux-source-6.14.0/CREDITS
adriel@adriel-System:/usr/src$ cd linux-source-6.14.0/
adriel@adriel-System:/usr/src/linux-source-6.14.0$ sudo nano kernel/helloworld.c
adriel@adriel-System:/usr/src/linux-source-6.14.0$ sudo nano kernel/misyscalladriel.c
adriel@adriel-System:/usr/src/linux-source-6.14.0$ sudo nano kernel/Makefile
```

Figura 1: Preparación del entorno 1: comandos iniciales previos a la edición del kernel.

```
asmlinkage long sys_helloworld(void);
asmlinkage long sys_MiSysCallAdriel(int numero);

#endif
```

Figura 2: Preparación del entorno 2: comandos iniciales previos a la edición del kernel.

```
465     common  listxattrat             sys_listxattrat
466     common  removexattrat           sys_removexattrat
467     common  helloworld              sys_helloworld
468     common  MiSysCallAdriel         sys_MiSysCallAdriel

#
```

Figura 3: Preparación del entorno 3: comandos iniciales previos a la edición del kernel.

A continuación, se presenta la **creación de las dos llamadas al sistema**. Cada subfigura muestra los cambios relevantes para su registro, implementación y exposición a espacio de usuario.

```
 GNU nano 8.3                    kernel/helloworld.c
#include <linux/kernel.h>
#include <linux/syscalls.h>

SYSCALL_DEFINE0(helloworld)
{
        printk(KERN_INFO "Hola desde la syscall helloworld \n");
        return 8888;
}
```

(a) Creación de la llamada al sistema 1.

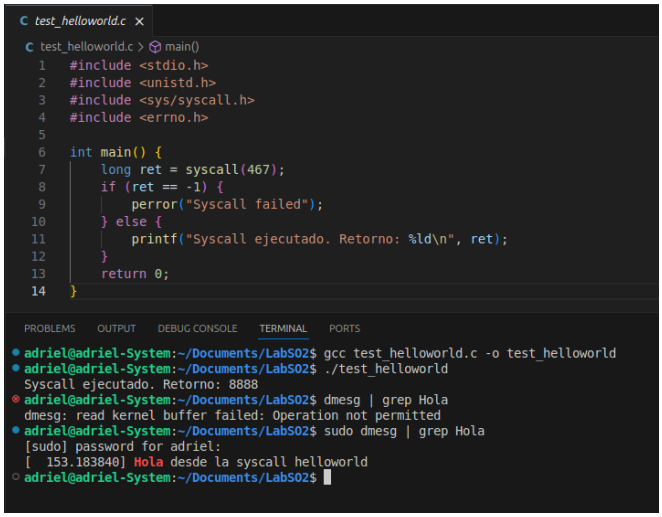
```
 GNU nano 8.3                    kernel/misyscalladriel.c
// kernel/misyscalladriel.c
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/tty.h>
#include <linux/string.h>

static void tty_write_all(struct tty_struct *tty, const char *s, size_t len)
{
    while (len > 0) {
        int n = tty->ops && tty->ops->write ? tty->ops->write(tty, s, len) : 0;
        if (n <= 0)
            break;
        s   += n;
        len -= n;
    }
    if (tty->ops && tty->ops->flush_chars)
        tty->ops->flush_chars(tty);
}

SYSCALL_DEFINE1(MiSysCallAdriel, int, numero)
{
    struct tty_struct *tty;
    char buffer[128];
    long long factorial = 1;
    int i;
    size_t len;

    if (numero < 0)  return -EINVAL;
    if (numero > 20) return -ERANGE;

    for (i = 1; i <= numero; i++)
        factorial *= i;

^G Help      ^O Write Out ^F Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^/ Go To Line
```

(b) Creación de la llamada al sistema 2.

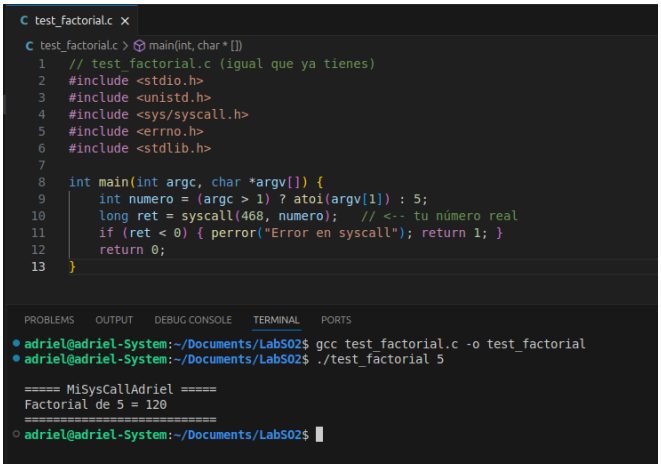
Figura 4: Registro e implementación de las llamadas al sistema en el código del kernel.

2.3. Ejecución de las llamadas al sistema

Finalmente, se documentan las **pruebas de ejecución** para validar el comportamiento de ambas llamadas. En la Figura 5 se observan las invocaciones desde espacio de usuario y los resultados obtenidos (*códigos de retorno, salida estándar y/o trazas*).



(a) Prueba de la llamada al sistema 1: invocación y resultado.



(b) Prueba de la llamada al sistema 2: invocación y resultado.

Figura 5: Ejecución y verificación de las llamadas al sistema desarrolladas.

2.4. Syscall `MiSysCallAdriel`: factorial con salida a TTY

Este es el código en C del syscall `MiSysCallAdriel`, que calcula el factorial de un número entero (válido en el rango $[0, 20]$ para evitar overflow en `long long`), escribe el resultado en la TTY si está disponible y devuelve el factorial como valor de retorno. En caso de error retorna `-EINVAL` (número negativo) o `-ERANGE` (mayor a 20).

Listing 1: Implementación del syscall MiSysCallAdriel (archivo: kernel/misyscalladriel.c)

```
1  // kernel/misyscalladriel.c
2  #include <linux/kernel.h>
3  #include <linux/syscalls.h>
4  #include <linux/tty.h>
5  #include <linux/string.h>
6
7  static void tty_write_all(struct tty_struct *tty, const char *s, size_t len)
8  {
9      while (len > 0) {
10         int n = tty->ops && tty->ops->write ? tty->ops->write(tty, s, len) : 0;
11         if (n <= 0)
12             break;
13         s += n;
14         len -= n;
15     }
16     if (tty->ops && tty->ops->flush_chars)
17         tty->ops->flush_chars(tty);
18 }
19
20 SYSCALL_DEFINE1(MiSysCallAdriel, int, numero)
21 {
22     struct tty_struct *tty;
```

```
23      char buffer[128];
24      long long factorial = 1;
25      int i;
26      size_t len;
27
28      if (numero < 0) return -EINVAL;
29      if (numero > 20) return -ERANGE;
30
31      for (i = 1; i <= numero; i++)
32          factorial *= i;
33
34      tty = get_current_tty();
35      if (tty) {
36          tty_lock(tty);
37
38          /* CRLF para arrancar en columna 0 y bajar linea */
39          tty_write_all(tty, "\r\n===== MiSysCallAdriel =====\r\n",
40                        strlen("\r\n===== MiSysCallAdriel =====\r\n"));
41
42          len = scnprintf(buffer, sizeof(buffer),
43                          "Factorial de %d = %lld\r\n", numero, factorial);
44          tty_write_all(tty, buffer, len);
45
46          tty_write_all(tty, "============================\r\n",
47                        strlen("============================\r\n"));
48
49          tty_unlock(tty);
50          tty_kref_put(tty);
51      } else {
52          pr_info("MiSysCallAdriel: no hay TTY. factorial( %d)= %lld\n",
53                  numero, factorial);
54      }
55
56      return factorial;
57  }
```

3. Guia rapida para agregar y recompilar *syscalls* (Linux 6.14.x)

Decidí agregar esta sección para futuros trabajos en los que se requiera algo similar, de modo que quede una referencia directa de los comandos. Sin embargo, esto no fue solicitado en las instrucciones del laboratorio.

3.1. Agregar un nuevo syscall

Listing 2: Crear y registrar un nuevo syscall

```
cd /usr/src/linux-source-6.14.0
sudo chown -R $USER:$USER .

# 1) Implementacion
nano kernel/NUEVOSYSCALL.c

# 2) Declarar el objeto en kernel/Makefile
# (agrega al final de los anteriores:)
#   obj-y += NUEVOSYSCALL.o
nano kernel/Makefile

# 3) Registrar en la tabla de syscalls (x86_64)
# Formato: <nr> <tab> common <tab> NUEVOSYSCALL <tab> sys_NUEVOSYSCALL
# Elige un numero libre (revisa el final del archivo).
nano arch/x86/entry/syscalls/syscall_64.tbl

# 4) Declarar la llamada en el header
nano include/linux/syscalls.h
# -> Ir al final del archivo y agregar el prototipo que coincida:
#    Ejemplo clasico:
#      asmlinkage long sys_helloworld(void);
#    Si tu entrada en la tabla usa el wrapper moderno, declara ese:
#      asmlinkage long __x64_sys_NUEVOSYSCALL(<args>);
#    Guardar (Ctrl+O, Enter) y salir (Ctrl+X).

# 5) Compilar e instalar (incremental)
make -j"$(nproc)"
sudo make install
sudo update-initramfs -u -k 6.14.8
sudo update-grub
sudo reboot
```

3.2. Recompilar tras editar un syscall

Listing 3: Recompilacion e instalacion incremental

```
cd /usr/src/linux-source-6.14.0
make -j"$(nproc)"
sudo make install
sudo update-initramfs -u -k 6.14.8
sudo update-grub
sudo reboot
```

Referencias

[1] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2015. [Online]. Available: https://www.pearson.com/us/higher-education/program/Tanenbaum-Modern-Operating-Systems-4th-Edition/PGM80736.html

[2] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018. [Online]. Available: https://www.wiley.com/en-us/Operating+System+Concepts%2C+10th+Edition-p-9781119320913

[3] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010. [Online]. Available: https://www.pearson.com/us/higher-education/program/Love-Linux-Kernel-Development-3rd-Edition/PGM202532.html

[4] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Pearson, 2018. [Online]. Available: https://www.pearson.com/us/higher-education/program/Stallings-Operating-Systems-Internals-and-Design-Principles-9th-Edition/PGM1262980.html

[5] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005. [Online]. Available: https://www.oreilly.com/library/view/understanding-the-linux/0596005652/

[6] W. Mauerer, *Professional Linux Kernel Architecture*. Wrox, 2008. [Online]. Available: https://www.wiley.com/en-us/Professional+Linux+Kernel+Architecture-p-9780470343432

[7] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010. [Online]. Available: https://man7.org/tlpi/

[8] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*, 3rd ed. Addison-Wesley Professional, 2013. [Online]. Available: https://www.pearson.com/us/higher-education/program/Stevens-Advanced-Programming-in-the-UNIX-Environment-3rd-Edition/PGM141163.html

[9] The Linux Kernel Organization, "Linux system call table," 2023, accessed: 2024. [Online]. Available: https://www.kernel.org/doc/html/latest/userspace-api/unistd.html