

# Sistemas Operativos

## Practica 2

### *Administración de Procesos/Hilos*

Integrantes:

No. Lista	Nombre
23	Angelo Mihaelle Ojeda Gómez
18	Hernández Hernández Julio Cesar
20	Martínez Robledo Luis Antonio
11	Estrada Corona Alexis
9	Esponzoza García Hilario Sebastian
15	García Silva Jonathan

Grupo:

4CV3

## Contenido

Objetivo .....	3
Descripción .....	3
Control de Procesos .....	3
Control de archivos .....	3
Control de procesos.....	4
fork() (uso e hijos) .....	4
Codigo Fuente.....	4
Prueba de pantalla( EjemFork() e hijosFork() ) .....	5
Otras llamadas al sistema (Control de procesos) .....	6
Código Fuente (Llamadas al sistema) .....	6
Prueba de Pantalla.....	7
Control de archivos .....	7
Codigo Fuente.....	8
Prueba de pantalla .....	8
Terminal.....	8
Archivo .....	8
Conclusion .....	9
Referencias.....	10

## Objetivo

Entender y hacer uso de las llamadas al sistema en LINUX que permiten crear procesos.

## Descripción

Las llamadas al sistema lucen como simples funciones las cuales reciben sus parámetros de entrada y entregan un valor de salida, y de hecho lo son, solo que estas funciones son implementadas por el núcleo del S.O. Esto significa que será el S.O. quien recibirá los parámetros de entrada, ejecutará la función que le sea requerida, siempre y cuando sea posible y permitida, y devolverá los valores necesarios, así como también puede o no cambiar el estado de su estructura interna. De esto también podemos deducir que estas llamadas al sistema se ejecutarán en modo kernel. De hecho, el procesador está constantemente cambiando de modo usuario a modo kernel y viceversa (en cualquier S.O. multitarea, como lo es Linux, en el cual se requiere una mayor responsabilidad del S.O. para administrar los procesos, el procesador alterna entre ambos modos al menos unas cuantas miles de veces por segundo).

Las llamadas al sistema o system call son los mecanismos usados por una aplicación para solicitar un servicio al sistema operativo. Proveen una interfaz de programación que invoca los servicios que el sistema operativo nos ofrece. Por lo general están escritas en lenguajes de ensamble, aunque existen ciertos sistemas que permiten trabajarlos con lenguajes de alto nivel como C y C++, en cuyo caso se asemejan a llamadas a funciones o procedimientos.

- En modo kernel se encuentran disponibles todas las instrucciones y funcionalidades que la arquitectura del procesador es capaz de brindar, sin ningún tipo de restricciones. Es en este modo en el cual corre el kernel (núcleo) del sistema operativo. Por lo general es el kernel el único que corre en este modo.
- En modo usuario tan sólo un subconjunto de las instrucciones y funcionalidades que la arquitectura del procesador ofrece se encuentran disponibles. En este modo se ejecutan los procesos de los usuarios del sistema (todo proceso corriendo en el sistema pertenece a un usuario). Conocido también como userland.

## Control de Procesos

Todos los procesos de un sistema informático deben controlarse para que en cualquier momento se puedan detener u otros procesos los puedan dirigir. Para esto, las llamadas al sistema de esta categoría supervisan, por ejemplo, el inicio o la ejecución o la detención/terminación de procesos.

## Control de archivos

Los programas de aplicación requieren este tipo de llamadas al sistema para acceder a las operaciones típicas con archivos. Estos métodos de manipulación de archivos incluyen la creación, eliminación, apertura, cierre, escritura y lectura (create, delete, open, close, write y read). Ahora bien, ¿para qué son necesarias las llamadas al sistema? Como dijimos antes, para que los procesos puedan comunicarse con el kernel del S.O. Pero seamos un poco más específicos, las llamadas al sistema nos brindan un medio para obtener recursos del S.O., obtener información del mismo, establecer o cambiar el set de los recursos que se ofrecen.

# Control de procesos

## fork() (uso e hijos)

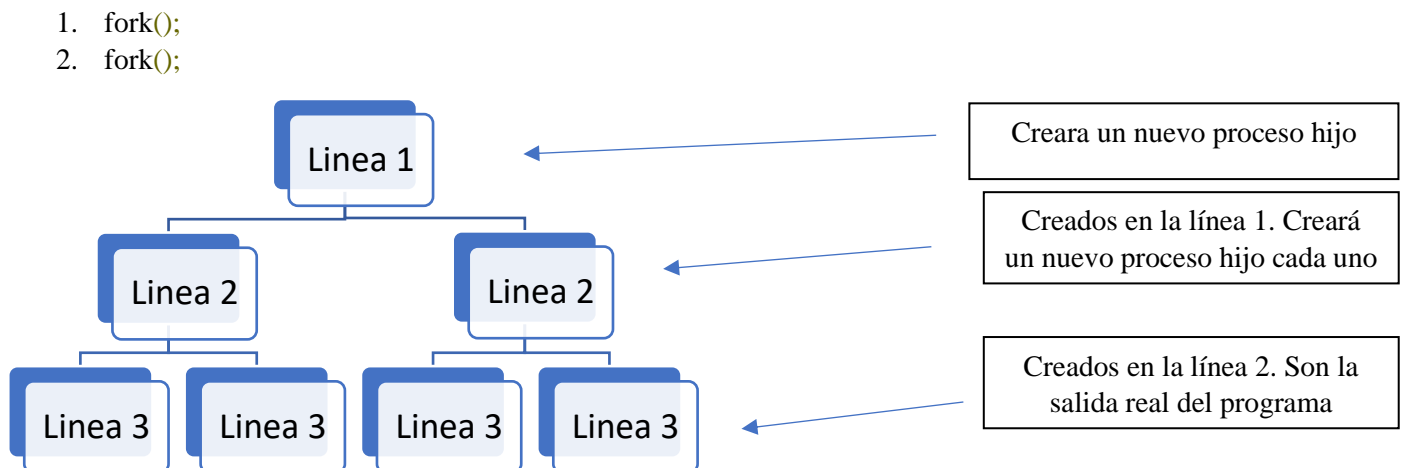
### Codigo Fuente

```
1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <unistd.h>
4.
5. void ejemFork(){
6.
7.     if(fork()==0){
8.         printf("Proceso hijo\n");
9.     }
10.    else{
11.        print("Proceso padre\n");
12.    }
13.
14. }
15.
16. void hijosFork(){
17.
18.     fork();
19.     fork();
20.     fork();
21.     printf("Hola!!");
22.
23. }
24.
25. int main(){
26.     hijosFork();
27.     ejemFork();
28.     return 0;
29. }
30.
```

### Al ejecutar la función hijosFork()

```
1. void hijosFork(){
2.
3.     fork();
4.     fork();
5.     fork();
6.     printf("Hola!!");
7.
8. }
```

se crea un proceso hijo creado por la línea uno (Primer fork();), lo que nos dejaría con 2 procesos, pero después cada uno de estos ejecuta otro fork() lo que al final nos dejaría con 8 procesos hijos, se puede ver de manera mas grafica con el siguiente diagrama:



## Prueba de pantalla( EjemFork() e hijosFork() )

Las salidas de nuestro código para esta función serian la siguiente ( 3 fork(); con 8 procesos hijos):

```
UsoFork.c
~/Documentos/myFile

Abrir Guardar x

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 void ejemFork(){
5
6     if(fork() == 0){
7         printf("Proceso hijo %d\n");
8     }
9     else{
10        printf("Proceso principal/padre %d\n");
11    }
12}
13 void hijosFork(){
14
15    fork();
16    fork();
17    fork();
18    printf("hello\n");
19
20
21}
22 int main()
23 {
24
25    hijosFork();
26    //ejemFork();
27    return 0;
28 }
```

```
angelo@debian: ~/Documentos/myFile$ gcc UsoFork.c -o usoFork
angelo@debian: ~/Documentos/myFile$ ./usoFork
hello
angelo@debian: ~/Documentos/myFile$ hello
hello
hello
hello
hello
hello
hello
S
```

Como podemos ver obtenemos el mensaje “Hello” en la pantalla luego de ejecutarlo, con los 8 procesos hijos, el problema es que estos procesos no terminan su ejecucion, por lo que debemos usar la instruccion `exit()`; para acabar con cada proceso hijo, agregandola al final de nuestra funcion obtendriamos la siguiente salida:

Actividades Editor de textos 18 de sep 11:47

UsoFork.c ~/Documentos/myFile

Guardar x

angelo@debian: ~/Documentos/myFile

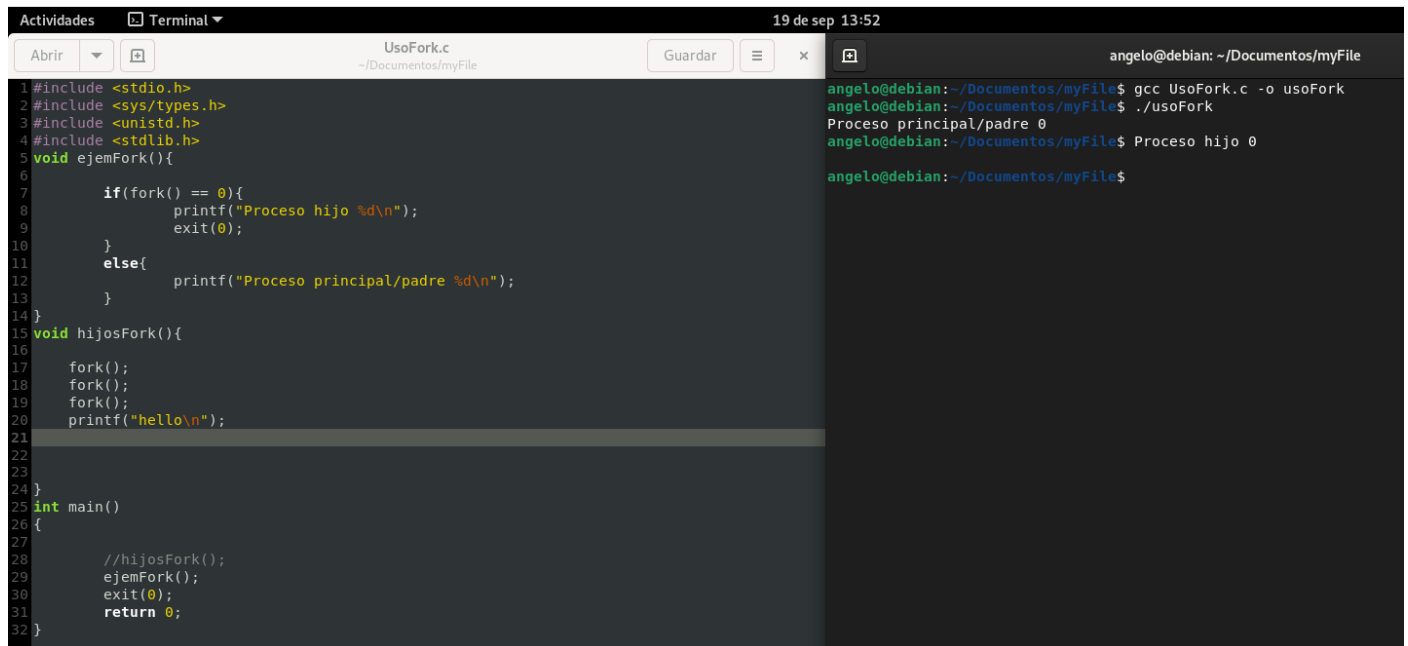
```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 void ejemFork(){
6
7     if(fork() == 0){
8         printf("Proceso hijo %d\n");
9     }
10    else{
11        printf("Proceso principal/padre %d\n");
12    }
13}
14 void hijosFork(){
15
16    fork();
17    fork();
18    fork();
19    printf("hello\n");
20
21}
22
23}
24 int main()
25 {
26
27    hijosFork();
28    //ejemFork();
29    exit(0);
30    return 0;
31 }
```

```
angelo@debian:~/Documentos/myFile$ gcc UsoFork.c -o usoFork
angelo@debian:~/Documentos/myFile$ ./usoFork
hello
angelo@debian:~/Documentos/myFile$ hello
hello
hello
hello
hello
hello
hello
angelo@debian:~/Documentos/myFile$
```

Ademas, la funcion fork retorna valores al ser ejecutada, este es un 0 cuando se crea un proceso hijo correctamente y algun valor positivo para el proceso padre, un valor negativo indicaria que no se pudo crear el proceso hijo, por lo que haciendo uso de nuestra funcion ejemFork(); podemos identificar en que punto de la ejecucion nos encontramos en el proceso:

```
1. void ejemFork(){
2.
3.     if(fork()==0){
4.         printf("Proceso hijo\n");
5.     }
6.     else{
7.         print("Proceso padre\n");
8.     }
9. }
```

La salida de este codigo seria la siguiente:



```
19 de sep 13:52
Abrir  UsoFork.c  Guardar  x  angelo@debian: ~/Documentos/myFile
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 void ejemFork(){
6
7     if(fork() == 0){
8         printf("Proceso hijo %d\n");
9         exit(0);
10    }
11    else{
12        printf("Proceso principal/padre %d\n");
13    }
14 }
15 void hijosFork(){
16
17     fork();
18     fork();
19     fork();
20     printf("hello\n");
21 }
22
23 }
24 }
25 int main()
26 {
27
28     //hijosFork();
29     ejemFork();
30     exit(0);
31     return 0;
32 }
angelo@debian:~/Documentos/myFile$ gcc UsoFork.c -o usoFork
angelo@debian:~/Documentos/myFile$ ./usoFork
Proceso principal/padre 0
angelo@debian:~/Documentos/myFile$ Proceso hijo 0
angelo@debian:~/Documentos/myFile$
```

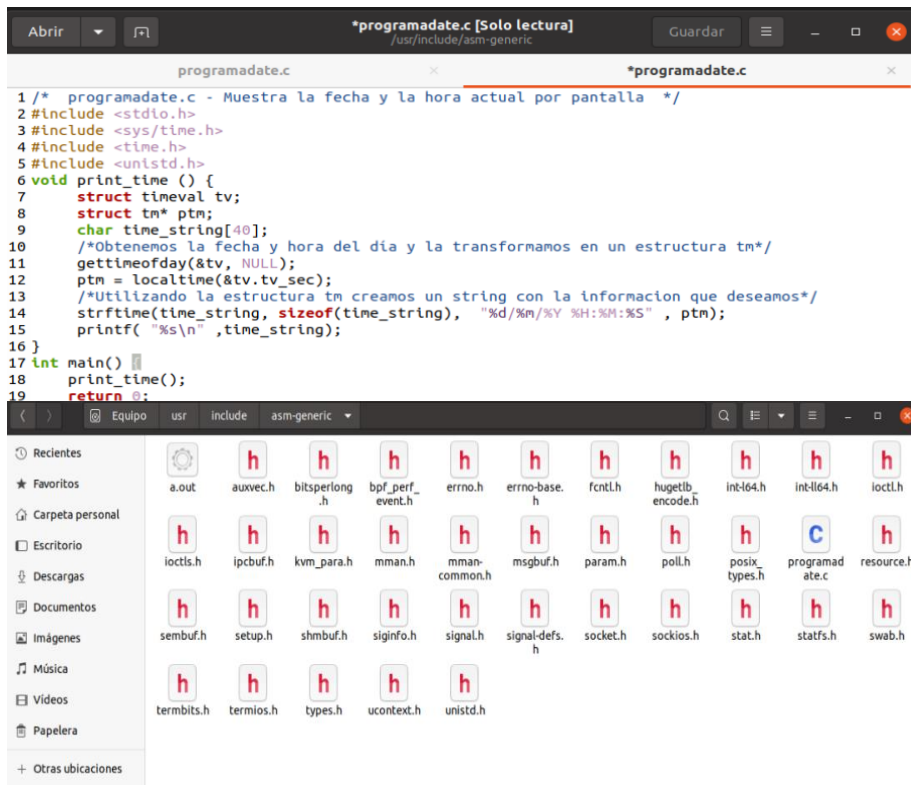
Como se puede ver en este caso se envia un mensaje desde nuestro proceso padre, y luego se crea un proceso hijo que al crearse correctamente regresa un valor de 0 y entonces podemos imprimir un mensaje desde allí.

## Otras llamadas al sistema (Control de procesos)

### Código Fuente (Llamadas al sistema)

```
1. /* programdate.c - Muestra la fecha y la hora actual por pantalla */
2. #include <stdio.h>
3. #include <sys/time.h>
4. #include <time.h>
5. #include <unistd.h>
6. void print_time () {
7.     struct timeval tv;
8.     struct tm* ptm;
9.     char time_string[40];
10. /*Obtenemos la fecha y hora del día y la transformamos en un estructura tm*/
11. gettimeofday(&tv, NULL);
12. ptm = localtime(&tv.tv_sec);
13. /*Utilizando la estructura tm creamos un string con la informacion que deseamos*/
14. strftime(time_string, sizeof(time_string), "%d/%m/%Y %H:%M:%S" , ptm);
15. printf( "%s\n" ,time_string);
16. }
17. int main() {
18. print_time();
19. return 0;
20. }
```

## Prueba de Pantalla



Captura de pantalla 1: En un editor de textos se escribe el código y se guarda con la extensión .c

Captura de pantalla 2: En la ruta especificada donde se encuentra el núcleo del sistema en modo kernel de Linux

```
[sudo] contraseña para antoniorobledo:
root@antoniorobledo-VirtualBox:/home/antoniorobledo# sudo cp /home/antoniorobledo/Documentos/programadate.c /usr/include/asm-generic
root@antoniorobledo-VirtualBox:/home/antoniorobledo# cd /usr/include/asm-generic
root@antoniorobledo-VirtualBox:/usr/include/asm-generic# gcc programadate.c
root@antoniorobledo-VirtualBox:/usr/include/asm-generic# ./a.out
12/09/2021 02:04:17
root@antoniorobledo-VirtualBox:/usr/include/asm-generic#
```

Captura de pantalla 3: Con el comando “sudo cp” se copia un archivo, en este caso nuestro programa en c para poder tener acceso a escribir en una carpeta del sistema (captura de pantalla 2), con el comando “gcc” compilamos el programa en c y se genera un ejecutable llamado

“a.out”(vease en captura de pantalla 2), y se ejecuta con el comando “./a.out” mostrando nuestro resultado encerrado en azul.

## Control de archivos

Hay muchas formas de utilizar este tipo de llamadas, al sistema, pero tambien son unas de las mas utiles, tanto o mas que praticamente todos los lenguajes de programacion vigentes las pueden utilizar de forma eficaz, para remarcar este punto utilizaremos python para esta parte, ya que en la anterior usamos C, cosa que seria complicada de hacer con este lenguaje. Para escribir o leer cadenas de caracteres para/desde archivos (otros tipos deben ser convertidas a cadenas de caracteres). Para esto Python incorpora un tipo integrado llamado [file](#), el cual es manipulado mediante un objeto archivo el cual fue generado a través de una función integrada en Python, a continuación se describen los procesos típicos y sus referencias a funciones propias del lenguaje, pero algunas tambien se pueden encontrar en algunos otros lenguajes:

- Abrir archivo
  - La forma preferida para abrir un archivo es usando la función integrada [open\(\)](#).
- Leer archivo
  - La forma preferida para leer un archivo es usando algunas de los métodos del tipo objeto [file](#) como [read\(\)](#), [readline\(\)](#) y [readlines\(\)](#).
- Escribir archivo
  - La forma preferida para escribir un archivo es usando el método del tipo objeto [file](#) llamado [write\(\)](#).
- Cerrar archivo
  - La forma preferida para cerrar un archivo es usando el método del tipo objeto [file](#) llamado [close\(\)](#).



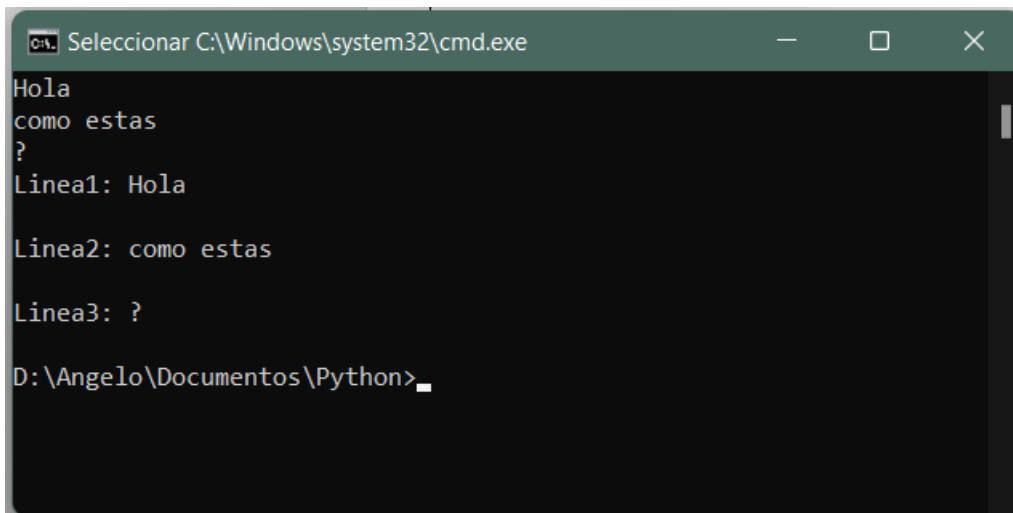
## Codigo Fuente

```
1. file = open ('holamundo.txt','r')
2. mensaje = file.read()
3. print(mensaje)
4. file.close()
5.
6. file = open("holamundo.txt",'r')
7. c =0
8. for i in file:
9.     c += 1
10.    print("Linea" + str(c) + ": " + i)
11. file.close()
12.
13. file = open("holamundo.txt",'a')
14. mensaje = "\nEsto es manipulacion de archivos"
15. file.write(mensaje)
16. file.close()
```

En este caso utilizamos funciones que nos dejan leer nuestro archivo en su totalidad, usando el modo “r” de las mismas, que significa read, luego usamos un ciclo for que nos deja recorrer cada línea de nuestro texto, cosa que, al igual que lo anterior, imprimimos en nuestra pantalla, luego de cada operación cerramos nuestro archivo, pero la última apertura del mismo se hace en modo “a” que viene de “append” lo que nos permite escribir en nuestro archivo en la última línea del mismo añadiendo nuestro texto al ya existente, eso nos deja con la siguiente salida y archivo

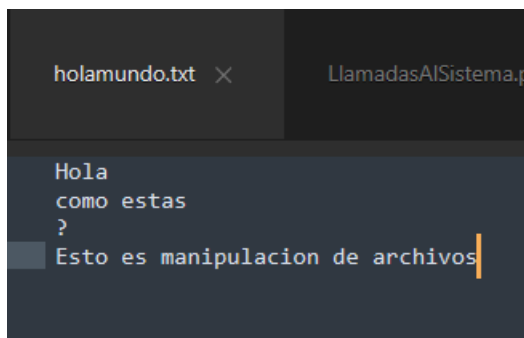
## Prueba de pantalla

### Terminal



```
Seleccionar C:\Windows\system32\cmd.exe
Hola
como estas
?
Linea1: Hola
Linea2: como estas
Linea3: ?
D:\Angelo\Documentos\Python>
```

### Archivo



En caso de abrir el archivo en modo “w” sobrescribiríamos el texto y perderíamos el texto que hayamos puesto antes, estas instrucciones funcionan casi en cualquier lenguaje tanto para linux como para windows



## Conclusion

Linux provee más de 300 llamadas al sistema diferentes que permiten al usuario que los procesos y el sistema operativo puedan comunicarse. Las llamadas al sistema son los mecanismos usados por una aplicación para solicitar un servicio al sistema operativo. La medida en que los tipos de llamadas al sistema enumerados se pueden implementar o realizar depende principalmente del hardware y de la arquitectura del sistema utilizados, pero también del sistema operativo. Las llamadas a sistema que revisamos durante esta practica pertenecen a 2 categoriar, principalmente: Control de procesos y Manipulación de archivos. Las llamadas de sistema tienen que ver con la creación, terminación y control de un proceso, la asignación y liberación de memoria, requerida y liberada por un proceso. La comunicación entre procesos son el modelo de paso de mensajes y el modelo de memoria compartida. El IPC es una función básica del sistema operativo, donde los procesos se comunican entre sí a través de compartir espacios de memoria.

## Referencias

- William J. Turkel y Adam Crymble, "Trabajar con archivos de texto en Python", traducido por Víctor Gayol, The Programming Historian en español 1 (2017), <https://doi.org/10.46430/phes0028>.
- Juan Carlos Perez y Sergio Saez. (n.d.). Tema 5. Llamadas al Sistema. Retrieved September 20, 2021, from <http://sop.upv.es/gii-dso/es/t5-llamadas-al-sistema/gen-t5-llamadas-al-sistema.html>
- L., F. A. (2019, August 12). Llamadas al sistema. Retrieved September 20, 2021, from <https://freddy-abadl.medium.com/llamadas-al-sistema-f71a77a6e6f3>
- Linux kernel. (2021, September 17). Retrieved September 20, 2021, from [https://en.wikipedia.org/wiki/Linux\\_kernel](https://en.wikipedia.org/wiki/Linux_kernel)
- Llamada al sistema. (2021, August 30). Retrieved September 20, 2021, from [https://es.wikipedia.org/wiki/Llamada\\_al\\_sistema](https://es.wikipedia.org/wiki/Llamada_al_sistema)
- System calls: ¿qué son y para qué se emplean? (n.d.). Retrieved September 20, 2021, from <https://www.ionos.mx/digitalguide/servidores/know-how/que-son-las-system-calls-de-linux/>
- Wielsch, M., Prahm, J., Esser, H., Liger, F., Wolf, P. M., & Springinsfeld, S. (2000). Linux. Paris: Micro Application.
- (n.d.). Retrieved September 20, 2021, from <https://man7.org/linux/man-pages/man2/syscalls.2.htm>