

Lab 1  
**Sistemas Operativos**  
Gus Méndez - 18500

## Ejercicio 1

```

12 root      20    0    0    0    0 S  0.0  0.0   0:00.01 sync_supers
13 root      20    0    0    0    0 S  0.0  0.0   0:00.01 bdi-default
os@debian:~/Desktop$ gcc -o program_2 program_2.c
os@debian:~/Desktop$ ./program_2
PID->2964
Hello World!
2964
os@debian:~/Desktop$ PID->2965
Hello World!
2965

```

Programa en C usando *fork()*

- Compile el primer programa y ejecútalo varias veces. Responda: ¿por qué aparecen números diferentes cada vez?

Esto se debe a que realmente crea un nuevo proceso en cada iteración, por eso es otro PID.

- Proceda a compilar el segundo programa y ejecútalo una vez. ¿Por qué aparecen dos números distintos a pesar de que estamos ejecutando un único programa?

Esto se debe a que el fork le crea un proceso hijo al proceso padre que se inicializó, entonces tiene otro PID.

- ¿Por qué el primer y el segundo números son iguales?

Cuando un proceso ejecuta exec, no se crea ningún proceso nuevo. El proceso de llamada es sobrescrito por el programa cuyo nombre de archivo se pasa como primer argumento. Al ejecutar el *fork()*, se hace simultáneamente.

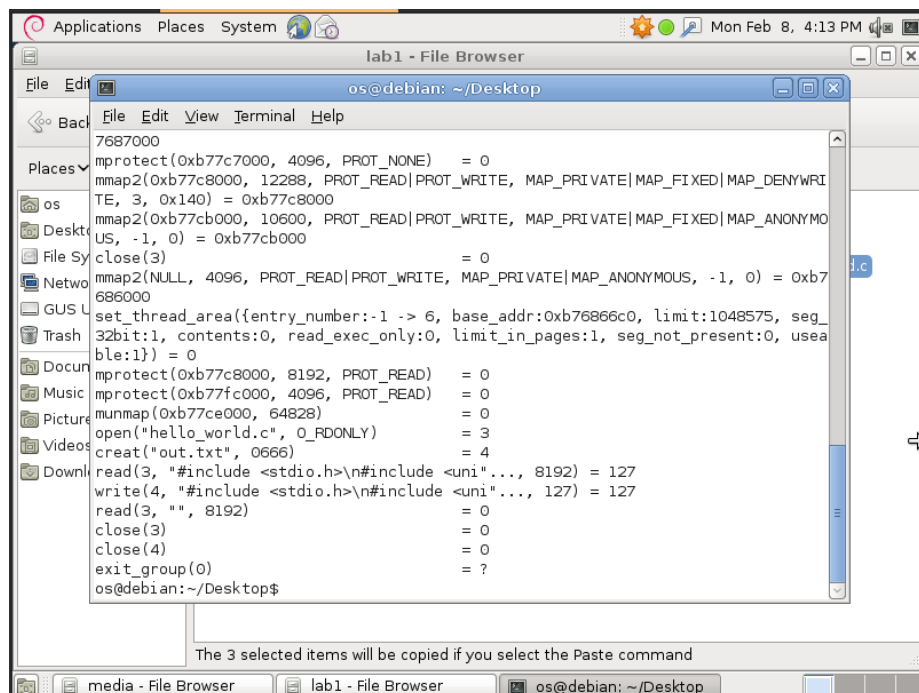
- En la terminal, ejecute el comando `top` (que despliega el top de procesos en cuanto a consumo de CPU) y note cuál es el primer proceso en la lista (con identificador 1). ¿Para qué sirve este proceso?

El primer proceso iniciado por el kernel de Linux obtiene PID 1. INIT es el primer programa que se ejecuta en el sistema Linux. Es por eso que se le da el PID 1 y el ID de proceso siempre comenzando desde 1 en Linux. Como sabemos, INIT es el primer proceso que se inicia y también es el padre de todos los demás procesos en Linux. El PID 1 es el principal responsable de iniciar y apagar el sistema, y cuando un proceso muere y deja hijos, se le asignan a él.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1383	root	20	0	41724	17m	7580	S	0.3	4.6	0:17.24	Xorg
1	root	20	0	2036	724	628	S	0.0	0.2	0:00.70	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/0
5	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
6	root	20	0	0	0	0	S	0.0	0.0	0:00.10	events/0
7	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cruisr

Init tiene PID 1

## Ejercicio 2



Resultado de strace al programa de copiado de archivos

- Observe el resultado desplegado. ¿Por qué la primera llamada que aparece es `execve`?

En Linux, para crear un nuevo proceso cuyo programa es diferente al programa del proceso original, el nuevo proceso hijo llama inmediatamente a `execve()`. Por tanto, `execve` reemplaza el proceso actual con un nuevo proceso, ejecutando el comando que especificó como su primer argumento.

- Ubique las llamadas de sistema realizadas por usted. ¿Qué significan los resultados (números que están luego del signo '=')?

Cada línea del output contiene el nombre de la rutina del kernel y sus parámetros, así como el valor de retorno de la llamada al sistema (el valor entero luego del signo '=').

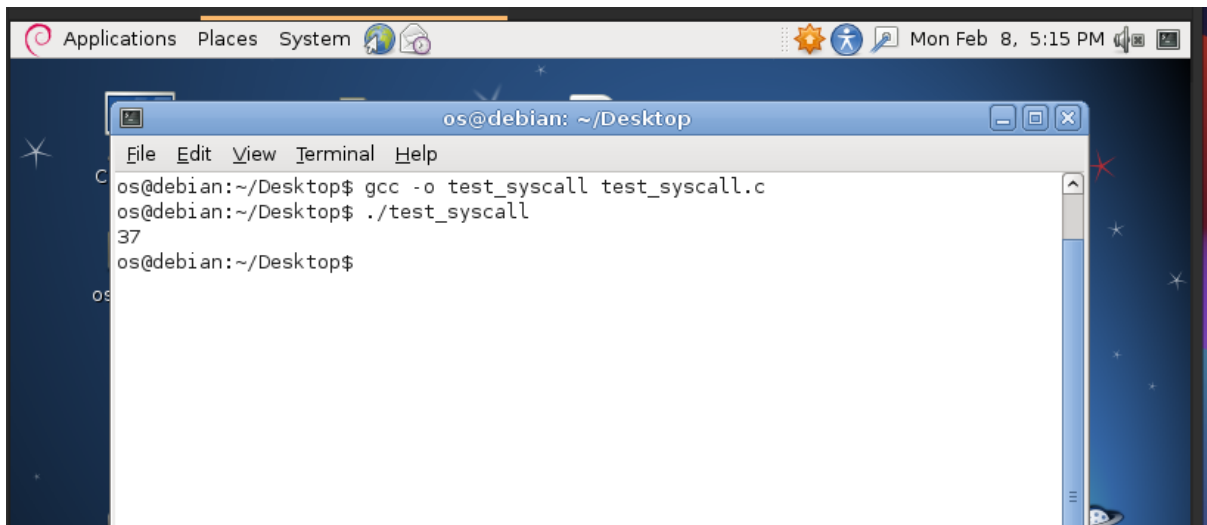
- ¿Por qué entre las llamadas realizadas por usted hay un read vacío?

Este coincide con la última línea del programa o archivo, lo cual la rutina del kernel interpreta como un EOF.

- Identifique tres servicios distintos provistos por el sistema operativo en este `strace`. Liste y explique brevemente las llamadas a sistema que corresponden a los servicios identificados (puede incluir `read`, `write`, `open` o `close` que el sistema haga por usted, no los que usted haya producido directamente con su programa).

- `mmap2`: crea una nueva asignación en el espacio de direcciones virtuales del proceso de llamada. La dirección de inicio para la nueva asignación está especificada en `addr`. El argumento de longitud especifica la longitud del mapeo (que debe ser mayor que 0).
- `fstat64`: Estas funciones devuelven información sobre un archivo. No se requieren permisos sobre el archivo. Además, devuelve atributos en una estructura `struct stat64`, que representa tamaños de archivo con un tipo de 64 bits, lo que permite que las funciones funcionen en archivos de 2GB y más.
- `mummap`: elimina las asignaciones para el rango de direcciones especificado y provoca más referencias a direcciones dentro del rango para generar referencias de memoria no válidas. La región también se desasigna automáticamente cuando finaliza el proceso.

### Ejercicio 3



```
os@debian: ~/Desktop
File Edit View Terminal Help
os@debian:~/Desktop$ gcc -o test_syscall test_syscall.c
os@debian:~/Desktop$ ./test_syscall
37
os@debian:~/Desktop$
```

Resultado de syscall, con valor de 22

- ¿Qué ha modificado aquí, la interfaz de llamadas de sistema o el API? Justifique su respuesta.

Solamente la interfaz de llamadas al sistema, ya que solamente hicimos un nuevo método que permite a un programa (en C) solicitar servicios del kernel. La interfaz de llamada al sistema invoca la llamada al sistema prevista en el kernel del sistema operativo y devuelve el estado de la llamada al sistema y un valor (en nuestro caso, una suma).

- ¿Por qué usamos el número de nuestra llamada de sistema en lugar de su nombre?

Porque la misma no pertenece directamente a la API, sino que la llamada al sistema se implementará manejando una interrupción de software. Esta interrupción disparada le dará el control a un manejador de llamadas al sistema que, basado en un ID de llamada al sistema, decidirá qué función debe llamarse para manejar esa interrupción específica. Estrictamente hablando, el uso de una API no requiere invocar un fragmento de código del kernel, mientras que invocar la llamada al sistema sí invoca al kernel.

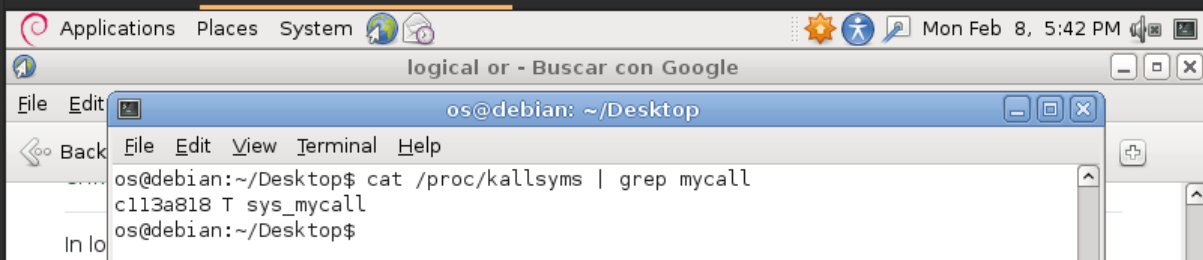
- ¿Por qué las llamadas de sistema existentes como read o fork se pueden llamar por nombre?

Esto es porque todas estas funciones son funciones de espacio de usuario reales en libc., de modo que su binario está vinculado. Pero la mayoría de ellos son pequeños

*wrappers* para llamadas al sistema que son la interfaz entre el espacio de usuario y el kernel. Por tanto, *read()* y *fork()* tienen *wrappers* en la librería de C.

La desventaja de la llamada al sistema por ID es que es menos claro, no portátil y omite las comprobaciones en tiempo de compilación sobre el número y tipo de argumentos.

- Incluya entre sus respuestas una captura de pantalla con el resultado de la ejecución de su llamada a sistema.



The screenshot shows a terminal window titled 'os@debian: ~/Desktop'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Terminal', and 'Help'. The command prompt shows the user 'os@debian' in the directory '~/Desktop'. The command entered is 'cat /proc/kallsyms | grep mycall'. The output of the command is 'c113a818 T sys\_mycall'. The prompt then returns to 'os@debian:~/Desktop\$'.

Prueba que sys\_mycall existe