

Taller de *syscalls* y señales

Sistemas Operativos

Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

22 de agosto de 2017

Segundo cuatrimestre de 2017

¿Cómo interactuamos con el SO?

- Como **usuarios**: programas o utilidades de sistema.
Por ejemplo: `ls`, `time`, `mv`, `who`, `akw`, etc.
- Como **programadores**: llamadas al sistema o *syscalls*.
Por ejemplo: `time()`, `open()`, `write()`, `fork()`, `wait()`, etc.

★ Ambos mecanismos suelen estar estandarizados.

★ Linux sigue el estándar **POSIX** (Portable Operating System Interface [for UNIX]).

- ★ Las *syscalls* proveen una **interfaz** a los servicios brindados por el sistema operativo: la API (Application Programming Interface) del SO.
- ★ La mayoría de los programas hacen un uso intensivo de ellas.
- ★ Implementación: en general, se usa una interrupción para pasar a modo *kernel*, y los parámetros se pasan usando registros o una tabla en memoria. En Linux: interrupción **0x80** (en 32 bits); el **número de syscall** va por EAX (o RAX).

- ★ Las *syscalls* proveen una **interfaz** a los servicios brindados por el sistema operativo: la API (Application Programming Interface) del SO.
- ★ La mayoría de los programas hacen un uso intensivo de ellas.
- ★ Implementación: en general, se usa una interrupción para pasar a modo *kernel*, y los parámetros se pasan usando registros o una tabla en memoria. En Linux: interrupción **0x80** (en 32 bits); el **número de syscall** va por EAX (o RAX).
- ★ Normalmente se las utiliza a través de *wrapper functions* en C. ¿Por qué no directamente?

Un primer ejemplo

tinyhello.asm

```
section .data
hello: db 'Hola SO!', 10
hello_len: equ $-hello

section .text
global _start
_start:
    mov eax, 4 ; syscall write
    mov ebx, 1 ; stdout
    mov ecx, hello ; mensaje
    mov edx, hello_len
    int 0x80

    mov eax, 1 ; syscall exit
    mov ebx, 0 ;
    int 0x80
```

Usando *wrapper functions* en C

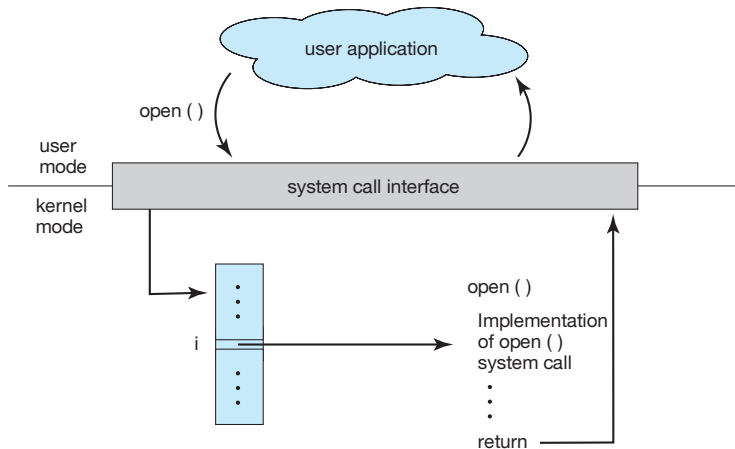
- ★ Claramente, el código anterior no es *portable*.
- ★ Además, realizar una *syscall* de esta forma requiere programar en lenguaje ensamblador.
- ★ Las *wrapper functions* permiten interactuar con el sistema con mayor **portabilidad** y **sencillez**.

El ejemplo anterior, pero ahora en C:

```
hello.c
#include <unistd.h>

int main(int argc, char* argv[]) {
    write(1, "Hola S0!\n", 9);
    return 0;
}
```

Usando *wrapper functions* en C

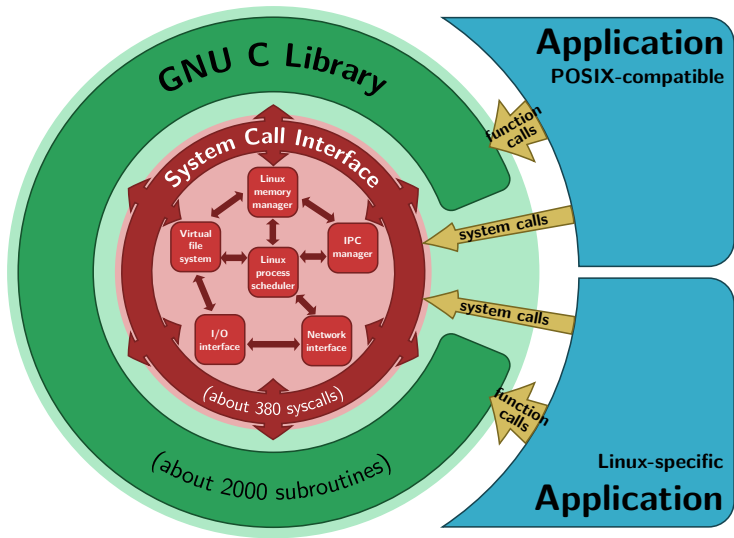


Invocación de la *syscall* `open()` desde una aplicación de usuario.

Imagen extraída de *Operating System Concepts* (Abraham Silberschatz et al.)

- ★ Pueden ver las *syscalls* con `man syscalls`
- ★ La biblioteca estándar de C incluye funciones que no son *syscalls*, pero las utilizan para funcionar. Por ejemplo, `printf()` invoca a la *syscall* `write()`. Estas funciones se detallan en la sección 3 del manual.
- ★ Están definidas en el archivo `unistd.h` de la biblioteca estándar de C. Puede verse una lista de todas ellas usando `man syscalls`.

Syscalls en Linux



Basado en una ilustración de Shmuel Csaba Otto Traian (Wikimedia Commons)



Signals, rules!



HELLO IT
DID YOU TRY TURNING
IT OFF AND ON AGAIN?

- ★ Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
- ★ Cuando un proceso recibe una señal, su ejecución se interrumpe y se ejecuta un *handler*.
- ★ Cada tipo de señal tiene asociado un *handler* por defecto, que puede ser modificado mediante la `syscall signal()`.

- ★ Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
- ★ Cuando un proceso recibe una señal, su ejecución se interrumpe y se ejecuta un *handler*.
- ★ Cada tipo de señal tiene asociado un *handler* por defecto, que puede ser modificado mediante la `syscall` `signal()`.
- ★ Toda señal tiene asociado un número que identifica su tipo. Estos números están definidos como constantes en el *header* `<signal.h>`. Por ejemplo: `SIGINT`, `SIGKILL`, `SIGSEGV`.
- ★ Las señales `SIGKILL` y `SIGSTOP` no pueden ser bloqueadas, ni se pueden reemplazar sus *handlers*.
- ★ Un usuario puede enviar una señal a un proceso con la herramienta `kill`. Un proceso puede enviar una señal a otro mediante la `syscall` `kill()`.

strace es una herramienta que nos permite generar una traza legible de las llamadas al sistema usadas por un programa dado.

Ejemplo de strace

```
$ strace -q echo hola > /dev/null
```

Algunas opciones útiles:

- `-q`: Omite algunos mensajes innecesarios.
- `-o <archivo>`: Redirige la salida a `<archivo>`.
- `-f`: Traza también a los procesos hijos del proceso trazado.

Usando strace

strace es una herramienta que nos permite generar una traza legible de las llamadas al sistema usadas por un programa dado.

Ejemplo de strace

```
$ strace -q echo hola > /dev/null
execve("/bin/echo", ["echo", "hola"], [/* 70 vars */]) = 0
write(1, "hola\n", 5)                = 5
exit_group(0)                        = ?
```

Usando strace

strace es una herramienta que nos permite generar una traza legible de las llamadas al sistema usadas por un programa dado.

Ejemplo de strace

```
$ strace -q echo hola > /dev/null
execve("/bin/echo", ["echo", "hola"], [/* 70 vars */]) = 0
write(1, "hola\n", 5)                = 5
exit_group(0)                        = ?
```

- `execve()` convierte el proceso en una instancia nueva de `./tinyhello` y devuelve 0 indicando que no hubo error.
- `write()` escribe en pantalla el mensaje y devuelve la cantidad de caracteres escritos (5).
- `exit_group()` termina la ejecución(y de todos sus *threads*) y no devuelve ningún valor.

strace y hello en C

Probemos strace con nuestra versión en C del programa.

```
hello.c
```

```
#include <unistd.h>

int main(int argc, char* argv[]) {
    write(1, "Hola SO!\n", 9);
    return 0;
}
```

Vamos a compilar estáticamente:

```
Compilación de hello.c
```

```
gcc -static -o hello hello.c
```


strace y hello en C

strace de hello.c

```
$ strace -q ./hello
execve("./hello", ["/hello"], [/* 17 vars */]) = 0
uname({sys="Linux", node="nombrehost", ...}) = 0
brk(0)                                = 0x831f000
brk(0x831fcb0)                        = 0x831fcb0
set_thread_area({entry_number:-1 -> 6, base_addr:0x831f830...}) = 0
brk(0x8340cb0)                        = 0x8340cb0
brk(0x8341000)                        = 0x8341000
write(1, "Hola S0!\n", 9)             = 9
exit_group(0)                         = ?
```

¿Qué es todo esto?

Llamadas referentes al manejo de memoria

<code>brk(0)</code>	<code>= 0x831f000</code>
<code>brk(0x831fcb0)</code>	<code>= 0x831fcb0</code>
<code>brk(0x8340cb0)</code>	<code>= 0x8340cb0</code>
<code>brk(0x8341000)</code>	<code>= 0x8341000</code>

- `brk()` y `sbrk()` modifican el tamaño de la memoria de datos del proceso. `malloc()` y `free()` (que no son *syscalls*) las usan para agrandar o achicar la memoria usada por el proceso.
- Otras comunes suelen ser `mmap()` y `mmap2()`, que asignan un archivo o dispositivo a una región de memoria. En el caso de `MAP_ANONYMOUS` no se mapea ningún archivo; solo se crea una porción de memoria disponible para el programa. Para regiones de memoria grandes, `malloc()` usa esta *syscall*.

¿Y compilando dinámicamente?

- Compilemos el mismo fuente `hello.c` con bibliotecas dinámicas (sin `-static`).
- Si corremos `strace` sobre este programa, encontramos **aún más** *syscalls*:

strace de `hello.c`, compilado dinámicamente

```
...
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or ...)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb8017000
access("/etc/ld.so.preload", R_OK)     = -1 ENOENT (No such file or ...)
open("/etc/ld.so.cache", O_RDONLY)     = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=89953, ...}) = 0
mmap2(NULL, 89953, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb8001000
close(3)                                = 0
...
```

¿Y strace cómo funciona?

¿Y strace cómo funciona?

El secreto es la `syscall ptrace()`. Veamos qué tiene para decir el manual.

man 2 ptrace

NOMBRE

`ptrace` - rastreo de un proceso

SINOPSIS

```
#include <sys/ptrace.h>
```

```
long ptrace(enum __ptrace_request request, pid_t pid,  
            void *addr, void *data);
```

DESCRIPCIÓN

La llamada al sistema `ptrace` proporciona un medio por el que un proceso padre puede observar y controlar la ejecución de un proceso hijo y examinar y cambiar su imagen de memoria y registros. Se usa principalmente en la implementación de depuración con puntos de ruptura y en el rastreo de llamadas al sistema.

Usando ptrace()

Vamos a usar ptrace() para monitorear un proceso.

Prototipo de ptrace()

```
long ptrace(enum __ptrace_request request, pid_t pid,  
            void *addr, void *data);
```

request puede ser alguno de estos:

- PTRACE_TRACEME, PTRACE_ATTACH, PTRACE_DETACH,
- PTRACE_KILL, PTRACE_CONT,
- PTRACE_SYSCALL, PTRACE_SINGLESTEP,
- PTRACE_PEEKDATA, PTRACE_POKEDATA,
- PTRACE_PEEKUSER, PTRACE_POKEUSER,
- ...y más¹.

¹Ver man 2 ptrace.

Usando ptrace()

Situación:

- Proceso **padre**.
- Proceso **hijo** que queremos monitorear.

Inicialización. Dos alternativas:

- 1 El proceso hijo solicita ser monitoreado por su padre haciendo una llamada a `ptrace(PTRACE_TRACEME)`. Por ejemplo, después de un `fork()` y antes de un `execve()`.
- 2 El proceso padre se engancha al proceso hijo con la llamada `ptrace(PTRACE_ATTACH, pid_child)`. Esto permite engancharse a un proceso que ya está corriendo (si se tienen permisos suficientes).

Finalización:

- Con la llamada `ptrace(PTRACE_DETACH, pid_child)` se deja de monitorear.

Usando ptrace()

ptrace() permite monitorear tres tipos de **eventos**:

- 1 Señales: cuando el proceso hijo recibe una señal.
- 2 *Syscalls*: cada vez que el proceso hijo entra o sale de la llamada a una *syscall*.
- 3 Instrucciones: cuando el proceso hijo ejecuta **una** instrucción.

Usando ptrace()

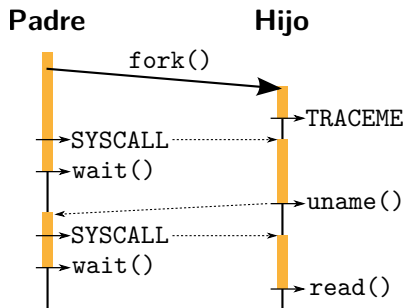
ptrace() permite monitorear tres tipos de **eventos**:

- 1 Señales: cuando el proceso hijo recibe una señal.
- 2 *Syscalls*: cada vez que el proceso hijo entra o sale de la llamada a una *syscall*.
- 3 Instrucciones: cuando el proceso hijo ejecuta **una** instrucción.

Cada vez que se genera un **evento**, el proceso hijo se detiene. El padre se entera mediante una llamada (bloqueante) a la *syscall* `wait()`, que retorna al producirse el evento. Luego, puede **reanudar** al hijo hasta:

- 1 la siguiente señal recibida, llamando a `ptrace(PTRACE_CONT)`.
- 2 la siguiente señal recibida o *syscall* ejecutada, llamando a `ptrace(PTRACE_SYSCALL)`.
- 3 solo por una instrucción, llamando a `ptrace(PTRACE_SINGLESTEP)`.

ptrace(): Esquema de uso (simplificado)

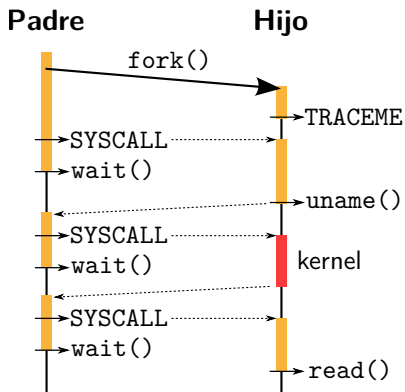


Ejemplo **simplificado** del mecanismo de bloqueo de `ptrace()`.

El hijo se detiene cada vez que llama a una *syscall*.

El padre lo reanuda con una llamada a `ptrace(PTRACE_SYSCALL)`.

ptrace(): Esquema de uso



En realidad, el padre recibe **dos** eventos, al entrar y salir de la *syscall*.

ptrace(): Esquema de comunicación

- ❶ Se inicializa el mecanismo de `ptrace()` (`PTRACE_TRACEME` o `PTRACE_ATTACH`).
- ❷ **padre**: Llama a `wait()`; espera el próximo evento del hijo.
- ❸ **hijo**: Ejecuta normalmente hasta que se genere un evento (recibir una señal, hacer una *syscall* o ejecutar una instrucción).
- ❹ **hijo**: Se genera el evento y el proceso se detiene.
- ❺ **padre**: Vuelve de la *syscall* `wait()`.
- ❻ **padre**: Puede inspeccionar y modificar el estado del hijo: registros, memoria, etc.
- ❼ **padre**:
 - Reanuda el proceso hijo con `PTRACE_CONT`, `PTRACE_SYSCALL` o `PTRACE_SINGLESTEP` y vuelve a 2,
 - o bien termina el proceso con `PTRACE_KILL` o lo libera con `PTRACE_DETACH`.

Ejemplo: launch

A modo de ejemplo, consideremos un programa, `launch.c`, que permite poner a ejecutar otro programa.

launch.c - main()

```
/* Fork en dos procesos */
child = fork();
if (child == -1) { perror("ERROR fork"); return 1; }
if (child == 0) {
    /* Solo se ejecuta en el hijo */
    execvp(argv[1], argv + 1);
    /* Si vuelve de exec() hubo un error */
    perror("ERROR child exec(...)"); exit(1);
} else {
    /* Solo se ejecuta en el padre */
    while(1) {
        if (wait(&status) < 0) { perror("wait"); break; }
        if (WIFEXITED(status)) break; /* Proceso terminado */
    }
}
```

Ejemplo: launch + ptrace()

launch.c + ptrace

```
/* Fork en dos procesos */
child = fork();
if (child == -1) { perror("ERROR fork"); return 1; }
if (child == 0) {
    /* Solo se ejecuta en el hijo */
    if (ptrace(PTRACE_TRACEME, 0, NULL, NULL)) {
        perror("ERROR child ptrace(PTRACE_TRACEME, ...)"); exit(1);
    }
    execvp(argv[1], argv + 1);
    /* Si vuelve de exec() hubo un error */
    perror("ERROR child exec(...)"); exit(1);
} else {
    /* Solo se ejecuta en el padre */
    while(1) {
        if (wait(&status) < 0) { perror("wait"); break; }
        if (WIFEXITED(status)) break; /* Proceso terminado */
        ptrace(PTRACE_SYSCALL, child, NULL, NULL); /* Continúa */
    }
    ptrace(PTRACE_DETACH, child, NULL, NULL); /* Liberamos al hijo */
}
```

ptrace(): Modificando el estado del proceso hijo

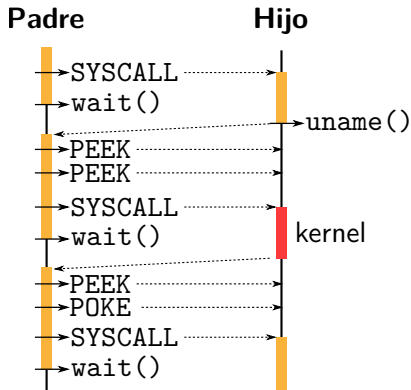
ptrace() permite acceder a la memoria del proceso hijo.

- PTRACE_PEEKDATA y PTRACE_POKEDATA: leer (PEEK) o escribir (POKE) cualquier dirección de memoria en el proceso hijo.
- PTRACE_PEEKUSER, PTRACE_POKEUSER: leer o escribir la memoria de usuario que el sistema guarda al iniciar la *syscall* (registros y estado del proceso).

Ejemplos

- Obtener el número de *syscall* llamada:
`int sysno = ptrace(PTRACE_PEEKUSER, child, 4 * ORIG_EAX, NULL);`
- Leer la dirección *addr* (memoria del proceso hijo):
`unsigned int valor = ptrace(PTRACE_PEEKDATA, child, addr, NULL);`
- Escribir otro valor en la dirección *addr*:
`ptrace(PTRACE_POKEDATA, child, addr, valor + 1);`

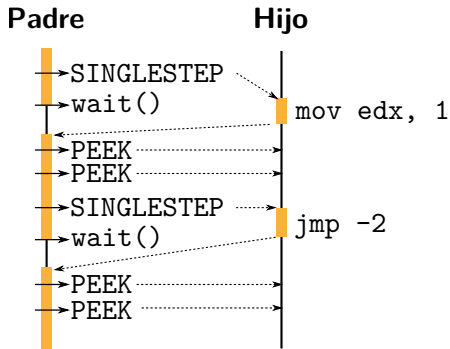
ptrace(): Esquema de uso - Obteniendo datos



Mientras el proceso hijo está detenido, se pueden obtener y modificar datos con

- `PTRACE_PEEKDATA`,
- `PTRACE_POKEDATA`,
- `PTRACE_PEEKUSER` y
- `PTRACE_POKEUSER`.

ptrace(): Esquema de uso - *Debugger*



Un debugger puede usar `PTRACE_SINGLESTEP` para ejecutar paso a paso cada instrucción.