Taller de *syscalls* y señales

Sistemas Operativos

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

23 de Agosto de 2018

Segundo cuatrimestre de 2018

Clase de hoy

- ¿Como interactuamos con el SO?
- ¿Como interactuamos con un proceso?
- Ingeniería inversa con strace 😎
- El detras de bambalinas de strace: ptrace
- Presentación del taller de syscalls 👉

¿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]).

Syscalls

- 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? Veamos un ejemplo.

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**.

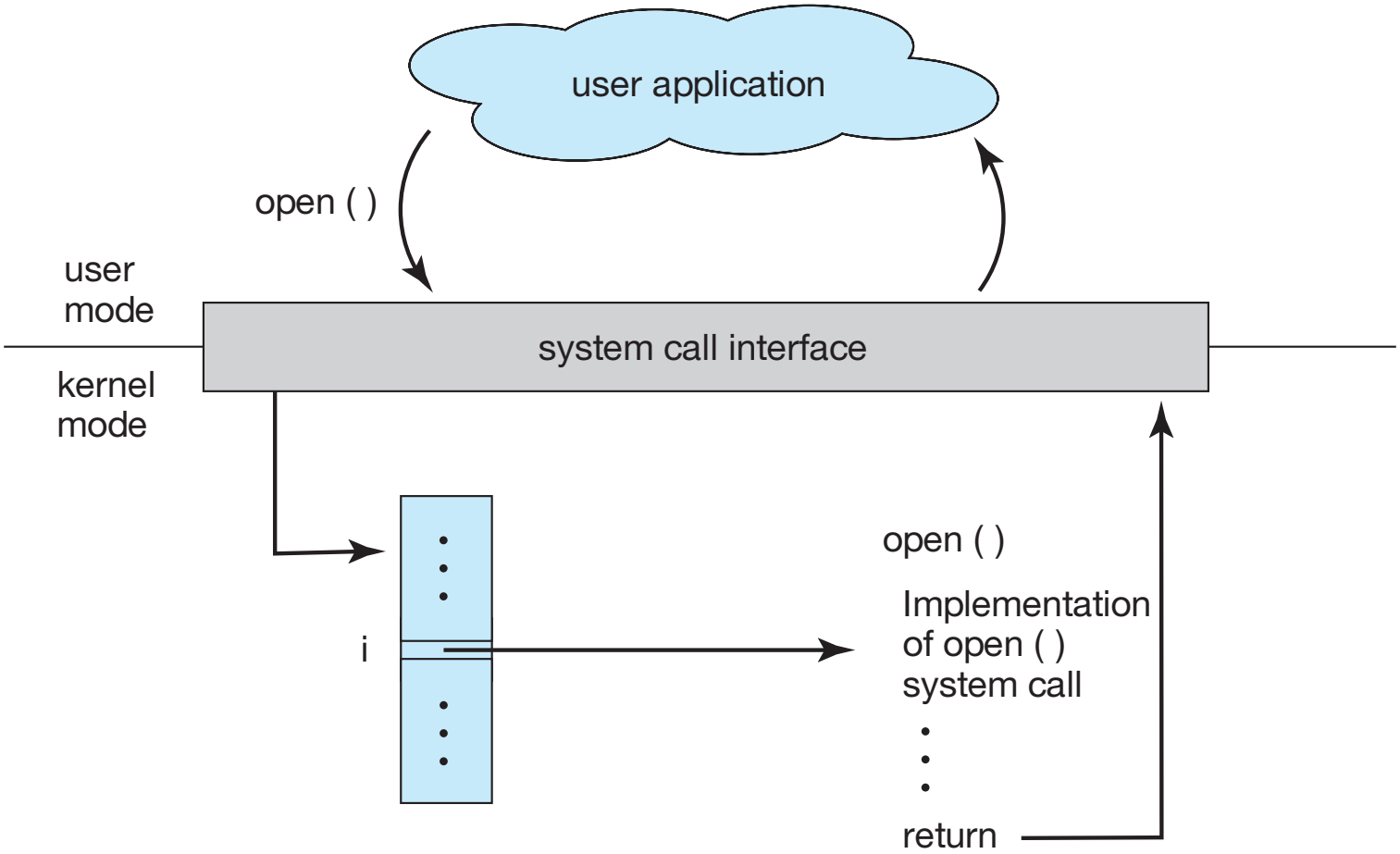
El ejemplo anterior, pero ahora en C:

hello.c

```
#include <unistd.h>

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

Ejemplo de invocación a *syscall*



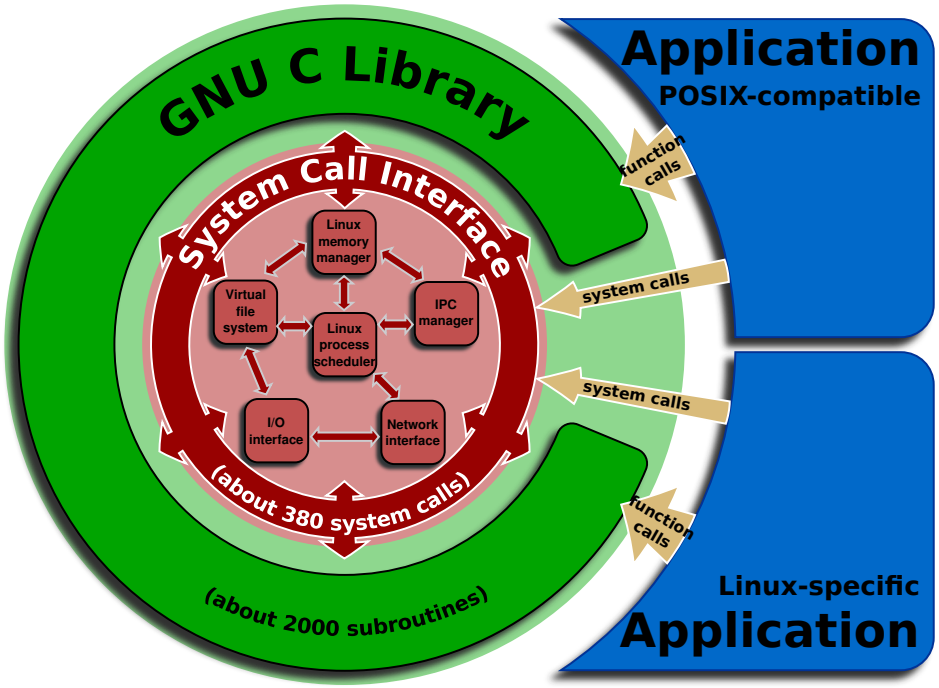
Invocación de la *syscall* `open()` desde una aplicación de usuario.

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

Syscalls en Linux

- La biblioteca estándar de C incluye funciones que no son *syscalls*, pero las utilizan para funcionar. Por ejemplo, `printf()` invoca a la *syscall* `write()`.
- 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).

El espacio correspondiente a la librería de C como al de aplicaciones se encuentran dentro del modo usuario. Por otro lado, aquello en color rojo corresponde al nivel kernel. Las aplicaciones pueden tener llamadas a funciones de la librería C o directamente a syscalls del sistema.

Algunos ejemplos de la API - Creación y control de procesos

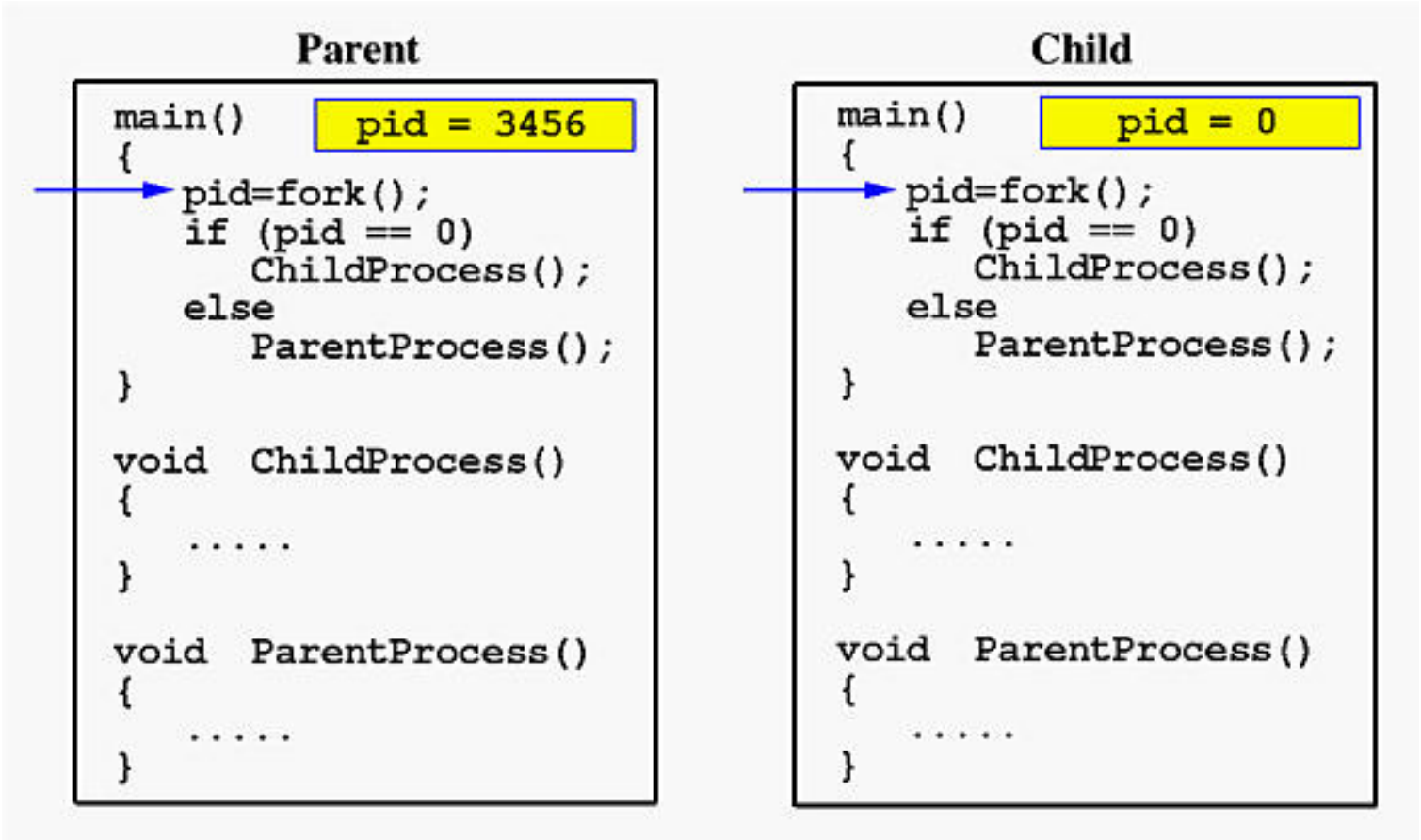
- `pid_t fork(void)`: Crea un nuevo proceso.
  En el caso del creador (padre) se retorna el process id del hijo. En caso del hijo, retorna 0.
- `pid_t vfork(void)`: Crea un hijo sin copiar la memoria del padre, el hijo tiene que hacer exec.
- `int execve(const char *filename, char *const argv[], char *const envp[])`: Sustituye la imagen de memoria del programa por la del programa ubicado en filename.

Algunos ejemplos de la API - Creación y control de procesos

- `pid_t wait(int *status)`: Bloquea al padre hasta que el hijo termine (si no se indica ningún status) o hasta que el hijo alcance el estado indicado.
- `pid_t waitpid(pid_t pid, int *status, int options)`: Igual a wait pero espera al proceso correspondiente al pid indicado.
- `void exit(int status)`: Finaliza el proceso actual.
- `int clone(...)`: Crea un nuevo proceso. El hijo comparte parte del contexto con el padre. Es usado en la implementación de threads.

Creación de procesos utilizando *fork*

El siguiente programa crea un proceso nuevo. Luego, implementa funciones distintas para el creador del proceso y el hijo.



Creación de procesos (fork)

```
1  int main(void) {
2    int foo = 0;
3    pid_t pid = fork();
4    if (pid == -1) exit(EXIT_FAILURE);
5    else if (pid == 0) {
6       printf("%d: Hello world\n", getpid());
7       foo = 1;
8    }
9    else {
10      printf("%d: %d created\n", getpid(), pid);
11      int s; (void)waitpid(pid, &s, 0);
12      printf("%d: %d finished(%d)\n", getpid(), pid, s);
13   }
14   printf("%d: foo(%p)= %d\n", getpid(), &foo, foo);
15   exit(EXIT_SUCCESS);
16 }
```

Creación de procesos (fork)

- Ejemplos de ejecuciones posibles

```
$ ./main
3724: 3725 created
3725: Hello world
3725: foo(0x7fff5431fb6c)= 1
3724: 3725 finished(0)
3724: foo(0x7fff5431fb6c)= 0

$ ./main
3815: Hello world
3815: foo(0x7fff58c3eb6c)= 1
3814: 3815 created
3814: 3815 finished(0)
3814: foo(0x7fff58c3eb6c)= 0
```

Algunos ejemplos la API - Manejo de archivos

- `int open(const char *pathname, int flags)`: Creación y apertura de archivos.
- `ssize_t read(int fd, void *buf, size_t count)`: Lectura de archivos.
- `ssize_t write(int fd, const void *buf, size_t count)`: Escritura de archivos.
- `off_t lseek(int fd, off_t offset, int whence)`: Actualiza la posición actual en el archivo. Se pueden dar los siguientes casos:
  - $whence = SEEK\_SET \rightarrow comienzo + offset$
  - $whence = SEEK\_CUR \rightarrow actual + offset$
  - $whence = SEEK\_END \rightarrow fin + offset$

Señales

- 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 desde la terminal una señal a un proceso con el *comando* `kill`. Un proceso puede enviar una señal a otro mediante la *syscall* `kill()`.

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
```

Algunas opciones útiles:

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

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 `./bin/echo` 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 SO!\n", 9)               = 9
exit_group(0)                           = ?
```

¿Qué es todo esto?

Las *syscalls* de `hello.c`

Llamadas referentes al manejo de memoria

```
brk(0)                                  = 0x831f000
brk(0x831fcb0)                          = 0x831fcb0
brk(0x8340cb0)                          = 0x8340cb0
brk(0x8341000)                          = 0x8341000
```

- `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
...
```