Syscalls y señales

Sistemas Operativos

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

23 de marzo de 2023

Clase de hoy

- ¿Como interactuamos con el SO?
- ¿Qué son las señales y como utilizarlas?
- Ingeniería inversa con strace 😎

¿Cómo interactuamos con el SO?- Recordemos

- Como **usuarios**: programas o utilidades de sistema.
  Por ejemplo: `ls`, `time`, `mv`, `who`, etc.

[1] Portable Operating System Interface [for UNIX]

¿Cómo interactuamos con el SO?- Recordemos

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

[1] Portable Operating System Interface [for UNIX]

¿Cómo interactuamos con el SO?- Recordemos

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

[1] Portable Operating System Interface [for UNIX]

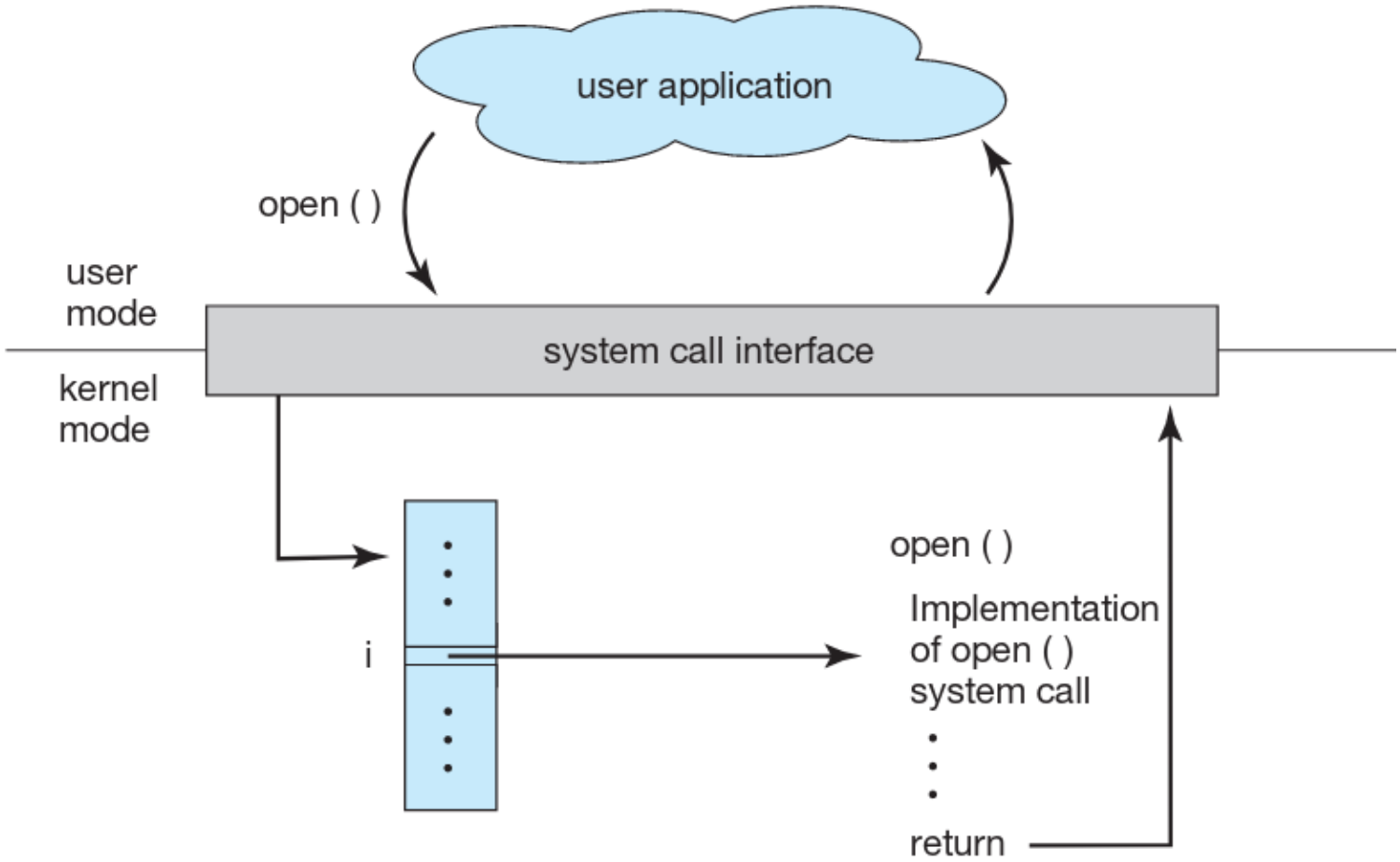
¿Cómo interactuamos con el SO?- Recordemos

- Como **usuarios**: programas o utilidades de sistema.
  Por ejemplo: `ls`, `time`, `mv`, `who`, 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**[1].

[1] Portable Operating System Interface [for UNIX]

Ejemplo de invocación a *syscall*- Recordemos



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

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

Syscalls en Linux

- Las syscalls las utilizamos mediante la biblioteca estandar de C: `unistd.h`

Syscalls en Linux

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

Syscalls en Linux

- ★ Las syscalls las utilizamos mediante la biblioteca estandar de C: `unistd.h`
- ★ La biblioteca estándar de C incluye funciones que no son *syscalls*, pero las utilizan para funcionar. Por ejemplo, `printf()` invoca a la *syscall* `write()`.
  - Puede verse una lista de todas ellas usando `man syscalls`.

Syscalls en Linux

- ★ Las syscalls las utilizamos mediante la biblioteca estandar de C: `unistd.h`
- ★ La biblioteca estándar de C incluye funciones que no son *syscalls*, pero las utilizan para funcionar. Por ejemplo, `printf()` invoca a la *syscall* `write()`.
  - Puede verse una lista de todas ellas usando `man syscalls`.
  - Las syscalls están en la hoja de manual 2. `man man`

Algunos ejemplos de la API - Creación y control de procesos

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.

API - S.O.

¡Manos a la obra!

Veamos un ejemplo de funcionamiento.

¡Manos a la obra!

Veamos un ejemplo de funcionamiento.
Suponga que Juan tiene 2 hijos, Jorge y Julieta. A su vez Julieta tiene una hija, Jennifer. Pero suponga que recién cuando nació Jennifer, Juan tuvo a Jorge. Suponga que quiere procesos que delimiten la vida de cada uno.

¡Manos a la obra!

Veamos un ejemplo de funcionamiento.
Suponga que Juan tiene 2 hijos, Jorge y Julieta. A su vez Julieta tiene una hija, Jennifer. Pero suponga que recién cuando nació Jennifer, Juan tuvo a Jorge. Suponga que quiere procesos que delimiten la vida de cada uno. El siguiente código (Fork2)

¡Manos a la obra!

Veamos un ejemplo de funcionamiento.
Suponga que Juan tiene 2 hijos, Jorge y Julieta. A su vez Julieta tiene una hija, Jennifer. Pero suponga que recién cuando nació Jennifer, Juan tuvo a Jorge. Suponga que quiere procesos que delimiten la vida de cada uno. El siguiente código (Fork2)¿En qué orden se imprimirá en pantalla cada mensaje?

¡Manos a la obra!

Veamos un ejemplo de funcionamiento.
Suponga que Juan tiene 2 hijos, Jorge y Julieta. A su vez Julieta tiene una hija, Jennifer. Pero suponga que recién cuando nació Jennifer, Juan tuvo a Jorge. Suponga que quiere procesos que delimiten la vida de cada uno. El siguiente código (Fork2)¿En qué orden se imprimirá en pantalla cada mensaje?
¿Cómo podría hacer para que se lancen los procesos en el momento adecuado y sin problemas? Veamos...

Algunos ejemplos de la API - Creación y control de procesos - Parte II

- `pid_t wait(int *status)`: Bloquea al padre hasta que el hijo cambie de estado (si no se indica ningún status). El cambio de estado mas comun a utilizar es cuando el hijo termina su ejecución
- `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.

¡Manos a la obra!

Podríamos usar `wait`

¡Manos a la obra!

Podríamos usar `wait`

```
int status;
// Si termino con errores
     if(wait(&status) < 0){perror("wait");exit(-1);}
```

Aclaraciones de las syscalls de wait

Aclaraciones

La syscall `wait` ademas permite liberar los recursos asociados al hijo. Si no se hace esta operacion, el proceso hijo continua en un estado `zombie`.

Aclaraciones de las syscalls de wait

Aclaraciones

La syscall `wait` ademas permite liberar los recursos asociados al hijo. Si no se hace esta operacion, el proceso hijo continua en un estado `zombie`. Que este en ese estado significa que la entrada del proceso en la tabla de procesos aun permanece a pesar de haber terminado su ejecución.

Otros 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 - Recordatorio

- Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.

Señales - Recordatorio

- ★ Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
- Se utiliza `signal.h`. Veremos bastante en el taller.

Señales - Recordatorio

- ★ Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
- Se utiliza `signal.h`. Veremos bastante en el taller.
- Ejemplo: `SIGINT`, `SIGKILL`, `SIGSEGV`.

Señales - Recordatorio

- ★ Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
  - Se utiliza `signal.h`. Veremos bastante en el taller.
  - Ejemplo: `SIGINT`, `SIGKILL`, `SIGSEGV`.
  - 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()`.

Señales - Recordatorio

- ★ Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
- Se utiliza `signal.h`. Veremos bastante en el taller.
- Ejemplo: `SIGINT`, `SIGKILL`, `SIGSEGV`.
- 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()`.
- Veamos `man kill`

Señales - Recordatorio

- ★ Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
  - Se utiliza `signal.h`. Veremos bastante en el taller.
  - Ejemplo: `SIGINT`, `SIGKILL`, `SIGSEGV`.
  - 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()`.
  - Veamos `man kill`
  - `kill -L`

Señales - Recordatorio

- ★ Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
- Se utiliza `signal.h`. Veremos bastante en el taller.
- Ejemplo: `SIGINT`, `SIGKILL`, `SIGSEGV`.
- 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()`.
- Veamos `man kill`
- `kill -L`
- Veamos un ejemplo de código. `signal.cpp`

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.

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
...
```

Clase de hoy - ¿Qué vimos?

- ¿Como interactuamos con el SO? - fork, wait
- señales, signal, kill
- Ingeniería inversa con strace 😎

Clase de hoy - ¿Como seguimos?

- Pueden hacer toda la primera parte de la guía 1.
- Vamos a hacer el taller de syscalls (28/03)
- Comienzo de IPC (Teórica y Práctica)

Eso es todo

¡Eso es todo amigos! ¡Gracias!

¡Gracias!