Syscalls y señales

Sistemas Operativos
DC - FCEN - UBA

27 de marzo de 2025

Clase de hoy

- ¿Cómo interactuamos con el SO?
- ¿Qué son las señales y cómo 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.
- 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]

Man

Recordemos

- Prácticamente todos los mecanismos están documentados en el manual de Linux, **man**. Podemos verlo con `man man`.
- Está dividido en varias páginas, y cada una corresponde a una clasificación específica.
- Por ejemplo: La página 1 corresponde a los comandos de la terminal, la 2 a las *syscalls*, etc.
- Ejemplo de uso: `man 2 kill`.

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 en x86

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
```

Un primer ejemplo en ARM

tinyhelloARM.asm

```
.data
/* char msg[10] = "Hola SO!" */
msg dbc "Hola SO!", 0
msglen = . - msg

.text
.global main                    /* Entrypoint */

print:
  mov r7, #0x900004             /* Call write() */
  mov r0, #1
  svc   $0                      /* invoke syscall */

quit:
  mov r7, #0x900001             /* Call sys_exit() */
  svc  $0
```

Usando *wrapper functions* en C

- Claramente, el código anterior no es portable.
- Además, realizar una syscall de esta forma requiere programar en lenguaje ensamblador.

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;
}
```

- Las wrapper functions permiten interactuar con el sistema con mayor **portabilidad** y **sencillez**.

Syscalls en Linux

- Las syscalls las utilizamos mediante la biblioteca estándar de C: `unistd.h`
- La biblioteca estándar de C incluye funciones que no son syscalls, pero las incluyen e invocan en su código. 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`.

Procesos

Recordemos

¿Qué es un programa?

Un conjunto de instrucciones diseñadas para realizar una tarea, almacenadas en la memoria.

¿Qué es un proceso?

Un proceso es una instancia de un programa que está en ejecución, incluyendo su estado y recursos asignados.

Múltiples procesos

Varios procesos pueden ejecutar el mismo programa simultáneamente.

Process ID

- Cada proceso se le otorga un único identificador, el número **PID** (*Process ID*).
- Podemos preguntar al SO qué número de PID tiene el proceso en ejecución usando la *syscall* `getpid()`.

`pid.c`

```c
#include <unistd.h>

int main(int argc, char* argv[]) {
  pid_t pid = getpid(); // pid_t es un renombre de int
  printf("Mi PID es %d\n", pid);
  return 0;
}
```

Creación de procesos

- `pid_t fork(void)`: Crea un nuevo proceso, que es un clon del primero.

Código en C con `fork`

Crea un nuevo proceso que es un clon del primero.

Proceso A

```c
int main()
{
    printf("Hola SO!\n");
    fork();
    printf("Nos vemos!\n");
    return 0;
}
```

Código en C con `fork`

Crea un nuevo proceso que es un clon del primero.

Proceso A

```c
int main()
{
    printf("Hola SO!\n");
    fork();
    printf("Nos vemos!\n");
    return 0;
}
```

Proceso B

```c
int main()
{
    printf("Hola SO!\n");
    fork();
    printf("Nos vemos!\n");
    return 0;
}
```

- El proceso A se denomina Padre y el proceso B Hijo. Cada uno tiene su propio PID.
- A partir de este punto sus ejecuciones son independientes y el orden de ejecución lo decide el scheduler.
- **¡Importante!** Cada proceso corre en espacios de memoria separados.

Creación de procesos utilizando `fork()`

¡Importantísimo!

NO SE COMPARTE MEMORIA

Creación de procesos utilizando `fork()`

- ¿Cómo podemos hacer que los procesos creados hagan cosas distintas?
- ¿Cómo sabe cada proceso si es padre o hijo?

El valor que devuelve `fork()` se ve diferente en el padre y en el hijo.

- En el proceso padre, `fork()` devuelve el valor del PID del hijo recién creado, pero en el proceso hijo esa variable queda con valor 0.
- Ojo: 0 NO es el PID del hijo.
- Para el padre, esta es la única forma de conseguir el PID del hijo.

De esta forma, podemos asignar distintos segmentos de código tanto al padre como al hijo.

Creación de procesos utilizando `fork()`

Veamos un demo...

Código en C con `fork`

Crea un nuevo proceso que es un clon del primero.

Proceso padre

```
int main()
{
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        Subrutina_proceso_hijo();
    } else {
        Subrutina_proceso_padre();
    }
    exit(EXIT_SUCESS);
}
```

pidOrZero: 1145

Proceso hijo

```
int main()
{
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        Subrutina_proceso_hijo();
    } else {
        Subrutina_proceso_padre();
    }
    exit(EXIT_SUCESS);
}
```

pidOrZero: 0

Código en C con `fork`

Crea un nuevo proceso que es un clon del primero.

Proceso padre

```
int main()
{
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        Subrutina_proceso_hijo();
    } else {
        Subrutina_proceso_padre();
    }
    exit(EXIT_SUCESS);
}
```

pidOrZero: 1145

Proceso hijo

```
int main()
{
    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        Subrutina_proceso_hijo();
    } else {
        Subrutina_proceso_padre();
    }
    exit(EXIT_SUCESS);
}
```

pidOrZero: 0

- **¡Importante!** No tenemos control sobre el orden en que se ejecutan los procesos.
- Sólo a modo de ilustración, se muestra un ejemplo donde cada proceso está ejecutando su respectiva función al mismo tiempo.

Identificación de procesos

- `pid_t getppid(void)`: Obtener el PID del padre del proceso actual.
- `pid_t getpid(void)`: Conseguir el PID del proceso actual.

Creación de procesos

fork()

- ¿Qué sucede si el proceso padre termina su ejecución antes que el hijo?
- En ese caso, se dice que el proceso hijo queda **huérfano**.
- Otro proceso se hace cargo de este proceso huérfano y pasa a ser su padre.
- Por lo general, el proceso `init` es el encargado.
- Sin embargo, en Linux existen los procesos **"subreaper"**, que son procesos que se pueden autodeclarar como padres de procesos huérfanos que sean descendientes suyos.

Creación de procesos

fork()

Veamos un ejemplo implementado...

Aclaraciones de Fork - Clone

- Cuando se realiza el llamado a `fork()`, por debajo se está llamando a la *syscall* `clone`.
- Es un mecanismo para realizar la creación de procesos. Podemos determinar sobre qué contextos de ejecución comparten padre e hijo.
- Por ejemplo, se puede controlar si se quiere que compartan el espacio virtual, el stack, dónde arranca la ejecución, entre otras cosas.
- Tanto procesos como threads utilizan esta syscall con sus parámetros correspondientes.
- Para más detalle: `man clone`.

¡Manos a la obra!

Supongamos que Juan tiene 2 hijos, Jorge y Julieta. A su vez Julieta tiene una hija, Jennifer. Pero supongamos que luego de que nació Jennifer, Juan tuvo a Jorge. Se requiere la creación y ejecución procesos que emulen la vida de cada uno.

Pare el siguiente código...

- ¿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...

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 más común 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.

¡Manos a la obra!

Podríamos usar `wait`.

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

Aclaraciones de wait

- La syscall `wait()` permite liberar los recursos asociados al hijo.
- Si no se hace esta operación, cuando el proceso hijo muere, continúa en un estado zombie.
- Esto significa que la entrada del proceso en la tabla de procesos permanece a pesar de haber terminado su ejecución.
- Sin embargo, si el proceso padre termina, esta operación se hace automáticamente.

Volviendo al tema de la memoria

Dijimos que los procesos no comparten memoria.
Veamos el siguiente ejemplo en código...

Volviendo al tema de la memoria

- ¿Cómo puede suceder que el padre y el hijo guarden diferente información en la misma dirección?
- El hijo es una copia de la memoria del padre, así que los punteros referencian la misma dirección virtual...
- ¡Pero no comparten memoria! Las direcciones virtuales de padre e hijo están mapeadas a distinta memoria física.
- ¿No es muy caro hacer copias de toda la memoria cuando se forkea?

Copy on write

- El sistema operativo sólo hace copias *lazy*.
- Ambos compartirán las mismas páginas físicas hasta que alguna de ellas cambia el contenido.
- En ese momento se asigna una página física distinta para el proceso que modifica la memoria.
- Es decir, sólo se comparten páginas en modo lectura.
- Este mecanismo se llama copy on write.

Control de procesos

Familia *exec*

La familia de syscalls `exec` reemplazan la imagen del proceso actual con una nueva.
Una de las más utilizadas es `execve`.

- `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`.

Control de procesos

Familia *exec*

Las funciones son: `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`, `execvpe`.
Cada letra luego del prefijo `exec`, nos indica un significado particular de lo que hace cada función:

- `l`: Indica que la función es variádica (aridad indefinida). Toma una secuencia de argumentos que se le pasa a la imagen a reemplazar.
  Es útil cuando sabemos de antemano la cantidad de parámetros a utilizar.
  El último parámetro tiene que ser `NULL`.
  ```
  $ execl [pathname] [arg1] [arg2] ... [argN] [NULL]
  ```
- `v`: Indica que la función toma un array de punteros a char como los parámetros a usar.
  ```
  $ execv [pathname] [array args]
  ```

Control de procesos

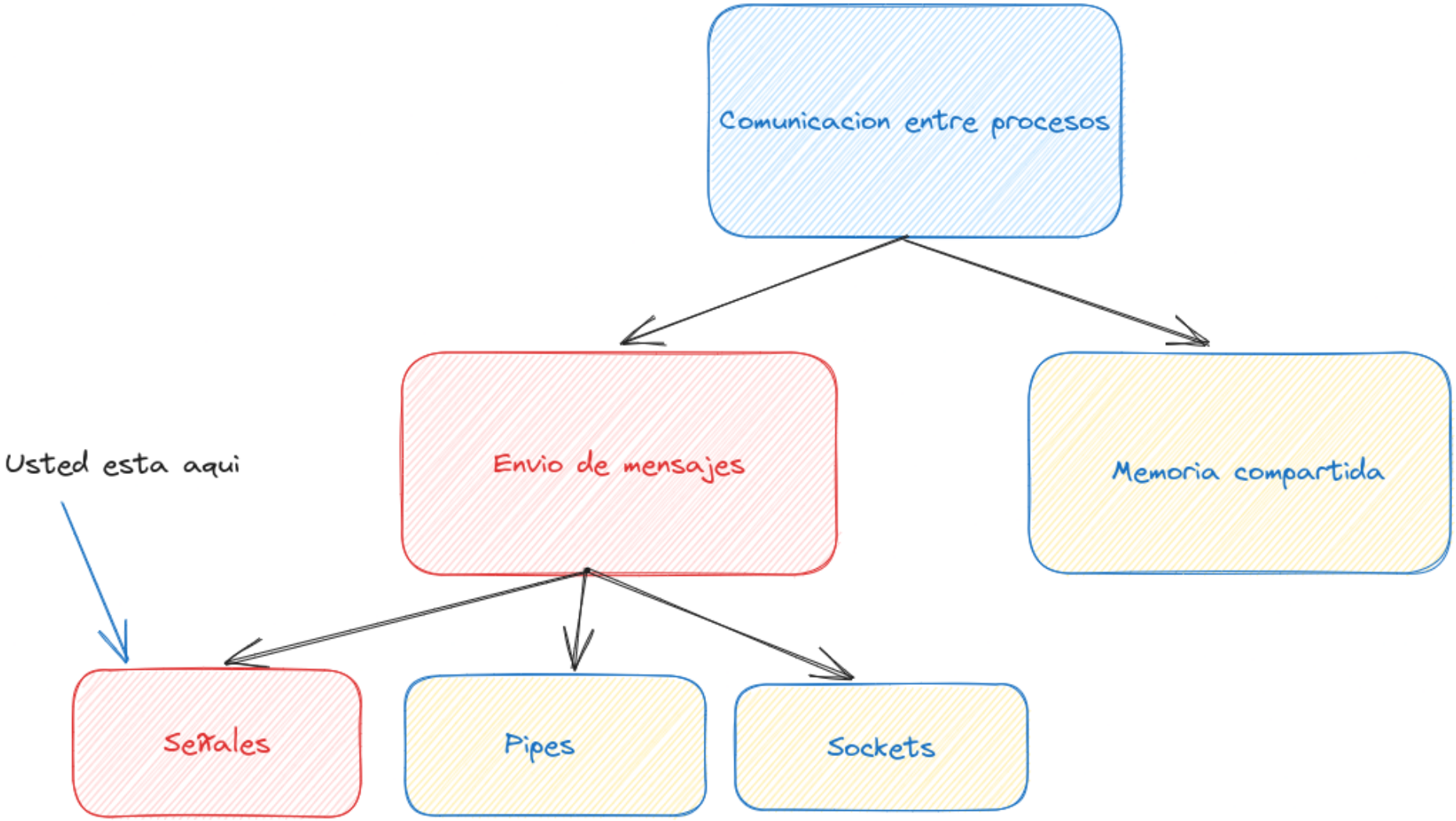
Familia *exec*

Las funciones son: `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`, `execvpe`.
Cada letra luego del prefijo `exec`, nos indica un significado particular de lo que hace cada función:

- `e`: Indica que se le pueden pasar variables de entorno, tanto de forma variádica como usando un array.
  `$ execve [pathname] [array args] [array env_var]`
- `p`: Indica que el nombre pasado en `filename`, por defecto lo busque en el pathname que indica la variable de entorno `PATH`.
  Por ejemplo, si utilizamos `execvp('ls', ['-a'])`, buscará el nombre del comando `ls` según el contenido de la variable `PATH`.
  `$ execvp [filename] [array args]`

Momento para preguntas

Comunicación entre procesos



Señales

Recordemos

- Las **señales** son un mecanismo que incorporan los sistemas operativos basados en POSIX, que permiten notificar a un proceso la ocurrencia de un evento.
- Para implementarlos en C, se utiliza `signal.h`.
- Cada señal es un número, pero comúnmente se les identifica mediante macros.
- Ejemplo: `SIGINT` (señal 2), `SIGKILL` (señal 9), `SIGSEGV` (señal 11).
- 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

Recordemos

- Es posible redefinir el comportamiento de algunas señales usando funciones void sin parámetros, comúnmente llamadas handlers.
- Para esto se utiliza la función en C `signal()`, que a su vez usa una syscall con el mismo nombre.
- Como primer parámetro, indicamos la señal a la que le queremos cambiar su comportamiento.
- Como segundo parámetro, le pasamos el nombre de una función, el `handler`.
- **¡Importante!** Algunas señales no se pueden *handlear*, como `SIGKILL`.

Veamos un ejemplo de código. `signal.cpp`

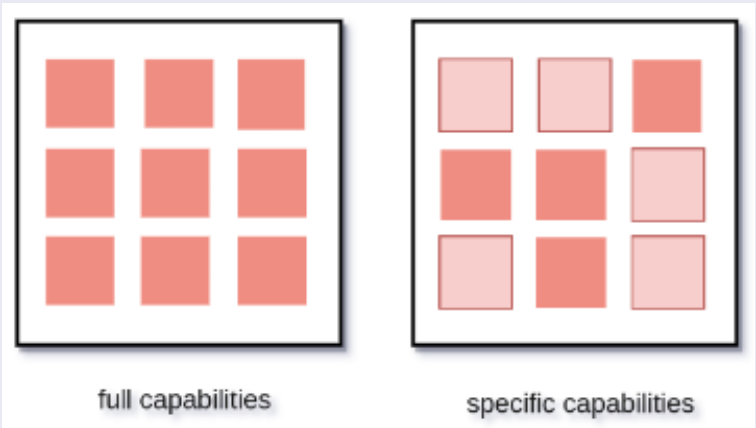
Señales - SIGCHLD

- Como dijimos previamente, `wait()` espera a que un hijo cambie de estado.
- Pero, ¿cómo funciona este mecanismo de espera?
- Cuando un proceso hijo termina su ejecución, envía la señal `SIGCHLD` y el padre la recibe.
- No solamente se envía en este caso, sino también cuando ocurre un cambio de estado: que un hijo terminó, que el hijo fue frenado por una señal, o el hijo fue reanudado por una señal.

Protección

Capabilities

- En Linux se distinguen procesos con permisos privilegiados (`root`) y no privilegiados.
- Un proceso no `root` no puede enviar señales a procesos `root`
- Sin embargo, en Linux es posible permitir este envío, ya que divide los privilegios tradicionalmente asociados con `root` en distintas unidades llamadas **capabilities**, que pueden ser habilitadas o deshabilitadas.
- Se puede usar `setcap` para cambiar las **capabilities**, `CAP_KILL` se denomina a la capacidad para enviar señales a cualquier proceso,



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`: Muestra también la traza de los procesos hijos.

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*, de haberlos) 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?

strace y hello en 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) = 0
    xb8001000
close(3)                                = 0
...
```

Momento para preguntas

Resumen

Hoy vimos...

- ¿Cómo interactuamos con el SO? → Syscalls y wrapper functions.
  - Creación de procesos: `fork()`
  - Identificación de procesos: `getpid()` y `getppid()`
  - Control de procesos: `wait()`, `waitpid()`, `exit()` y la familia `exec`.
- Uso de señales con `signal()` y `kill()`.
- Ingeniería inversa con `strace`.

Cómo seguimos...

- Pueden hacer toda la primera parte de la guía 1.
- Pueden comenzar el taller de syscalls.
- Siguientes clases → Comunicación Inter-procesos ó IPC.