Taller de *syscalls* y señales

Sistemas Operativos

25 de agosto de 2022

Segundo cuatrimestre - 2022

1. Ejercicios

1.1. Ejercicio 1

Utilizar el comando *strace* para analizar el comportamiento del programa correspondiente al archivo binario *hai* cuyo uso es el siguiente:

```
./hai [PROGRAMA]
```

donde PROGRAMA es la ruta de otro programa con sus parametros de entrada[1].
Luego, responder las siguientes preguntas:

- ¿Cuántos procesos se lanzan y qué comportamiento se puede observar de cada uno?
- ¿Utilizan los procesos algúna forma de IPC?¿Cuál es y para qué se usa en este caso?
- ¿Qué puede inducir del programa a partir de las syscalls observadas?

1.2. Ejericio 2

Escribir un programa en C que presente el *mismo* comportamiento que el programa analizado en el punto anterior. Es decir, que se observe la misma salida al ser ejecutado por un usuario con los mismos argumentos y que las respuestas para el punto anterior sean las mismas.[2]

1.3. Ejercicio 3

Completar el programa *antikill.c* en C que ejecute un comando pasado por parámetro, PERO, en caso de que este comando envíe la señal SIGKILL a otro proceso, *antikill* debe evitar que se ejecute esta señal y debe terminar el proceso que la intentó enviar. Si ocurre esto, *antikill* debe, además, imprimir un mensaje indicando que se impidió el envío de la señal y también el pid del proceso detenido.

Por ejemplo:

```
$ ./antikill kill -9 28988
El proceso 28976 ha intentado envíar la señal SIGKILL y por eso ha sido detenido.
```

[1] por ejemplo: ./hai echo “imprimir este texto” , ./hai ls, etc.
[2] Sugerencia: Usar el programa `hai.c` como base.

2. Notas útiles para la resolución del taller

2.1. Preeliminares

Es conveniente para la resolución del taller repasar la clase práctica correspondiente al mismo, así como poder entender el código de los programas provistos junto con la misma y haberlos compilado y ejecutado por separado.

2.2. `strace`

`strace` es una herramienta que permite generar una traza legible de las syscalls usadas por un programa dado. Sintaxis:

```
$ strace [opciones] comando [argumentos]
```

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.

2.3. `ptrace`

La *syscall* `ptrace()` permite observar y controlar un proceso hijo. En particular permite obtener una traza del proceso, desde el punto de vista del sistema operativo, al permitir detener el proceso hijo antes y después de realizar un *syscall*.

Su sintaxis es la siguiente:

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

El parámetro `request` permite elegir qué se desea hacer. Dependiendo de este parámetro, algunos de los siguientes parámetros de la *syscall* no se utilizan. Por ejemplo, `PTRACE_TRACEME` no utiliza ninguno de los siguientes tres parámetros, y `PTRACE_POKEDATA` usa todos ellos. `request` puede ser alguno de los siguientes valores:

- `PTRACE_TRACEME`
- `PTRACE_SYSCALL`, `PTRACE_SINGLESTEP`,
- `PTRACE_PEEKDATA`, `PTRACE_POKEDATA`,
- ...y más[3].

El parámetro `pid` es el *process id* del proceso hijo.

2.3.1. `PTRACE_SYSCALL`

Cada vez que se genera un evento en el proceso hijo, el mismo es detenido. Para continuar la ejecución del proceso hijo se debe hacer una llamada a `ptrace` desde el padre. Esta llamada puede hacerse a `PTRACE_SYSCALL`, `PTRACE_CONT`, o `PTRACE_SINGLESTEP`, dependiendo de qué tipo de evento es el próximo evento que se desea atrapar. Para detenerse por el siguiente ingreso o egreso de una *syscall* se debe usar el valor `PTRACE_SYSCALL`.

[3] Ver `man 2 ptrace`

2.3.2. PTRACE_KILL

Una forma de terminar el proceso hijo que está siendo monitoreado es enviarle una señal de KILL a través de `ptrace`. Para ello se debe usar el valor de `request` `PTRACE_KILL` e indicar el *pid* del hijo que se desea terminar.

2.3.3. PTRACE_PEEKUSER y PTRACE_PEEKDATA

Los `request` `PTRACE_PEEKUSER` y `PTRACE_PEEKDATA` le permiten al proceso padre obtener información sobre la memoria del proceso hijo.

Con `PTRACE_PEEKDATA` se puede leer *cualquier* dirección del espacio de direcciones del proceso hijo. Pero, aún así, eso no es suficiente, dado que además de los datos visibles desde el proceso hijo, hay más información relativa a este proceso.

Para ello, `PTRACE_PEEKUSER` nos permite acceder al espacio de memoria *del kernel* que guarda información sobre el proceso hijo. Esta información no es directamente visible desde el proceso hijo, es decir, no está en ninguna dirección de memoria del mismo.

De esta información del kernel, un valor que nos interesa es qué valor tenía el registro RAX al momento de hacer la llamada al sistema, dado que ese valor determina qué *syscall* se está llamando. En el archivo `<sys/reg.h>` se encuentran definidas algunas constantes útiles, como `ORIG_RAX`. Dentro de este espacio, el valor de RAX al generarse la llamada al sistema se encuentra en la dirección `8 * ORIG_RAX`.

Para hacer una llamada a `PTRACE_PEEKUSER` o a `PTRACE_PEEKDATA` la dirección se debe colocar en el parámetro `addr`, pero el parámetro `data` no se utiliza. Por el contrario, siempre se lee una *palabra* (8 bytes en el caso de x86_64) y se devuelven como valor de retorno de la función.

Ejemplo tomado de las slides de la clase:

```
int sysno = ptrace(PTRACE_PEEKUSER, child, 8 * ORIG_RAX, NULL);
```

Al utilizar `PTRACE_SYSCALL`, el proceso se detiene al *entrar* y *salir* de una *syscall*. Para determinar esto, se puede consultar el valor de `ptrace(PTRACE_PEEKUSER, child, 8 * RAX, NULL)`, que devolverá un error (`-ENOSYS`) cuando el proceso se encuentre entrando en la *syscall*.

Para obtener el parámetro de la syscall, se podrá leer del registro RSI, realizando el llamado `signal = ptrace(PTRACE_PEEKUSER, child, 8 * RSI, NULL)`.

En el caso de una arquitectura x86 de 32 bits puede ser que tengan que usar `EAX` y `ORIG_EAX` en vez de `RAX` y `ORIG_RAX` junto al tamaño correcto de los registros (4 bytes).

2.4. *Includes* recomendados

```
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/reg.h>
#include <unistd.h>
#include <syscall.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

2.5. Otros

En los headers `<sys/syscall.h>` se encuentran definidos símbolos para cada una de las *syscalls* del sistema. Por ejemplo, el número de *syscall* de `write` está definido por el símbolo `SYS_write`.