Práctica de BASH: Traza de llamadas del sistema en los procesos con strace.

1. Objetivos

Es hora de aprender a manejar los conceptos vistos para crear scripts por nosotros mismos que automaticen las tareas más comunes y que además nos permita profundizar en aspectos importantes del sistema operativo.

2. Información previa

El proyecto se centra en seguir la traza de las llamadas del sistema utilizadas por los procesos, algo que se utiliza en el estudio y depuración de los programas. Pero, antes de empezar, vamos a revisar algunos conceptos sobre este tema.

El comando **strace** permite monitorizar las **llamadas del sistema** utilizadas en la ejecución de un proceso, así como las **señales del sistema** recibidas. Su funcionamiento se basa en la llamada del sistema ptrace y tiene dos modos de funcionamiento: monitorización de procesos descendientes, y monitorización de procesos en ejecución o modo attach.

Un ejemplo de su utilización básica en el modo de monitorización de procesos descendientes:

```
strace cp p.txt p1.txt
```

Que, asumiendo que existe el archivo `p.txt` indicado en los argumentos (de lo contrario el comando `cp` fallaría) produce en el terminal una salida que termina así:

```
newfstatat(AT_FDCWD, "p1.txt", {st_mode=S_IFREG|0664, st_size=0, ...}, 0) = 0
newfstatat(AT_FDCWD, "p.txt", {st_mode=S_IFREG|0664, st_size=0, ...}, 0) = 0
newfstatat(AT_FDCWD, "p1.txt", {st_mode=S_IFREG|0664, st_size=0, ...}, 0) = 0
openat(AT_FDCWD, "p.txt", O_RDONLY)     = 3
newfstatat(3, "", {st_mode=S_IFREG|0664, st_size=0, ...}, AT_EMPTY_PATH) = 0
openat(AT_FDCWD, "p1.txt", O_WRONLY|O_TRUNC) = 4
newfstatat(4, "", {st_mode=S_IFREG|0664, st_size=0, ...}, AT_EMPTY_PATH) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3247397000
read(3, "", 131072)                     = 0
close(4)                                = 0
```

```
close(3)                                = 0
munmap(0x7f3247397000, 139264)          = 0
lseek(0, 0, SEEK_CUR)                   = -1 ESPIPE (Illegal seek)
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)
```

En el ejemplo, llamamos a `strace` sin opciones. Como argumento le pasamos un programa a monitorizar con sus diferentes opciones y argumentos, en este caso, el comando `cp p.txt p1.tx.`

El programa strace, cuando se ejecuta de este modo, crea el proceso hijo donde lanza el comando cp, haciendo que el proceso hijo sea monitorizable por él mediante llamadas a ptrace.

Veamos algunas opciones importantes. La forma general de llamar a `strace` es:

```
strace [opciones] comando [argumentos del comando]
```

El trabajo de `strace` es monitorizar las llamadas del sistema usadas por el proceso del programa indicado como argumento, mostrando su salida **por la salida de error (fd=2),** salvo que se utilice la opción:

```
-o filename
```

en cuyo caso la salida se realiza en el archivo `filename`.

Si vemos la salida del ejemplo, observaremos que para cada llamada del sistema se muestra la información en una línea, donde está el nombre de la llamada con sus argumentos y al final el valor entero que ha retornado dicha llamada en el proceso monitorizado. Por ejemplo:

```
openat(AT_FDCWD, "p.txt", O_RDONLY)      = 3
```

indica que se invocó la llamada `open`, que intentó abrir el archivo `p.txt` y, tras conseguirlo, devolvió el valor 3, que es el descriptor de archivo.

Si se produjera un error en la llamada, se incorporaría al valor de la salida (en muchas llamadas sería -1) la cadena que representa el tipo de error. Por ejemplo, si ejecutamos el `strace` anterior, pero el archivo `p.txt` no existe, obtendríamos en la traza:

```
newfstatat(AT_FDCWD, "p.txt", 0x7ffd4f44af00, 0) = -1 ENOENT (No such file or directory)
```

Donde la llamada `newfstatat` trata de obtener información sobre los atributos del archivo y al no encontrarlo devuelve -1 y nos dice el tipo de error, guardado `errno`.

En el ejemplo anterior el comando `cp` es iniciado por `strace`, pero `strace` también puede utilizarse para monitorizar **procesos que están ya en ejecución en el sistema, es el modo attach.**

Nota importante. Es posible que al ensayar en tu sistema los siguientes ejemplos obtengas errores del tipo "Operación No Permitida". La razón es que en la configuración de muchos sistemas linux, se usa un módulo de seguridad denominado Yama, que establece una restricción sobre el uso de ptrace. Eso lo podemos comprobar haciendo:

cat /proc/sys/kernel/yama/ptrace_scope

Si la salida es:

1

Significa que ptrace está restringida por defecto a procesos descendientes (modo de los ejemplos anteriores) y no se puede realizar el attach sobre cualquier proceso, sino sólo sobre aquellos que se configuran para ser monitorizados. Después veremos cómo hacerlo.

Si la salida fue 0, no habría restricciones y podrías usar strace con otros procesos cuyo usuario (uid) fuese el mismo que el tuyo.

Si eres administrador del sistema podrías cambiar la configuración del módulo de seguridad Yama en el archivo

/etc/sysctl.d/20-ptrace.conf

Donde encontrarás

kernel.yama.ptrace_scope=1

Para eliminar las restricciones de Yama hay que poner ese atributo a 0 y reiniciar el sistema.

En caso de que no puedas cambiar la configuración por no tener permisos de administración en el sistema hay varias posibilidades.

a) Usar para las pruebas algún programa concreto configurado para eliminar en él las restricciones. Se hace con la llamada del sistema **prctl.**

Por ejemplo:

```
#include <iostream>
#include <unistd.h>
#include <sys/prctl.h>
int main()
{
        prctl(PR_SET_PTRACER,PR_SET_PTRACER_ANY);
while(1){
   sleep(2);
   std::cout << “Ejemplo” << std::endl;
}

}
```

Aquí la llamada prctl, que se utiliza para modificar la configuración de un proceso, aplica la acción PR_SET_PTRACER para establecer qué programa trazador puede realizar el “attach” sobre él una vez está en funcionamiento. El segundo argumento puede ser el pid del proceso admitido, 0 si queremos eliminar el permiso o PR_SET_PTRACER_ANY para que cualquier proceso pueda considerarse como posible trazador.

b) Utilizar un programa “lanzador” que configure el proceso como en el ejemplo anterior y luego lance de un programa (ver código en el aula virtual). Esto se consigue gracias a la llamada del sistema execve que permite sustituir la imagen del programa actual por otro en el proceso actual sin modificar las propiedades del mismo. El siguiente código sería el lanzador.

La forma de utilizarlo sería la siguiente. Primero compilamos el código fuente:

```
g++ l2t.cc -o l2t
```

Luego ya podemos utilizarlo con cualquier programa con argumentos, por ejemplo el comando sleep:

```
./l2t sleep 20 >/dev/null &
```

El comando sleep 20 se lanzaría en segundo plano en un proceso con un PID determinado y podríamos hacer:

```
strace -p PID
```

donde PID es el pid del proceso lanzado con l2t.

En cualquier caso, siempre deben cumplirse las restricciones básicas sobre la coincidencia de los usuarios de ambos procesos.

Nota: Ubuntu como subsistema linux de Windows no tiene restricciones de Yama. Las instalaciones estándar de Ubuntu sí la tienen (ptrace_scope es 1)

En los siguientes ejemplos asumimos que tu ptrace_scope es 0 (sin restricciones de Yama), pero si las tuvieras puedes utilizar el programa ejemplo o el lanzador l2t compilar el ejemplo anterior y lanzarlo en segundo plano redireccionando la salida estándar a /dev/null.

Por ejemplo, si lanzamos el comando sleep en segundo plano (recuerden que sleep se suspende el número de segundos indicados para luego volver a ejecución y terminar):

sleep 10 &

suponiendo el que el pid del proceso sleep es 124, ejecutamos:

strace -p 124 -o trace.txt

Obtenemos en trace.txt solamente:

```
restart_syscall(<... resuming interrupted read ...>) = 0
close(1)                        = 0
close(2)                        = 0
exit_group(0)                     = ?
+++ exited with 0 +++
```

Es decir, cuando el proceso sleep deja de estar suspendido, se produce una continuación en una llamada del sistema read que fue interrumpida en el momento de la suspensión.

Otras opciones básicas de strace son:

- Opción -f: Realiza la traza de procesos hijos derivados del proceso monitorizado. Suele combinarse con la opción –output-separately para que en el caso de que la salida se vuelque a un archivo mediante la opción -o filename, se cree un archivo filename.pid por cada proceso monitorizado.
- Opción -e trace=syscall_set (--trace=syscall_set). Permite especificar el conjunto de llamadas del sistema a monitorizar. Esta opción es muy importante porque permite filtrar el contenido de la traza. Podemos especificar directamente el nombre de las llamadas del sistema que queremos monitorizar:

  strace -e open,read,write,close,stat -o trace.txt cp p.txt p1.txt

  o su tipología, por ejemplo si usamos el valor file después de -e:

  strace -e file -o trace.txt cp p.txt p1.txt

Solo se incluirá en trace.txt las llamadas del sistema que tengan como alguno de sus argumentos un archivo.

- Opción -e status=status_set (--status=status_set). En este caso se filtra por el estado de salida de la llamada. Por ejemplo:

  strace -e status=failed -o trace.txt cp p.txt p1.txt

  Solo incluirá en la traza las llamadas del sistema que devuelven algún código de error. Este caso es habitual así que existe la opción -Z que hace lo mismo, así como -z que solo retiene las llamadas del sistema que acabaron con éxito.
- Opción -c. Esta opción modifica el comportamiento de strace porque solo muestra información final a modo de resumen del comportamiento de las llamadas del sistema y las señales. Por ejemplo:

  strace -c -e trace=file -o trace.txt cp p.txt p1.txt

  Muestra la siguiente tabla:

  ```
  % time     seconds  usecs/call     calls    errors syscall
  ------ ----------- ----------- --------- --------- ----------------
   42.57    0.000639          49        13           newfstatat
   40.31    0.000605          60        10           openat
   11.33    0.000170          85         2         2 access
    5.80    0.000087          43         2         2 statfs
    0.00    0.000000           0         1           execve
  ------ ----------- ----------- --------- --------- ----------------
  100.00    0.001501          53        28         4 total
  ```

  En la ayuda de strace verán opciones para modificar el formato de la salida, por ejemplo suprimir los mensajes relacionados con la actividad propia de strace, o determinar el ordenamiento preferido para la tabla resumen obtenida con -c.

3. Recomendaciones generales.

La práctica es estrictamente individual: no compartan código con los compañeros, ni utilicen sistemas de IA como CoPilot o Chat GPT. Si tienen problemas en la realización, acudan a tutorías de sus profesores de prácticas.

Además, cumplir o no las siguientes recomendaciones tiene influencia en la calificación del trabajo, como se refleja en la rúbrica de la tarea.

- Recuerda facilitar la lectura de tu código:

    - Usa comentarios en los elementos más complejos del programa, para que te sea más sencillo entenderlo cuando haya pasado un tiempo y no te acuerdes.
    - Usa funciones para compartimentar el programa, así como variables y constantes (variables en mayúsculas) que hagan tu código más legible.
    - Incluye código para procesar la línea de comandos.
- Incluir la ayuda del comando:
    - Se debe mostrar ayuda sobre el uso si el usuario emplea la opción `-h` o `--help`.
    - Se debe indicar el error y mostrar ayuda sobre el uso si el usuario emplea una opción no soportada
- Maneja adecuadamente los errores:
    - En caso de error muestra un mensaje y sal con código de salida distinto de 0. Recuerda la función `error_exit` de ejercicios anteriores y, si lo crees conveniente, reúsala o haz la tuya propia.
    - Trata como un error que el usuario emplee opciones no soportadas.
    - Haz lo mismo con las otras posibles condiciones de error que se te ocurran, ¿has probado a invocar tu programa con opciones absurdas a ver si lo haces fallar?
- Ojo con la sustitución de variables y las comillas. En caso de problemas, piensa en cómo quedarían las sentencias si las variables no valieran nada, ¿tendría sentido para BASH el comando a ejecutar?

4. Desarrollo del script.

Te proponemos que realices el script **scdebug** que incluya las siguientes funcionalidades. Ten en cuenta que salvo que se diga expresamente lo contrario, todas las opciones son combinables en cualquier orden.

4.1. Funcionamiento básico

En este apartado consideraremos la siguiente sintaxis admisible, que irá cambiando a medida que avanzamos:

scdebug [-h] [-sto arg] [-v | -vall] [-nattch progtoattach] [prog [arg1 …]]

a) Cuando se ejecuta el script scdebug, debe hacer lo siguiente:

Si se ha especificado prog (programa) con posibles argumentos, strace debe lanzarse en segundo plano para ejecutar y monitorizar dicho programa.

En todos los usos de strace, la salida de strace se volcará a un archivo cuya ruta debe ser (expresada de forma relativa al directorio home del usuario)

.scdebug/PROG/trace_UUID.txt

donde PROG es el nombre del comando lanzado y UUID la cadena "identificador único universal" generada con la utilidad **uuidgen.** Si los directorios especificados en la ruta no existen deben crearse.

Además, si el comando se introduce con la opción -sto arg el comando strace se ejecutará tomando como opciones lo indicado en arg, que es un solo argumento, de modo que si se quieren pasar varias opciones para strace, habrá que ponerlas todas entre comillas. Veamos algunos ejemplos:

scdebug myprog arg1 arg2

Tendrá que ejecutar algo en segundo plano como:

strace -o .scdebug/myprog/trace_2ca5cbc3-c110-4b11-8d4d-6276bbbe983b.txt myprog arg1 arg2

scdebug -sto '-c -e file' myprog arg1 arg2

Acabará ejecutando en segundo plano algo como:

strace -c -e file -o .scdebug/myprog/trace_2ca5cbc3-c110-4b11-8d4d-6276bbbe983b.txt myprog arg1 arg2

En el caso de que en cualquier ejecución del script que requiera la monitorización de procesos, un lanzamiento de strace produzca un error el script debe terminar con un error indicado también por un mensaje en la salida de error. Este error también deberá quedar reflejado en el archivo de salida.

b) Incorpora la opción -nattch para monitorizar otros procesos que ya están en ejecución **(modo attach)**, de forma que se pueda especificar el nombre del programa a monitorizar, y se opte por **el proceso del usuario** cuya ejecución se inició más recientemente con ese comando. Es aquí donde puedes tropezar con las restricciones de seguridad configuradas en los sistemas, revisa el apartado de Introducción de esta guía para saber cómo proceder.

Un ejemplo de un posible uso sería:

scdebug -sto '-c -e file' -nattch myprog

Aquí, tu script debe lograr que strace **haga un "attach" al proceso del usuario en ejecución más reciente cuyo nombre de comando es myprog**. Es decir, el script debe producir en segundo plano una ejecución como:

strace -c -e file -p 1204 -o .scdebug/myprog/trace_2ca5cbc3-c110-4b11-8d4d-6276bbbe983b.txt

si 1204 es el pid del proceso del usuario, con tiempo de inicio más reciente iniciado por un comando myprog.

Del mismo modo que en el apartado a), el archivo donde se guarda la traza debe guardarse en una ruta coherente con el nombre del comando. Por ejemplo, en el caso anterior, la ruta sería (relativa al directorio home del usuario)

.scdebug/myprog/trace_UUID.txt

Asumiendo el mismo significado para UUID.

Recomendación: Te facilitará el resto del proyecto si creas una función genérica que reciba como argumentos una lista de PIDs y se encargue de realizar el attach de strace. En los siguientes apartados encontraremos más casos que derivan en una lista de PIDs sobre los que hacer el attach de strace y todo el trabajo para irlos resolviendo puedes concentrarlo en esta función.

c) Completaremos el funcionamiento básico permitiendo consultar las trazas ya realizadas añadiendo las opciones -v y -vall a nuestro script.

Estas opciones establecen **el modo de consulta** de nuestro script, de forma que en este modo no se lanzarán operaciones strace. Se trata solamente de consultar los archivos de depuración generados por ejecuciones de strace. El nombre del programa para establecer la ruta del archivo o archivos a consultar, se determina a partir de los argumentos del script, en la misma forma en que se determina el nombre del comando para el programa a ejecutar.

La opción -v sirve para devolver en la salida estándar el contenido del archivo de depuración más reciente entre los guardados para el programa especificado, mientras que -vall va a devolver todos los archivos de depuración ordenados de más reciente a más antiguo. En ambas opciones el volcado de cada archivo debe ir precedido de las líneas:

========== COMMAND: nombre del comando ==========
========== TRACE FILE: nombre del archivo de traza ==========
========== TIME: tiempo de última modificación del archivo==========

Por ejemplo,

scdebug -v myprog

debe revisar el directorio .scdebug/myprog, buscar el archivo con fecha de modificación más reciente y mostrarlo por la salida estándar tras la cabecera descrita anteriormente.

scdebug -vall myprog

Debe revisar el directorio .scdebug/myprog, tomar todos los archivos de ese directorio y mostrarlos por la salida estándar, en orden de más reciente a más antiguo, cada uno con la cabecera descrita anteriormente.

4.2. Monitorización de varios procesos en ejecución

En esta fase la forma de llamar al script obedece a la siguientes sintaxis:

scdebug [-h] [-sto arg] [-v | -vall] [-k] [prog [arg …]] [-nattch progtoattach …] [-pattch pid1 …]

Este apartado amplía las funcionalidades del apartado anterior, no las sustituye. Por ejemplo, el guardado de la salida de los comandos strace se realiza siempre y según las indicaciones del apartado anterior.

a) Cada vez que se ejecute el script (a menos que se utilice la opción -v o -vall) se mostrará información de los procesos del usuario, incluyendo el PID y nombre del proceso bajo trazado (tracee) y el PID y nombre del proceso trazador (tracer). Se ordenarán por el tiempo de comienzo de los procesos trazados, de más reciente a más antiguo.

   Puedes resolver este apartado usando ps con las opciones adecuadas para obtener la lista de procesos del usuario y revisar los archivos /proc/PID/status de los procesos. En ese archivo, el campo **TracerPid** tendrá el pid del proceso trazador en el caso de que ese proceso esté siendo trazado. Si el proceso no está siendo trazado el valor de TracerPid es 0.

b) Implementa la opción -k que tratará de terminar todos los procesos trazadores del usuario, así como todos los procesos trazados. Utiliza la señal KILL.

c) Implementa la monitorización de varios procesos en ejecución con la opción -nattch. Ahora la opción admite una lista de nombres de comando. Para cada nombre de comando se elige el proceso de inicio más reciente para hacer el attachment desde strace.

d) Implementa la monitorización de varios procesos con la opción -pattch que puede recibir un número indeterminado de pids de procesos a monitorizar con strace. Los nombres de

los comandos a utilizar para el registro de las salidas se obtendrán a partir de la información de los procesos dado su pid.

e) Si se incluyen las opciones -v o -vall su funcionamiento será equivalente a la de la sección 1: en lugar de tratar de iniciar la traza de los programas, se volcará la información de los archivos de depuración en la salida estándar. Como ahora se pueden elegir más nombres de comandos el volcado se realizará sobre todos.

f) Tratamiento de errores. Vamos a distinguir dos casos:
i) Caso general. Por ejemplo, detectamos opciones inválidas en los argumentos de nuestro script, detectamos que comandos esenciales para nuestro script no están disponibles (como strace o uuidgen), u otros errores. En este caso, debemos detectar el error, producir un mensaje en la salida de error indicando los motivos del mismo y hacer un exit del script con un valor diferente de 0. Miren los apuntes de la semana 6.

ii) **Llamadas a strace en segundo plano.** En este caso, nuestro script no "esperará" a la terminación del comando strace, porque ha sido lanzado como un trabajo en segundo plano y por tanto no vamos a saber a continuación del lanzamiento cómo terminó. Sin embargo, las ejecuciones de strace para depurar programas pueden acabar en un error, que se refleja como en cualquier otro programa en el código de salida de error (parámetro ?, cuando se consulta en expansión de variables es $?) Si este código de salida es diferente de 0, sabemos que algo ha ido mal: puede haber sucedido un error en el programa monitorizado o bien en el propio strace cuando está realizando el attach, por ejemplo, "operación no permitida".

Puesto que no vamos a esperar en nuestro script a que el strace en segundo plano termine, vamos a dar cuenta del error producido mostrando un mensaje de error producido por nuestro script en la salida de error y añadiendo este mismo mensaje al archivo de depuración.

Sin embargo hay un problema. Supongamos que tratamos de usar || para lanzar un comando en caso de que strace en segundo plano devuelva un código de error. Veamos algunas soluciones erróneas:

strace -p 1024 -o archivo.log & || echo "mensaje de error" **# OJO: Esto está muy mal**

Lo anterior no tiene sentido y además no se ejecuta, porque da un error de sintaxis (& y || son ambos operadores de separación de comandos en una lista y no pueden ir seguidos.) Otra alternativa también equivocada:

strace -p 1024 -o archivo-log || echo "mensaje de error" & **# OJO: Esto no es lo que buscamos**

El anterior comando lanzará strace y luego el siguiente comando en segundo plano si strace acaba en error. Con esto, nuestro script tendría que esperar por cada strace lanzado y ese no es el funcionamiento requerido del script.

Veamos ahora cómo podemos proceder. Hay diferentes soluciones para esto, todas ellas pasan para lanzar una lista de comandos en segundo plano. Una que puede ser práctica para este problema, es crear una función para lanzar los strace. Un ejemplo sería:

```
launch_strace()
{
strace -p $1 -o $2 || echo “mensaje de error”
}
```

Entonces podríamos lanzar la función en segundo plano:

```
launch_strace  1024   archivo.log &
```

De esta manera, después de esta línea, podríamos continuar con los comandos sin esperar al strace, mientras que al acabar el strace, si termina en error se ejecutaría el comando echo.

Recuerden que sería interesante que el mensaje de error contuviese más información como el pid del proceso monitorizado y el código de salida de strace.

Se nos pide también que el mensaje de error se muestre en la salida error y se incorpore a nuestro archivo de depuración. Una solución está en el comando **tee**, que puede actuar como un filtro volcando la entrada estándar a la vez en la salida estándar y en un archivo. El comando tee tiene la opción -a, para hacer append al archivo en lugar de sobreescribirlo. Además recuerden que disponen del operador de redirección 1>&2 que establece la salida estándar al mismo destino que la salida error.

4.3 Monitorización a partir de un grupo de procesos con resumen de estadísticas

Objetivo

En este apartado vamos a crear un modo de funcionamiento de nuestro script para procesos de los que queremos una tabla con datos referentes al uso de las llamadas del sistema tras su ejecución de principio a fin. La mecánica tiene dos fases:

a) Primero añadimos cada programa a monitorizar al grupo. Esto se hace para cada programa, preparando un proceso detenido e identificable en el conjunto de procesos del usuario, que se pasará a ejecución al iniciar la monitorización.
b) A continuación lanzamos la monitorización. Los procesos añadidos al grupo en la primera fase son puestos en ejecución lo que implica la aplicación de strace en modo attach a cada uno de ellos y la ejecución del programa a monitorizar a continuación.

Herramientas necesarias

Para esta parte del proyecto, necesitarás poder monitorizar a la propia bash en modo attach. Probablemente estés trabajando en un sistema con el módulo de seguridad Yama impidiendo este modo de uso de la llamada ptrace, por lo que tendrás que utilizar una bash configurada como trazable por cualquier proceso. En el aula virtual puedes descargarte el código de ptbash.cc, donde se llama a prctrl antes de lanzar el programa /bin/bash con execve para lanzar este modo.

Tras compilar con:

```
g++ ptbash.cc -o ptbash
```

Con el ejecutable de ptbash en la misma carpeta que tu script, deberás modificar la primera línea para que el intérprete sea ptbash, es decir:

```
#!./ptbash
```

Tras este cambio, la ejecución del script estará a cargo de bash, solo que ahora esta bash puede ser monitorizada en modo attach.

Tareas a realizar

a) Implementa la “acción stop”. Esta acción obedece a la siguiente sintaxis:

```
scdebug [-h] [-k] -S commName prog [arg...]
```

Como ves, la idea es que no sea compatible con prácticamente ninguna de las opciones anteriores, salvo con -h y -k. La opción -S va acompañada obligatoriamente de un nombre de comando commName, que usaremos para forzar el nombre de comando de nuestro proceso monitorizado. Además es obligatorio especificar una programa a ejecutar con sus posibles argumentos: prog [arg …]

Esta acción tiene tres fases:

1) Forzar el nombre de comando. Lo haremos del siguiente modo:

```
echo -n "traced_$commName" > /proc/$$/comm
```

Recuerda que $$ es el pid del proceso de tu bash, de modo que el archivo especial /proc/$$/comm es el nombre de comando que tiene en cada momento tu proceso. Recuerda que este nombre de comando es el que muestra “ps” cuando se usa comm en la opción -o. Como ves, este nombre de comando se puede modificar desde el propio proceso, cambiando el archivo especial comm en la carpeta de archivos especiales del proceso en ejecución en /proc/PID.

El nombre de comando de los procesos del grupo a monitorizar tendrá el prefijo traced_ , de modo que los podemos identificar fácilmente por ese nombre.

2) Tras forzar el nombre de comando, detendremos el script con:

   kill -SIGSTOP $$

La señal STOP será activada en el propio proceso que ejecuta nuestro script y quedará detenido. Esto hará que el usuario regrese a la bash desde donde lanzó el script, dado que automáticamente pasará a primer plano.

3) Sin embargo, el proceso no ha muerto, sigue siendo un proceso de usuario, esperando a continuar su ejecución. Después de la línea kill, debemos poner lo que queremos que ocurra cuando vuelva a ejecución, que será por ejemplo

   exec $LaunchProg

   Donde LaunchProg es una variable con el programa a ejecutar seguido de sus argumentos (prog [arg ….])
   El comando exec sustituye el binario y los datos del programa en ejercicio, en este caso bash, por el programa que establezcamos. Es así como se lanzan los programas. De este modo tras la reanudación después de la detención, se ejecutaría el programa a monitorizar.

   Esta última línea completa la mencionada acción STOP. Ahora tenemos que ocuparnos de la monitorización.

b) La sintaxis del comando para iniciar la monitorización del grupo de procesos detenidos es:

scdebug [-h] [-k] -g | -gc | -ge [-inv]

Nuevamente las únicas opciones compatibles son -h y -k. La opciones -g, -gc y -ge son excluyentes entre sí.

Independientemente de la opción elegida, el trabajo a realizar se hará sobre la lista de pids de los procesos del usuario que estén detenidos y cuyo nombre de comando comience por

“traced_”, es decir, la lista de pids del grupo de procesos preparados por acciones STOP previas.

En los tres casos la mecánica de funcionamiento para cada pid de la lista sigue los pasos siguientes:

a) Llamada a strace **en segundo plano** para hacer el attach al pid.
b) Hacer un sleep durante un tiempo prudencial, por ejemplo sleep 0.1, de forma que de tiempo a que strace complete el attach sobre el proceso detenido.
c) Activar la señal CONT en el proceso hasta ahora detenido: kill -SIGCONT pid

Si se utiliza la opción -g, la llamada a strace se realizará en la misma forma que en la monitorización normal, guardando el archivo de depuración en la ruta construida como se ha explicado anteriormente.

En cambio, si se usan las opciones -gc o -ge, la llamada a strace se hará con la opción -c para que la salida de strace sea una tabla de datos con información sobre las llamadas del sistema realizadas. Junto con la opción -c utilizaremos la opción -U para obtener los parámetros: nombre de la llamada del sistema (name), tiempo máximo en la llamada (max-time), el tiempo total (total-time), número de llamadas (calls) y número de errores (errs). En esta llamada a strace no usaremos la opción -o para guardar a un archivo, sino que haremos una redirección de la salida error, que es donde strace envía la información a la salida estándar para guardar la salida en una variable, ya que la tabla de datos que ofrece strace así la vamos a condensar para cada proceso en una fila resumen.

En esta fila resumen hay que incluir el nombre de la llamada del sistema que tiene el mayor max-time, este max-time, y los totales acumulados (última fila de la salida de strace -c) del total-time, calls y errs.

Con las filas de cada proceso construiremos una tabla que será lo que se muestre por la salida estándar tras la ejecución del comando. Esta tabla estará ordenada por el número de llamadas en el caso de usar la opción -gc y por el número de errores si se usa -ge (en orden ascendente). Si se aplica la opción -inv el orden será descendente.