Programas de Usuario utilizando POSIX

Silvana Ardanaz, Fernando Cerdeira, Juan Martín Hernández, Marcelo Luciano, Juan Manuel Roccetti
silvanaardanaz@yahoo.com.ar-fercerdeira@yahoo.com.ar-juanmartinhernandez@hotmail.com-marcheloluc@gmail.com
juanmroccetti@gmail.com

Los comandos de uso general en SODIUM han sido programados junto al shell en su primera etapa, cuando solo podía ejecutarse código en modo kernel. Desde que se le dio la capacidad al SODIUM de ejecutar procesos en modo usuario, solo el shell y una pequeña parte de los comandos programados en el antiguo shell han sido convertidos a programas de usuario propiamente dichos.
El objetivo de este trabajo práctico es adaptar los programas de usuario, para que dependan únicamente de bibliotecas provistas por SODIUM, cuya interfaz obedezca y se limite al estándar POSIX para interfaces de sistema.
Para este fin, se deberá corregir, o crear las funciones e interfaces que estén mal o faltantes respectivamente. Al final del trabajo práctico, los antiguos comandos de Shell que se ejecutaban en kernel, pasarán a ser programas que funcionen íntegramente mediante interfaces definidas por el estándar POSIX. También se buscará mantener los dos modos de trabajo (Syscall/Callgate) mediante dos bibliotecas.

Palabras Claves: Sodium, Callgate, SystemCall, Posix, API, Modo Kernel, Modo Usuario, Instrucción privilegiada.

1 Introducción

El estándar POSIX está actualmente en desarrollo, y su principal objetivo es permitir la portabilidad de aplicaciones a nivel de código fuente, es decir, que sea posible portar una aplicación de una computadora a otra sin más que recompilar su código. Está siendo desarrollado por la *Computer Society* de IEEE. El POSIX es un grupo de estándares en evolución. Cada uno de los estándares que lo componen cubre diferentes aspectos de los sistemas operativos. Algunos de ellos ya han sido aprobados, mientras que otros están aún en fase de desarrollo.

El sistema operativo Sodium dependía mucho del kernel. Prácticamente en su totalidad, ya que todos los comandos a ejecutar, independientemente de que podrían

haber estado implementados en programas de usuario, se encontraban completamente implementados del lado del kernel, como, por ejemplo, el comando ayuda que solo realiza impresiones en pantalla, con lo que el mismo se sobrecargaba. Además, hemos encontrado que había algunos archivos, tales como scrap.c o sodshell.c, que poseían muchas funciones que no tenían relación funcional alguna entre ellas y que a su vez son esenciales en el funcionamiento de, tanto el Shell como los comandos.

La mayoría de los comandos encontrados, realizaban callgates al kernel. En realidad, esto no era útil ya que como los comandos y funciones asociadas se encontraban del lado del kernel, llamarlo de su mismo lado carecía de sentido.
Los comandos ps.c y shutdown.c se encontraban implementados correctamente. Sin embargo, se pueden implementar con syscalls con lo que la llamada sería más directa, es decir, de usuario a kernel (nivel 3 a nivel 0) directamente, no como con las callgates que se va pasando de nivel uno a uno.

Como consecuencia del Trabajo Práctico anterior logramos mediante la creación de programas de usuarios que utilizan callgates disminuir la sobrecarga del kernel. Otra consecuencia fue la eliminación del archivo scrap.c en el que se encontraban funciones utilizadas por los comandos. Estas mismas fueron distribuidas en otros .c utilizando un criterio funcional.

1.1 Análisis de la estructura

Para la resolución del Trabajo Práctico realizamos una tabla compuesta por los comandos encontrados principalmente en scrap.c y sodshell.c. Luego de un análisis de la funcionalidad de cada uno, se propusieron cuales serían las ubicaciones correctas y su justificación correspondiente. La tabla se adjunta al final del documento.

Una vez finalizada, y utilizando los comandos de ejemplo ps.c y shutdown.c, se seleccionaron los comandos que debían ser programas de usuario, los cuales realizan callgates a sus correspondientes funciones ubicadas en los lugares definidos en la tabla creada anteriormente.

El principal objetivo de este trabajo es no sólo modificar los programas de usuario creados anteriormente para que utilicen Syscalls sino también crear las llamadas e implementar un mecanismo que permita al usuario del Sodium la posibilidad de elegir como desea ejecutar los comandos, con Syscall o con Callgate. Para su posterior análisis también implementaremos un Log en el que quedará documentado las llamadas a funciones utilizadas para poder ejecutar el llamado al sistema solicitado.

2 Llamadas al sistema

La mayoría de los comandos de Shell utilizan/requieren un servicio del sistema para llevar a cabo la funcionalidad correspondiente. Estas llamadas tienen una API que el estándar POSIX brinda para comunicar al proceso de usuario con el sistema.

Para que se pueda decir que un sistema cumple el estándar POSIX, como dice Stefan Beyer [BEY-05], "el sistema tiene que implementar por lo menos el estándar base de POSIX. Sin embargo, muchas de las interfaces más útiles están definidas en extensiones. La implementación de esas extensiones no es obligatoria, pero casi todos los sistemas modernos soportan las extensiones más importantes".

Las más utilizadas son las interfaces para:

- La creación y la gestión de procesos
- Entrada-salida
- Comunicación sobre redes (sockets)
- Comunicación entre procesos (IPC)
- Señales

Gestión de procesos

Cada proceso tiene su propio espacio de memoria.

Generalmente se utiliza un algoritmo de planificación para dar intervalos de tiempo de ejecución a los distintos procesos activos en el sistema (identificados mediante un PID)

Entrada/Salida

Los recursos para entrada y salida aparecen como un archivo en el sistema de archivos. Por eso las llamadas al sistema para abrir y cerrar archivos y para escritura y lectura son de las más importantes. Cada archivo abierto al nivel del sistema está identificado con un número entero positivo llamado *descriptor de archivos*.

Comunicación sobre redes

Comunicación mediante sockets.

Un socket es un extremo de un canal de comunicación. Al nivel de programación un socket es solamente un descriptor de archivo. Así, es posible utilizar las llamadas entrada/salida introducidas anteriormente sobre un socket.

Comunicación entre procesos

Esta interfaz (también llamada POSIX IPC) permite compartir datos entre procesos del mismo sistema mediante tres tipos diferentes de comunicación: *colas de mensajes, semáforos y memoria compartida*.

Señales

Las señales son notificaciones enviadas a un proceso para notificar el procesamiento de varios eventos importantes. Una señal hace que un proceso en ejecución se detenga y obliga al sistema operativo a manejar la señal. Cada señal es representada por un valor de entero, número entero, tal como 1 ó 2, así como un nombre simbólico. Este nombre es normalmente definido en un archivo header, /usr/include/signal.h.

Cada señal puede tener un manejador de señal, que es una función que es llamada cuando el proceso recibe la señal. Un manejador de Señal se dice que está en modo asincrónico, ya que no es llamada por el código en un programa.

Después de ejecutar la señal, el manejador (handler) retorna el control al proceso que estaba corriendo antes de la interrupción.

Las señales son muy similares a las interrupciones en su comportamiento. La diferencia es que mientras las interrupciones se envían al sistema operativo a través del hardware, las señales se envían al proceso por el sistema operativo, o por otros procesos.

Note que las señales no están relacionadas con las interrupciones de software, que todavía son enviadas por el hardware.

Existen tres formas diferentes de enviar una señal a un proceso.

- Desde el teclado.
- A través del Shell mediante comandos.
- Desde un proceso utilizando syscalls.

3 Programas de Usuario

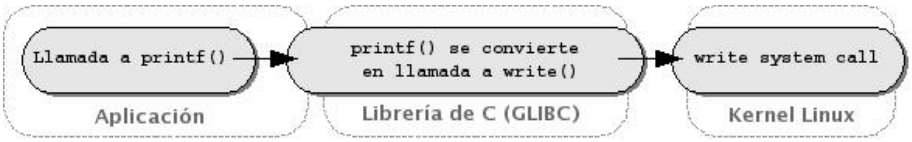
Estos comandos se manejan mediante systemcalls. Un syscall se usa específicamente para ejecutar una operación en modo kernel ya que el código de usuario habitual no se le permite hacer esto por razones de seguridad. [LOV-05]

Por ejemplo, si desea asignar memoria, el sistema operativo tiene el privilegio de hacerlo (ya que sabe las tablas de páginas y se le permite acceso a la memoria de otros procesos), pero a un programa de usuario no se le debe permitir debido a que puede afectar la memoria de otros procesos.

3.1 El puente API/syscall

Para que nuestra aplicación de espacio de usuario pueda emitir una syscall, es necesario poder realizar la llamada mediante un API que alguien nos proporcione. Este API lo proporciona la librería estándar de C de nuestro sistema, la libc o glibc en el caso de Linux. Una de las APIs más comunes del mundo Unix es la basada en el estándar POSIX.

En la siguiente figura podemos ver la relación existente entre las aplicaciones, la librería de C y la llamada al kernel.



Hay que decir que el estándar POSIX se refiere a las APIs, no a las syscalls, es decir, define un comportamiento, pero no como lo debe hacer. Un sistema puede ser certificado como compatible POSIX, pues ofrece un apropiado conjunto de APIs a los programas, no importando como las correspondientes funciones han sido implementadas. Desde el punto de vista del programador de aplicaciones, la distinción entre una API y una syscall es irrelevante, lo único que importa es el nombre de la función, los tipos de los parámetros y el valor devuelto.

Por otro lado, también hay que destacar que el estándar POSIX recomienda que exista una correlación uno a uno entre la función de API y la llamada al sistema. Es decir, el API de una syscall debe ser igual al formato usado en la syscall del kernel. Pero no se garantiza que detrás de un API de una syscall se invoque la pertinente syscall, sino que puede que la funcionalidad de la misma se esté proporcionando desde la propia librería de C, sin invocar al kernel o que el resultado sea la combinación de varias syscalls del kernel. Lo ideal, en aras de la eficiencia y velocidad de ejecución es que la syscall exista realmente en el kernel.

4 Desarrollo

Se comenzó a modificar los programas de usuario para que utilicen Syscall.
Comandos que ya están implementados utilizando Syscall:

Comandos	Nombre	Origen	Parametros Recibidos	
verSem	lFnSysVerSem	\kernel\semaforo.c	0	-
verShm	lFnSysVerShm	\kernel\semaforo.c	0	-
cambtec	lFnSysCamb	kernel\drivers\teclado.c	1	icomandoPos (int)
cls	lFnSysCls	kernel\interfaz\consola.c	0	-
ver	lFnSysVer	kernel\interfaz\consola.c	0	-
lpart	lFnSysListPart	kernel\mem\sodheap_kernel.c	0	-
lporc	lFnSysListPorc	\kernel\mem\sodheap_kernel.c	0	-
borrarpf	lFnSysBorrarPf	\kernel\mem\sodheap_kernel.c	0	-
resetpf	lFnSysResetPf	\kernel\mem\sodheap_kernel.c	0	-
addpf	lFnSysAddPf	\kernel\mem\sodheap_kernel.c	1	iPorcentaje
mem	lFnSysmem	\kernel\mem\sodheap_kernel.c	0	-

ps	lFnSysPs	kernel\gdt.c	0	-
dump	lFnSysDump	\kernel\mem\sodheap_kernel.c	2	iArg1, iArg2
kill	lFnSysKill	kernel\syscall.c	2	pid,sig (int)
stack	lFnSysStack	kernel\gdt.c	2	iPar1 iPar2
gdt	lFnSysGdt	kernel\gdt.c	1	iComandoPos
gdtall	lFnSysGdt	kernel\gdt.c	1	iComandoPos
idt	lFnSysIdt	kernel\system.c	1	iParam
idtall	lFnSysIdt	kernel\system.c	1	iParam
tss	lFnSysTss	\kernel\gdt.c	1	iPid
segs	lFnSysSegs	\kernel\mem\sodheap_kernel.c	0	
pag	lFnSysPag	\kernel\gdt.c	1	iPid
log	lFnSysLog	kernel\interfaz\consola.c	0	
fd	lFnSysFd	\kernel\fs\fat12.c	0	
ls	lFnSysLs	kernel\fs\fat12.c	1	iArg1

4.1 Pasos realizados para realizar una nueva llamada al sistema

Para implementar una nueva llamada al sistema se siguieron los siguientes pasos:

1. Agregar en include/common/syscall_def.h un #define con el numero de syscall
2. En el comando correspondiente agregar la llamada al syscall
3. En system.c agregar en la funcion lFnHandlerSyscall en el case el numero de define del nuevo llamado syscall que se va a crear
4. En syscall.c crear la syscall (ejemplo: SysPS) que llama a la funcion correspondiente que usa el comando (en este caso:MenuPs()) y devolver un un long indicando si se pudo hacer la llamada.

4.2 Cargar los programas de usuario

Debido a que son varios los programas de usuario creados y en su totalidad no pueden ser cargados en memoria principal, se necesitó realizar una carga por separado de los programas. Igualmente este procedimiento de ejecución será de manera momentánea hasta que se pueda utilizar el dispositivo de almacenamiento de masivo para poder cargarlos todos allí.

La manera que implementamos para resolver ésta *carga por separado* fue al script donde se listan todos los programas binarios de usuarios indicarle por orden alfabético la lista a cargar. En total son cuatro listas y solo se podrá levantar una al mismo tiempo. Esto garantiza que una vez que se ejecuten los scripts de inicio, se van a generar los nuevos makefile con los binarios pasados. Esto permitirá que en usr\bin podamos tener todos los fuentes de los programas

de usuario. La consecuencia de no trabajar de esta manera produciría una falla en el Programa Cargador (Loader) impidiendo la ejecución del Sistema Operativo.

4.3 Implementación de bibliotecas

Dado que la principal finalidad del Sodium es didáctica hemos desarrollado un mecanismo para que el sistema se pueda utilizar con Syscall o Callagate según lo que el usuario desee.
Para esto hemos desarrollado 2 bibliotecas basándonos en libsodium.c, la biblioteca libsodium_call.c en la que se encuentran desarrolladas todas las llamadas a callgate y libsodium_sys.c en la que están desarrolladas las llamadas a syscall. También se han modificado los llamados en los programas de usuario por llamados genéricos.
La única desventaja de esto es que el usuario debe seleccionar manualmente la biblioteca. Lo que se debe hacer es: antes de compilar el Sodium se debe ir a la carpeta usr\lib y buscar libsodium_call.c o libsodium_sys.c y cambiar el nombre del archivo a libsodium.c. Si se quiere que el Sodium trabaje con syscall lo que se debe hacer es cambiar el nombre al programa libsodium_sys.c; por otro lado si se quiere que el Sodium trabaje con callgate se debe cambiar el nombre a libsodium_call.c. Siempre debe haber un único programa que se llame libsodium.c para que el Sodium funcione.

4.4 Log

Uno de los aspectos interesantes de los comandos a analizar son: cuales llamadas al sistema hace, qué parámetros pasa y cuáles registros del procesador usa. Para ello se pensó en utilizar al log. Como objetivo se propuso obtener la trazabilidad de que parámetros utilizan las syscall, que valor devuelven y que registros del procesador utilizan. Esto es visualizado por pantalla utilizando el Log ya implementado años anteriores en el sistema.

El log se puede visualizar de dos maneras: con el comando log, (ejecutar log nuevamente para salir) u oprimir la tecla F2 (para salir del log apretar F1).

5. Conclusión

Nuestro trabajo consistió en primer lugar que los comandos se ejecuten como programas de usuario y luego realicen las llamadas al sistema para ejecutar todas aquellas acciones que se requieran bajo modo kernel. Un total de aproximadamente 30 comandos fueron pasados a programas de usuario.
Luego, decidimos no dejar únicamente al sistema con un único modo de trabajo de manera excluyente, para ello nos propusimos manejar la interfaz bajo dos bibliotecas, según cual sea compilada podrá trabajar con callgate o syscall. Como una mejora la carga de ésta biblioteca deberá ser dinámica, actualmente se hace de manera estática.
Por último, para que se pueda visualizar la trazabilidad de las llamadas al sistema que produce un comando, en el log del sistema se registrarán los parámetros de la llamada, nombre y registros que utiliza.
Los siguientes comandos se deben verificar debido a que su funcionalidad es deficiente: verSem, verShm, rm, stack y pag.

6. Bibliografía

1. [WAL98] Walli Stephen R.,"The Posix Family of Standars",ACM,1998.
2. [LOV05] Love Robert, "Linux Kernel Development" Second Edition, Novell Press, 2005.
3. [CAR01]Carretero Jesús. "Prácticas de Sistemas Operativos",Mc Graw Hill,2001. pág 50-60.
4. [IEE08]1003.13 TM IEEE Standard for Information Technology Standardized Application Environment Profile (AEP) "POSIX® Realtime and Embedded Application Support",IEEE,2008.
5. [MIC05] Microsoft Consulting Service, "Writing Posix", Microsoft, 2005.
6. [BEY05]Beyer Stefan, "Programación de sistemas UNIX(POSIX)",Instituto Tecnológico de informática de Valencia,2005.