Cómo añadir una nueva llamada al sistema en Linux 3.5

Diego Rivera V.

13 de septiembre de 2012

Las llamadas a sistema, son procedimientos que se pueden invocar desde el espacio de usuario (en un programa en C, por ejemplo), y que requieren que sean atendidas por el kernel del Sistema operativo (en el espacio del kernel), por lo que se debe realizar un cambio de modo de ejecución para atender estas llamadas. Algunos ejemplos de estas llamadas son las primitivas de la libc `read`, `write`, `open`, `rand`, `socket`, entre otras.

1. Esquema organizacional entre el SO y los programas

Por motivos de seguridad, el Kernel del Sistema Operativo (Linux en este caso), corre en un espacio de memoria distinto al que corre cualquier programa que podemos compilar, esto se debe a que el sistema operativo es capaz de hablar directamente con el hardware de la máquina, y él mismo sirve de interfaz para que el programador aproveche las carcaterísticas de esta sin tener que llegar a programación tan a bajo nivel como el Assembler, ni pasar a llevar alguna configuración primordial de algún componente que podría llevar al kernel al pánico.

Por otro lado, estas interfaces se ofrecen en una arquitectura de capas, como se muestra a continuación:

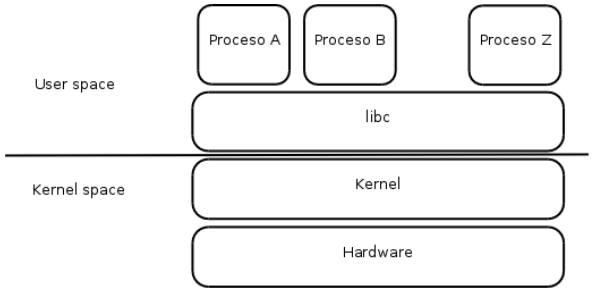

Figura 1: Organización de procesos, libc y Kernel

En ella se puede ver que los programas NO interactúan directamente con el Kernel (aunque podrían hacerlo, a través de algunas macros en C definidas en algunos headers de Linux), sino que lo hacen a través de la librería C standard, más conocida como *libc*.

La libc es la encargada de implementar funciones como `read` y hacer las llamdas al Kernel necesarias para obtener los datos que se requieren (en este caso, `read` realiza una llamada a la funcion `sys_read`, la cual está implementada en el núcleo de Linux).

2. Añadiendo una llamada al sistema nueva

2.1. Obtener los fuentes de Linux 3.5 y descomprimirlos

```
$ wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.5.2.tar.bz2
$ tar xvfz linux-3.5.2.tar.bz2
$ cd linux-3.5.2/
```

2.2. Añadir el archivo con la nueva primitiva

En este paso, se añadirán el(los) archivos con el código fuente de la nueva llamada. Estos archivos deben quedar bajo el directorio emphkernel

```
$ cd kernel/
$ touch dummy_syscall.c
```

Con esto creamos el archivo `dummy_syscall.c` en blanco, el cual contendrá el código fuente de nuestra llamada. En este ejemplo, el código será el siguiente:

```
#include <linux/linkage.h>
#include <linux/kernel.h>

asmlinkage int sys_dummy_syscall(){
        return(127);
}
```

El header `<linux/linkage.h>` contiene la definición de la macro `asmlinkage` que se encarga de definir la función como visible afuera del archivo en donde se define. Asimismo, el header `<linux/kernel.h>` contiene definiciones para funciones utilitarias como `printk`.

Por otro lado, cualquier llamada al sistema debe ser nombrada con el prefijo "sys_" de lo contrario el Kernel no compilará correctamente. En este caso, nuestra llamada al sistema se llama `dummy_syscall`, por lo que la función implementada en el kernel se llama `sys_dummy_syscall`.

2.3. Modificando Makefiles de Linux

Una vez que insertamos el código de nuestra llamada, debemos modificar el Makefile de esta carpeta para que nuestro archivo se compile. Esto se puede hacer con cualquier editor (vim, gedit, etc). Sin modificar, el Makefile dice:

```
obj-y = fork.o exec_domain.o panic.o printk.o \
        cpu.o exit.o itimer.o time.o softirq.o resource.o \
        sysctl.o sysctl_binary.o capability.o ptrace.o timer.o user.o \
        signal.o sys.o kmod.o workqueue.o pid.o task_work.o \
        rcupdate.o extable.o params.o posix-timers.o \
        kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \
        hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o \
        notifier.o ksysfs.o cred.o \
        async.o range.o groups.o lglock.o
```

y debe quedar:

```
obj-y = fork.o exec_domain.o panic.o printk.o \
        cpu.o exit.o itimer.o time.o softirq.o resource.o \
        sysctl.o sysctl_binary.o capability.o ptrace.o timer.o user.o \
        signal.o sys.o kmod.o workqueue.o pid.o task_work.o \
        rcupdate.o extable.o params.o posix-timers.o \
        kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \
        hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o \
        notifier.o ksysfs.o cred.o \
        async.o range.o groups.o lglock.o dummy_syscall.o
```

Observar que hemos añadido nuestro archivo con extensión .o al final de la lista *obj-y*; el proceso de compilación sabrá que para generar ese .o hay que compilar el .c que tiene el mismo nombre.

2.4. Añadiendo nuestra nueva llamada a la tabla de llamadas al sistema

Con esto se busca registrar la nueva llamada en el kernel en la carpeta *syscalls* de la arquitectura correcta[1]:

```
kernel $ cd ..
linux-3.5.2 $ cd arch/x86/syscalls/
syscalls $ ls
Makefile  syscall_32.tbl  syscall_64.tbl  syscallhdr.sh  syscalltbl.sh
```

en esta carpeta vemos 2 archivos importantes `syscall_32.tbl` y `syscall_64.tbl`. En estos archivos se definen los códigos de llamadas al sistema para cada arquitectura (32 y 64 bits respectivamente). En este caso editaremos `syscall_64.tbl`. Casi al final del archivo aparece:

[1]Para máquinas Intel y AMD de 32 y 64 bits, la arquitectura correcta es x86. ia64 es otro conjunto de instrucciones que probablemente NO soporte el CPU de tu máquina

```
310     64      process_vm_readv        sys_process_vm_readv
311     64      process_vm_writev       sys_process_vm_writev
312     common  kcmp                    sys_kcmp

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512     x32     rt_sigaction            sys32_rt_sigaction
513     x32     rt_sigreturn            stub_x32_rt_sigreturn
514     x32     ioctl                   compat_sys_ioctl
515     x32     readv                   compat_sys_readv
516     x32     writev                  compat_sys_writev
517     x32     recvfrom                compat_sys_recvfrom
```

y debemos añadir la línea correspondiente a nuestra llamada a sistema, junto con su numero, nombre de la llamada y la función del kernel que la implementa:

```
310     64      process_vm_readv        sys_process_vm_readv
311     64      process_vm_writev       sys_process_vm_writev
312     common  kcmp                    sys_kcmp
313     common  dummy_syscall           sys_dummy_syscall

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512     x32     rt_sigaction            sys32_rt_sigaction
513     x32     rt_sigreturn            stub_x32_rt_sigreturn
514     x32     ioctl                   compat_sys_ioctl
515     x32     readv                   compat_sys_readv
516     x32     writev                  compat_sys_writev
517     x32     recvfrom                compat_sys_recvfrom
```

Los script en bash (.sh) de la misma carpeta *syscalls* se encargaran de traducir estas tablas en las definiciones reales de las llamadas a sistema. Por el momento, tenemos que recordar cual será el valor único que identifica a nuestra llamada, que en este caso es el 313.

2.5. Compilar la imagen de Linux

En este paso, compilaremos la imagen del Kernel modificado que acabamos de producir:

```
syscalls $ cd ../../..
linux-3.5.2 $ make olconfig
```

con esto, copiamos las configuraciones de compilación del Kernel que estamos ejecutando, para que se compilen en el Kernel modificado, y finalmente

```
linux-3.5.2 $ make bzImage
```

lo que empezará a compilar la imagen del núcleo modificada. Este proceso puede tardar varios minutos en demorarse.

2.6. Instalando la imagen de linux y configurando grub

Una vez terminado, la consola dirá en donde dejó la imagen:

```
Setup is 16848 bytes (padded to 16896 bytes).
System is 4649 kB
CRC 76a8a8f3
Kernel: arch/x86/boot/bzImage is ready  (#5)
linux-3.5.2 $
```

por lo que para instalarla, debemos copiar `bzImage` desde donde está, hasta `/boot`:

```
linux-3.5.2 $ cp arch/x86/boot/bzImage /boot
```

Lo único que falta ahora es configrar grub2[2]. Debemos editar el archivo `grub.cfg` ubicado en `/boot/grub2`. Buscar una entrada que sea del tipo:

```
menuentry 'Fedora (3.5.2-1.fc17.x86_64.debug)' --class fedora ...{
  load_video
  set gfxpayload=keep
  insmod gzio
  insmod part_msdos
  insmod ext2
  set root='hd0,msdos5'
  if [ x$feature_platform_search_hint = xy ]; then
    search --no-floppy --fs-uuid --set=root --hint-bios=hd0,msdos5 ...
  else
    search --no-floppy --fs-uuid --set=root ...
  fi
  echo 'Loading Fedora (3.5.2-1.fc17.x86_64.debug)'
  linux   /boot/vmlinuz-3.5.2-1.fc17.x86_64.debug ...
  echo 'Loading initial ramdisk ...'
  initrd /boot/initramfs-3.5.2-1.fc17.x86_64.debug.img
}
```

[2]Grub 2 es la versión de grub que está presente en la mayor parte de las distribuciones linux modernas, es por eso que este how-to se refiere a esa versión. Para saber cómo configurar grub legacy dirijase a `http://www.linuxchix.org/content/courses/kernel_hacking/lesson4`

copiarla y pegarla arriba de ella, modificando el String de *menuentry*(para identificar el Kernel modificado en la lista de GRUB) y la imagen que carga, en la entrada *linux*. Debería quedar como:

```
menuentry 'Mi Linux(3.5.2)' --class fedora ...{
  load_video
  set gfxpayload=keep
  insmod gzio
  insmod part_msdos
  insmod ext2
  set root='hd0,msdos5'
  if [ x$feature_platform_search_hint = xy ]; then
    search --no-floppy --fs-uuid --set=root --hint-bios=hd0,msdos5 ...
  else
    search --no-floppy --fs-uuid --set=root ...
  fi
  echo 'Loading Fedora (3.5.2-1.fc17.x86_64.debug)'
  linux   /boot/bzImage ...
  echo 'Loading initial ramdisk ...'
  initrd /boot/initramfs-3.5.2-1.fc17.x86_64.debug.img
}
```

Luego, guardar el archivo y reiniciar la máquina con el Kernel modificado. Debería aparecer con el nombre `Mi Linux(3.5.2)` en el menú de boot de GRUB.

2.7. Programa de ejemplo para usar la llamada

Un programa de ejemplo que usa nuestra llamada es:

```
#include <stdio.h>
#include <sys/syscall.h>
#include <errno.h>

int main(){
        int aux;
        printf("ejemplo de llamada al kernel especial\n");

        aux = syscall(313);

        printf("syscall returned = %d, errno = %d\n", aux, errno);

        exit(0);
}
```

Observemos que usa la primitiva de la libc `syscall`, la cual se encarga hacer la llamada a sistema con el ćodigo que se le pasa de parámetro (en este caso 313, la llamda que implementamos). Si todo salió bien, al compilar este programa (`gcc -o test test.c`) y ejecutarlo, debería dar el siguiente resultado:

```
$./test
ejemplo de llamada al kernel especial
syscall returned = 127, errno = 0
$
```

lo que nos dice que la llamada está funcionando y haciendo lo que debe. Si ejecutamos este mismo programa en un kernel que no tiene implementada esta llamada a sistema, la salida será:

```
$./test
ejemplo de llamada al kernel especial
syscall returned = -1, errno = 38
$
```

en donde `syscall` retornó -1(algún error ocurrió, segun man), y `errno` tiene el valor 38, que significa *function not implemented*.

3. Referencias

- Códigos de errno y sus significados - `http://www.barricane.com/c-error-codes-include-errno`
- Algunas funciones para copiar memoria entre espacios de memoria (kernel - user y vice versa) - `http://www.ibm.com/developerworks/linux/library/l-kernel-memory-access/index.html`