Agregar un system call a Minix

Alejandro Valdez <avaldez(ARROBA)dc.uba.ar>

1 de Enero de 2005

1. Escenario

Se quiere agregar el system call mysystemcall() al kernel de Minix, se decidió que el system call será atendido por la tarea de FS. La función de mysystemcall() es imprimir un mensaje en la consola.

2. Solución

2.1. Agregar el código del system call dentro del FS task

Este es el código que formará parte del kernel y se ejecutará cuando se invoque el system call. Escribir el código de la función do_mysystemcall() y agregarla al archivo /usr/src/fs/misc.c.

```
...
PUBLIC int do_mysystemcall()
{
    printf("Ejecutando codigo en modo kernel.\n");
    printf("Podriamos hacer muchas cosas aqui...\n");
    return(OK);
}
```

El archivo /usr/src/fs/proto.h contiene las declaraciones de las funciones contenidas dentro de la tarea de FS. Debemos agregar la declaración de la función do_mysystemcall() al archivo /usr/src/fs/proto.h:

```
/* misc.c */
...
_PROTOTYPE( int do_mysystemcall, (void) );
```

2.2. Publicar la nueva función como un system call

Cada tarea tiene un arreglo llamado call_vector de tamaño NCALLS donde cada posición es un puntero a una función. Cada vez que la tarea recibe la petición de atender el system call número K se realiza un llamado a la función apuntada por call_vector[K].

Incrementar en uno la cantidad de system calls permitidos y definir un número identificatorio para el systemcall mysystemcall(). Editar el archivo /usr/include/minix/callnr.h:

```
#define NCALLS  78  /* number of system calls allowed */
...
...
#define MYSYSTEMCALL  77
```

Indicar a la tarea de FS que debe invocar la función do_mysystemcall() cada vez que se intente acceder al system call número 77. Modificar /usr/src/fs/table.c y agregar:

```
...
PUBLIC _PROTOTYPE (int (*call_vector[NCALLS], (void) ) = {
    no_sys,           /* 0 = unused */
    ...
    ...
    no_sys,           /* 76 = REBOOT */
    do_mysystemcall,  /* 77 = MYSYSTEMCALL */

};
```

Indicar a la tarea MM que debe invocar la función no_sys() cada vez que se intente acceder al system call número 77. La función no_sys() retorna un código de error indicando que se intentó utilizar un servicio que no brinda la tarea. Modificar /usr/src/mm/table.c y agregar:

```
...
PUBLIC _PROTOTYPE (int (*call_vector[NCALLS], (void) ) = {
    no_sys,           /* 0 = unused */
    ...
    ...
    do_reboot,        /* 76 = reboot */
    no_sys,           /* 77 = MYSYSTEMCALL */
};
```

Compilar el código del kernel que acabamos de modificar, ejecutar como el usuario root los siguientes comandos:

```
# cd /usr/src/tools
# make
# make hdboot
# reboot
```

2.3. Invocar el system call desde lenguaje C

Para poder invocar el system call desde un programa escrito en lenguaje C, es necesario utilizar una función de librería como lo es por ejemplo printf(). A continuación veremos los pasos para crear una librería que provea la función de lenguaje C mysystemcall() que podrá ser invocada desde cualquier programa.

Escribimos el cuerpo de la función de librería mysystemcall() que tiene como propósito pedir al kernel la ejecución del system call cuyo número es MYSYSTEMCALL.

Notar que el llamado a _syscall() toma como parámetro un mensaje (dummy_msg), ese mensaje es enviado al kernel y se lo puede acceder desde las funciones implementadas dentro del kernel como ser do_mysystemcall(). De esa manera se pueden enviar datos desde los programas de usuario a las funciones que corren desde el kernel.[1] Crear el archivo _mysyscall.c que contenga:

```
#include <lib.h>
#define mysystemcall _mysystemcall
#include <unistd.h>

PUBLIC int mysystemcall(void)
{
    message dummy_msg;
    return(_syscall(FS,MYSYSTEMCALL,&dummy_msg));
}
```

Al igual que cualquier función en lenguaje C es necesario proveer la declaración de la función de librería. Escribir el archivo mysyscall.h:

[1]Este artículo no incluye los pasos para enviar datos dentro de los mensajes, ver la sección 2.5

```
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( int mysystemcall, (void) );
```

Escribir el archivo mysyscall.s:

```
.sect .text
.extern __mysystemcall
.define _mysystemcall
.align 2

_mysystemcall:
    jmp __mysystemcall
```

Compilar los archivos cuyo código objeto formará nuestra librería, ejecutar como usuario root:

```
#cc -c ./_mysyscall.c
#cc -c ./mysyscall.s
```

Enlazar los archivos _mysyscall.o y mysyscall.o para crear nuestra librería libmysc.a. El compilador de Minix busca las librerias en el directorio /usr/lib/i386 por lo que es necesario copiar nuestra librería a ese sitio. Ejecutar como usuario root:

```
#aal cr ./libmysc.a ./mysyscall.o
#aal cr ./libmysc.a ./_mysyscall.o
#cp ./libmysc.a /usr/lib/i386
```

2.4. Ejemplo

Estamos en condiciones de invocar el nuevo system call que agregamos al kernel, para utilizarlo escribir el programa ejemplo.c:

```
#include "mysystemcall.h"
int main (void)
{
    mysystemcall();
    return(0);
}
```

Compilar ejemplo.c incluyendo el código de nuestra librería y ejecutar el archivo binario resultante:

```
{

#cc ejemplo.c -o ejemplo -l mysc
#./ejemplo
Ejecutando codigo en modo kernel.
Podriamos hacer muchas cosas aqui...
#
```

Este programa está invocando la función de librería mysystemcall() cuya implementación fue escrita en el archivo _mysyscall.c y luego compilada e incluida en la librería libmysc.a. La implementación de la función mysystemcall() envía un mensaje al kernel indicando que se desea la ejecución del system call cuyo número es MYSYSTEMCALL. El kernel envía el mensaje a la tarea de FS y esta realiza el llamado a la función do_mysyscall() definida en el archivo misc.c

2.5. Enviar datos a un system call

El formato de los mensajes que se envian al kernel está definido en /usr/include/minix/types.h en una estructura union llamada message. El union está compuesto por seis tipos de mensajes distintos (mess_1, ..., mess_6), cada uno de ellos está preparado para contener una combinación distinta de enteros y punteros.

Los punteros pasados como parámetro no pueden usarse directamente desde dentro del kernel, esto se debe a que el espacio de memoria donde se ejecuta el programa de usuario (desde donde se envía el mensaje) y el espacio asignado al kernel (donde se recibe el mensaje) son distintos.

Para poder utilizar los punteros recibidos en el mensaje desde dentro del kernel es necesario copiar la información apuntada desde el espacio de memoria de usuario, al espacio de memoria del kernel. La función phys_copy(...,K) permite ralizar esta tarea, K indica la cantidad de bytes a copiar y los puntos suspensivos punteros al origen y al destino de los datos. Esta función no trabaja con strings terminados en NULL, así que será necesario utilizar alguno de los int provistos en la estructura del message para indicar la longitud del dato o bien utilizar una longitud conocida de antemano.

Esta sección solo pretende dar una breve introducción al pasaje de parámetros via mensajes. Una implementación de ejemplo donde se envian datos puede verse en la librería libquota (/usr/src/quota/libquota) incluida en el paquete quotad que puede obtenerse en:

http://www.midnightsoret.com.ar/personales/alejandrovaldez/minix/quotad.tar.Z