NOTAS: PRACTICA 4 NUEVA LLAMADA AL SISTEMA

IMPLEMENTAR UNA NUEVA LLAMADA AL SISTEMA

Supongamos que vamos a definir una nueva llamada al sistema que le damos el nombre de **nueva_llamada** y que tiene tres parámetros, dos de entrada y uno de salida.

int nueva_llamada (int param1, int param2, int *param3)

Los pasos a seguir para implementar la nueva llamada son:

1. Definir un nuevo número para esta llamada

Una llamada al sistema se caracteriza por un nombre, y un número único que la identifica, todas las llamadas están definidas en el fichero:
/usr/src/linux-2.4/include/asm/unistd.h
Veamos un trozo se este fichero unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
….
….
#define __NR_mincore            218
#define __NR_madvise            219
#define __NR_madvise1           219   /* delete when C lib stub is removed */
#define __NR_getdents64         220
#define __NR_fcntl64            221
```

Hay que editar este fichero, buscar el último número utilizado (221) y añadir una línea #define para nuestra llamada.

```
/* añadir nueva_llamada */
#define   __NR_nueva_llamada      222
```

2. Definir un nombre para la nueva llamada

Editar el fichero **usr/src/linux-2.4/arch/i386/kernel/entry.S**
Veamos un trozo de este fichero, referente a sys_call_table

```
.data
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)  /* 0  -  old "setup()" system call*/
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
        .long SYMBOL_NAME(sys_write)
        .long SYMBOL_NAME(sys_open)             /* 5 */
…
…

        .long SYMBOL_NAME(sys_pivot_root)
        .long SYMBOL_NAME(sys_mincore)
        .long SYMBOL_NAME(sys_madvise)
        .long SYMBOL_NAME(sys_getdents64)       /* 220 */
        .long SYMBOL_NAME(sys_fcntl64)
```

y al final de este apartado de la tabla de llamadas, añadir las líneas

```
/* llamada al sistema añadida */
        .long SYSMBOL_NAME (sys_nueva_llamada)

/* mas abajo modificar la línea */
        .rept   NR_syscalls-223
```

Con la actualización de estos dos ficheros, ya tenemos definidos los enlaces para utilizar esta nueva llamada.

3. Escribir el código para la nueva_llamada e integrarlo por ejemplo en el fichero **/usr/src/linux-2.4/kernel/sys.c** que contiene definidas otras llamadas al sistema.

```
asmlinkage int sys_nueva_llamada ( int param1, int param2, int *param3){

. . .

return
}
```

4. **Recompilar el núcleo** e iniciarlo con esta nueva llamada integrada

5. **Escribir una función** que utilice esta nueva_llamada

Para utilizar esta nueva llamada, hay que escribir una función que haga uso de la misma. Esta nueva_llamada no existe en la librería de C libc y hay que declararla explícitamente.

Existe un conjunto de macroinstrucciones en el fichero cabecera linux/unistd.h que permiten declarar este tipo de nuevas llamadas. En el caso de una nueva llamada que tenga tres parámetros hay que utilizar la macroinstrucción
_syscall3:

```
# include <stdio.h>
# include <stdlib.h>
# include <linux/unistd.h>

_syscall3 (int, nueva_llamada, int, x, int, y, int*, result);

main ()
{

nueva_llamada (….);
{
```

La función nueva_llamada es generada por _syscall3, que inicializa los registros del procesador, el número de llamada al sistema se coloca en el registro eax y los parámetros se colocan en los registros ebx, ecx, y edx, y ejecuta la instrucción en ensamblador INT 0x80. Esta es la interrupción software que nos lleva a que el núcleo ejecute la nueva llamada. Al volver de la llamada se retorna un valor al proceso que hizo la llamada, cero es que fue todo bien y negativo contiene un código de error, y puede testearse la variable global del sistema **errno** que contiene información del error.

Veamos el código de **_syscall3** definido en **include/linux/unistd.h**

```
#define _syscall3 ( type, name, type1, arg1, type2, arg2, type3, arg3 ) \
type name ( type1 arg1, type2 arg2, type3 arg3) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" ( __NR_##name ),\
          "b" ( ( long )( arg1 ) ),\
          "c" ( ( long )( arg2 ) ), \
          "d" ( ( long )( arg3 ) ) ); \
__syscall_return ( type,__res ); \
}
```

Donde __syscall_return es:

```
#define __syscall_return(type, res) \
do { \
        if ((unsigned long)(res) >= (unsigned long)(-125)) { \
                errno = -(res); \
                res = -1; \
        } \
        return (type) (res); \
```

```
} while (0)
```

Existen otras funciones para implantar nuevas llamadas que tengan distinto número de parámetros, por ejemplo _syscall4, _syscall5, _syscall6:

```
#define _syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4, \
          type5,arg5,type6,arg6) \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6) \
{ \
long __res; \
__asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int
$0x80 ; pop %%ebp" \
         : "=a" (__res) \
         : "i" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
           "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)), \
           "0" ((long)(arg6))); \
__syscall_return(type,__res); \
}
```