

Proyecto System Call

Link github: https://github.com/CarlosAlvarado-git/SO_Proyecto1.git

¿Qué es un Syscall?

Un System Call, Syscall o Llamada al Sistema es una función con la que se puede invocar los servicios que el sistema operativo ofrece. Usualmente están disponibles como rutinas escritas en lenguajes como C y C++. Usualmente es realizado para que haya un punto de enlace entre el modo de usuario y el modo de kernel, esto para que el usuario no tenga un acceso directo recursos como el cpu o la memoria y sea el sistema operativo quien los administre.

¿Cómo funciona un Syscall?

Utilizando como referencia los System Calls del Kernel de Linux, específicamente la versión 0.12, se puede conocer el funcionamiento de estos y como están estructurados dentro de los ficheros de dicho Kernel.

Primero, cada Syscall está hecho de una forma funcional, por lo que reciben uno o más parámetros y retornan valores, usualmente 0 significa que la ejecución fue exitosa y un valor negativo significa error. En este último caso el tipo de error es almacenado en la variable global '**errno**', y una llamada a la función de librería '**perror()**' podemos imprimir la información del error correspondiente al número de error que obtuvimos.

Cada Syscall tiene un número de función. La lista con los Syscalls se encuentra disponible en el fichero **include/unistd.h**. En nuestro caso, se podría decir que el archivo syscall_64.tbl, ahí es donde podemos encontrar el listado de los syscalls que cuenta nuestro linux kernel (5.6.1).

```

324 common membarrier      __x64_sys_membarrier
325 common mlock2           __x64_sys_mlock2
326 common copy_file_range __x64_sys_copy_file_range
327 64 preadv2             __x64_sys_preadv2
328 64 pwritev2            __x64_sys_pwritev2
329 common pkey_mprotect    __x64_sys_pkey_mprotect
330 common pkey_alloc       __x64_sys_pkey_alloc
331 common pkey_free        __x64_sys_pkey_free
332 common statx            __x64_sys_statx
333 common io_pgetevents    __x64_sys_io_pgetevents
334 common rseq             __x64_sys_rseq
# don't use numbers 387 through 423, add new calls after the last
# 'common' entry
424 common pidfd_send_signal __x64_sys_pidfd_send_signal
425 common io_uring_setup   __x64_sys_io_uring_setup
426 common io_uring_enter   __x64_sys_io_uring_enter
427 common io_uring_register __x64_sys_io_uring_register
428 common open_tree        __x64_sys_open_tree
429 common move_mount       __x64_sys_move_mount
430 common fsopen           __x64_sys_fsopen
431 common fsconfig         __x64_sys_fsconfig
432 common fsmount          __x64_sys_fsmount
433 common fspick           __x64_sys_fspick
434 common pidfd_open       __x64_sys_pidfd_open
# Syscall agregado a tabla de syscalls
438 common pidfd_getfd     __x64_sys_pidfd_getfd
439 common expr_arit        __x64_sys_expr_arit

```

- En nuestro caso, la forma de agregar nuestro syscall a la tabla es más sencilla. En linux 0.12 por ejemplo, aparte de agregarlo de una forma parecida a nuestra, al vector de syscalls, se debe de agregar nuestro syscall. Sólo se especifica el número de syscall que será, la carpeta en donde está la implementación y su “definición”.

```

unsigned int old = current->personality;

if (personality != 0xffffffff)
    set_personality(personality);

return old;

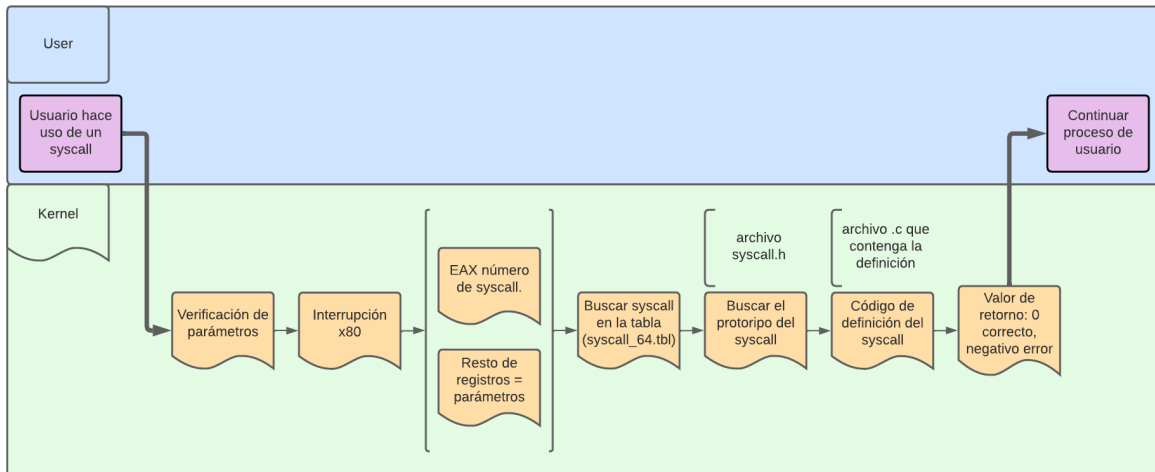
/* for __ARCH_WANT_SYS_IPC */
long ksys_semtimedop(int semid, struct sembuf __user *tsops,
    unsigned int nsops,
    const struct __kernel_timespec __user *timeout);
long ksys_semget(key_t key, int nsems, int semflg);
long ksys_old_semctl(int semid, int semnum, int cmd, unsigned long arg);
long ksys_msgget(key_t key, int msgflg);
long ksys_old_msgctl(int msqid, int cmd, struct msqid_ds __user *buf);
long ksys_msgrcv(int msqid, struct msgbuf __user *msgp, size_t msgsz,
    long msgtyp, int msgflg);
long ksys_msgsnd(int msqid, struct msgbuf __user *msgp, size_t msgsz,
    int msgflg);
long ksys_shmget(key_t key, size_t size, int shmflg);
long ksys_shmctl(char __user *shmaddr);
long ksys_old_shmctl(int shmid, int cmd, struct shmid_ds __user *buf);
long compat_ksys_semtimedop(int semid, struct sembuf __user *tsops,
    unsigned int nsops,
    const struct old_timespec32 __user *timeout);
asmlinkage long sys_expr_arit(char __user *ex, int *result);
#endif

```

- Aquí podemos ver la definición de nuestro prototipo, es decir, la forma en la que el kernel tendrá almacenado la existencia de nuestro syscall, para que luego pueda hacer referencia en donde está nuestra definición del syscall.

Cuando una aplicación invoca una interrupción **INT 0x80**, es la interrupción utilizada para ejecutar Syscalls por un programa, el número de System Call es almacenado en el registro **EAX** y los parámetros en los registros **EBX**, **ECX** y **EDX**. El límite de parámetros posibles para esta versión del Kernel es 3, aunque también podría no recibir parámetros.

Para facilitar la ejecución de un Syscall el código fuente del Kernel define la macro de la función **_syscalln()** en el fichero **include/unistd.h** donde **n** representa el número de parámetros. En caso de ser porciones de data muy grandes puede ser enviado el puntero del chunk de data. Para cada Syscall macro en **include/unistd.h** hay $2+2*n$ parámetros; el primero corresponde al tipo de dato del retorno del Syscall, el segundo es el nombre del Syscall seguido del tipo y nombre del parámetro acarreado por el Syscall. En casos donde se pasen más de 3 parámetros, se utiliza la macro **_Syscall1()** cuyo único argumento es el puntero del primer argumento del Kernel.



En esta gráfica se puede visualizar de una forma más gráfica el proceso que se realiza cuando una función de System Call es llamada y ejecutada tanto desde la vista del modo de usuario como del modo de kernel.

Cada macro será extendida a una función de C que contiene declaraciones inline del lenguaje assenbly (ensamblador). Por ejemplo, como fue visto en clase:

```
SYSCALL_DEFINE0(int, read, int, fd, char *, buf, int, n)
{
    long __res;
    __asm__ volatile (
        "int $0x80"
        : "=a" (__res)
        : "0" (__NR_read), "b" ((long)(fd)), "c" ((long)(buf)),
        "d" ((long)(n)));
    if (__res >= 0)
        return int __res;
    errno = -__res;
    return -1;
}
```

En nuestro caso el código es el siguiente para verificar si una expresión es correcta o no, sólo validando la abertura y cierre de paréntesis y corchetes.

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

```

#include <linux/syscalls.h>

SYSCALL_DEFINE1(expr_arit, char *, ex){
    int i[255];
    int cont = 0;
    int bandera = 0;
    while(*(ex+1) != 0 && bandera == 0){
        switch (*ex){
            case 40:// (
                printk("Desde la funcion:  %c\n", *ex);
                i[cont++] = 1;
                break;
            case 41:// )
                printk("Desde la funcion:  %c\n", *ex);
                if (i[--cont] == 1){
                    i[cont] = 0;
                }
                else
                {
                    bandera = 1;
                    printk("ERROR ARITMETICO\n");
                }
                break;
            case 91:// [
                printk("Desde la funcion:  %c\n", *ex);
                i[cont] = 2;
                cont = cont + 1;
                break;
            case 93:// ]
                printk("Desde la funcion:  %c\n", *ex);
                if (i[--cont] == 2){
                    i[cont] = 0;
                }
                else
                {
                    bandera = 1;
                    printk("ERROR ARITMETICO\n");
                }
                break;
            default:
                break;
        }
        ex = (ex+1);
    }
    if (i[0] == 0){
        printk("EXPRESION CORRECTA\n");
    }else {
        printk("EXPRESION INCORRECTA\n");
    }
    return 0;
}

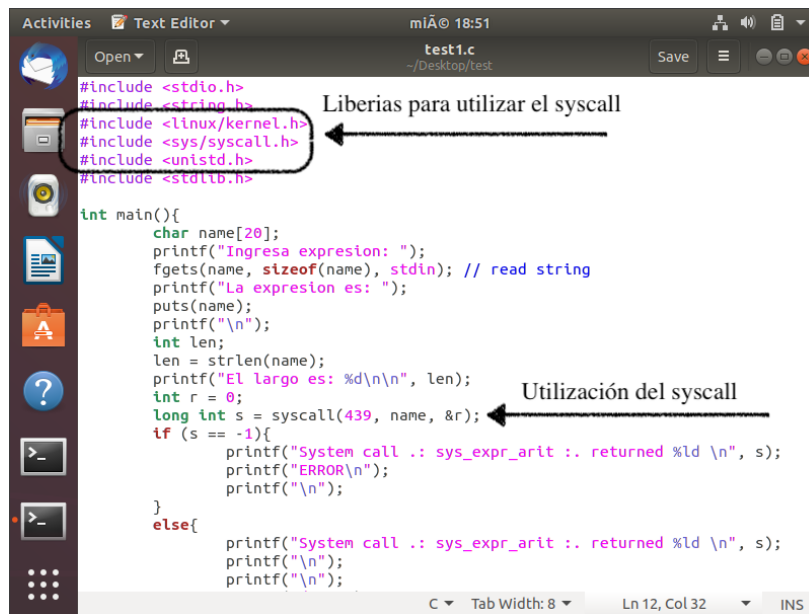
```

El código que implementa la evaluación de la expresión aritmética está en el github.

Este archivo está en una carpeta llamada **expr_arit**. En ella se definen el archivo .c, ejemplo de abajo, y nuestro Makefile, donde definimos el archivo .o para hacer el linker al momento de compilar el kernel.

Como se puede ver que utilizamos SYSCALL_DEFINE1, lo que quiere decir es que el syscall es de un sólo parámetro. Primero vemos el nombre del syscall que implementamos, luego viene el tipo del parámetro y su variable.

Implementación del syscall

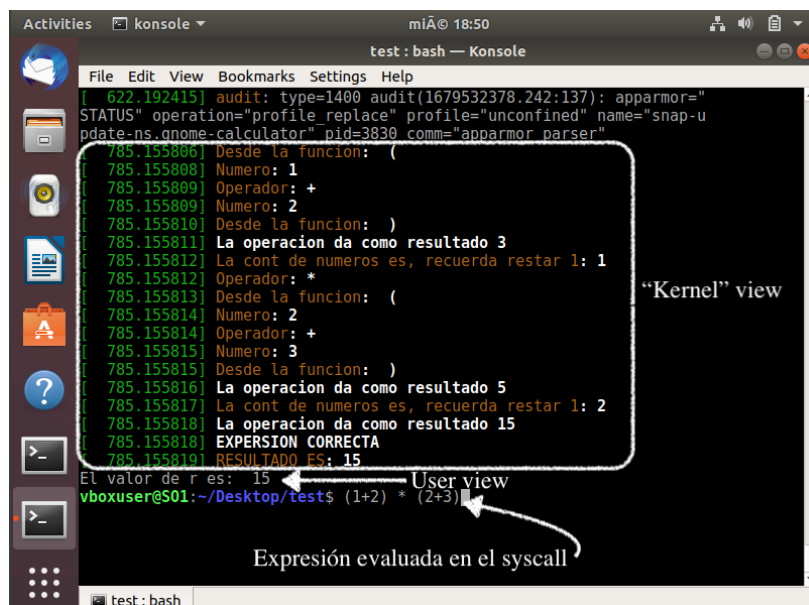


```
#include <stdio.h>
#include <string.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
    char name[20];
    printf("Ingresa expresion: ");
    fgets(name, sizeof(name), stdin); // read string
    printf("La expresion es: ");
    puts(name);
    printf("\n");
    int len;
    len = strlen(name);
    printf("EL largo es: %d\n\n", len);
    int r = 0;
    long int s = syscall(439, name, &r);
    if (s == -1){
        printf("System call :: sys_expr_arit :: returned %ld \n", s);
        printf("ERROR\n");
        printf("\n");
    }
    else{
        printf("System call :: sys_expr_arit :: returned %ld \n", s);
        printf("\n");
        printf("\n");
    }
}
```

Librerías para utilizar el syscall

Utilización del syscall



```
File Edit View Bookmarks Settings Help
[ 622.192415] audit: type=1400 audit(1679532378.242:137): apparmor="
STATUS" operation="profile_replace" profile="unconfined" name="snap-u
pdate-ns.gnome-calculator" pid=3830 comm="apparmor parser"
[ 785.155808] Desde la funcion: (
[ 785.155808] Numero: 1
[ 785.155809] Operador: +
[ 785.155809] Numero: 2
[ 785.155810] Desde la funcion: )
[ 785.155811] La operacion da como resultado 3
[ 785.155812] La cont de numeros es, recuerda restar 1: 1
[ 785.155812] Operador: *
[ 785.155813] Desde la funcion: (
[ 785.155814] Numero: 2
[ 785.155814] Operador: +
[ 785.155815] Numero: 3
[ 785.155815] Desde la funcion: )
[ 785.155816] La operacion da como resultado 5
[ 785.155817] La cont de numeros es, recuerda restar 1: 2
[ 785.155818] La operacion da como resultado 15
[ 785.155818] EXPERSION CORRECTA
[ 785.155819] RESULTADO ES: 15
El valor de r es: 15
vboxuser@S01:~/Desktop/test$ (1+2) * (2+3)
```

"Kernel" view

User view

Expresión evaluada en el syscall