Llamadas al sistema en Linux — 2009

Introducción:

Todo procesador moderno posee al menos dos modos de funcionamiento distintos **modo kernel** (*Protected Mode*) y **modo usuario** (*System Management Mode – SMM*).

A grandes rasgos las diferencias entre estos dos modos son las siguientes:

- En **modo kernel** se encuentran disponibles todas las instrucciones y funcionalidades que la arquitectura del procesador es capaz de brindar, sin ningún tipo de restricciones. Es en este modo en el cual corre el *kernel* (núcleo) del sistema operativo. Por lo general es el *kernel* el único que corre en este modo.

- En **modo usuario** tan sólo un subconjunto de las instrucciones y funcionalidades que la arquitectura del procesador ofrece se encuentran disponibles. En este modo se ejecutan los procesos de los usuarios del sistema (todo proceso corriendo en el sistema pertenece a un usuario). Conocido también como *userland*.

Una de las responsabilidades de un S.O. es administrar los recursos del sistema y los procesos que corren en el mismo. Cada vez que un proceso necesite algún recurso del sistema (memoria, uso de un hardware específico, ejecutar otro proceso), será el S.O. el encargado de suministrárselo. Para esto deberá existir un medio de comunicación entre el S.O. y los procesos en ejecución.
Una de las maneras que existe para que los **procesos** y el **sistema operativo** puedan comunicarse son las **llamadas al sistema**.

A primer vista las llamadas al sistema lucen como simples funciones las cuales reciben sus parámetros de entrada y entregan un valor de salida, y de hecho lo son, solo que estas funciones son implementadas por el núcleo del S.O. Esto significa que será el S.O. quien recibirá los parámetros de entrada, ejecutará la función que le sea requerida, siempre y cuando sea posible y permitida, y devolverá los valores necesarios, así como también puede o no cambiar el estado de su estructura interna. De esto también podemos deducir que estas llamadas al sistema se ejecutarán en **modo kernel**. De hecho, el procesador está constantemente cambiando de **modo usuario** a **modo kernel** y viceversa (en cualquier S.O. multitarea, como lo es Linux, en el cual se requiere una mayor responsabilidad del S.O. para administrar los procesos, el procesador alterna entre ambos modos al menos unas cuantas miles de veces por segundo).

Ahora bien, **¿para qué son necesarias las llamadas al sistema?**. Como dijimos antes, para que los procesos puedan comunicarse con el *kernel* del S.O. Pero seamos un poco más específicos, las llamadas al sistema nos brindan un medio para obtener recursos del S.O., obtener información del mismo, establecer o cambiar el seteo de los recursos que se ofrecen. Veamos algunos ejemplos de esto:

- En los sistemas UNIX (Linux es uno de ellos) todo los dispositivos son representados mediante archivos, y dado que es el S.O. el encargado de administrar el sistema de archivos, *file system,* (así como los dispositivos, por supuesto), es por medio de éste que podremos utilizar un dispositivo específico como puede ser una lectora de CD, una placa de video, el mouse o el teclado.

- Otro de los recursos administrados por el S.O. es la memoria. Cada vez que un proceso requiera de más cantidad de memoria, o desee liberar parte de la misma que tiene asignada, necesitará usar las llamadas al sistema para informar al núcleo del S.O.

- Hay cosas más simples que solo pueden ser accedidas mediante el S.O. como la hora actual, el nombre de un equipo, hasta el simple almacenamiento de datos en un archivo de texto.

Todas estas tareas no pueden realizarse sin la ayuda del S.O., entonces; *¿qué es lo que se puede hacer sin la ayuda del S.O.?* La respuesta es nada, ya que para poder ejecutar un proceso, necesitamos del S.O., y sin un proceso corriendo no podemos hacer nada. Ahora, supongamos que tenemos a un proceso corriendo en el sistema, que es lo que este puede hacer sin emplear llamadas al sistema, la respuesta será virtualmente la misma. Este podrá ejecutar todas las instrucciones que la arquitectura del procesador permita que sean ejecutadas en **modo usuario** pero nada más. Si tenemos en cuenta que toda comunicación entre el procesador y el mundo exterior se hace mediante dispositivos periféricos (teclado, monitor, placa de red, etc.)

y que es el S.O. el encargado de administrarlos, y por tanto, el único que puede con ellos comunicarse, rápidamente nos daremos cuenta hasta que punto somos dependientes del S.O. para poder llevar a cabo cualquier tipo de tarea.

Para resumir, si alguien nos preguntase que es todo lo que podemos hacer como usuarios de un S.O., podríamos responder que todo lo que uno puede hacer, se encuentra acotado por las instrucciones que la arquitectura del sistema en que trabajamos nos permite ejecutar en **modo usuario**, más las tareas que pueden ser realizadas por medio de llamadas al sistema. Eso es todo, chequeando estos dos parámetros podremos saber que es lo que un S.O. nos permite realizar.

El S.O. Linux posee más de 200 llamadas al sistema, las cuales se encuentran enumeradas en el archivo ***/usr/include/asm/unistd.h***.

Cómo utilizar las llamadas al sistema:

Más del 90% del código de Linux se encuentra escrito en C y si bien una llamada al sistema puede invocarse utilizando cualquier lenguaje de programación, el lenguaje que nosotros utilizaremos será el C, dado que se trata del lenguaje nativo de dicho S.O. Podría parecer que analizar las mismas desde un lenguaje assembler es quizás una aproximación más pura, pero esto es algo falso, dado que Linux funciona sobre un gran número de arquitecturas diferentes, y para cada arquitectura existe una forma distinta de expresar en código máquina las llamadas al sistema, en C estos detalles resultan transparentes al ser un lenguaje de nivel superior menos relacionado con la arquitectura subyacente.
Sin embargo diremos algunas pocas palabras acerca de cómo utilizar las llamadas al sistema en un lenguaje assembler.

Llamadas al sistema en assembler:

Las llamadas al sistema en Linux se realizan mediante la línea de interrupción por software numero **0x80** (80 en hexadecimal), y los parámetros de las mismas se pasan usando los registros del procesador.
En **EAX** se guardará el *número* de la llamda al sistema al cual queremos invocar, estos números están descriptos en ***/usr/include/asm/unistd.h***, luego, los parámetros serán pasados en los registros siguientes, **EBX**, **ECX**, **EDX**, **ESI** y **EDI** consecutivamente. Por tanto, el máximo número de parámetros que puede recibir una llamada al sistema es cinco. El valor devuelto por la llamada se almacena en **EAX**.
Para finalizar analizaremos este ejemplo escrito en *NASM* en el cual se realiza una llamada a la función **read**:

Fragmento código (1)

```
mov eax,0x3                          (1)
mov ebx,0                            (2)
mov ecx,edi                          (3)
mov edx,0x1f                         (4)
int 0x80
```

Recordemos el prototipo de la función **read**:

```
ssize_t read(int fd, void *buf, size_t count);
```

En caso de ignorar el funcionamiento de la función **read** (o de otra llamada al sistema), pueden consultarse las páginas del manual de Linux, utilizando el comando:

```
$ man 2 read
```

Aquí se especifica la búsqueda de información sobre la función **read** en la sección 2 del manual. Podemos obtener información de las llamadas al sistema también haciendo uso del comando:

```
$ man 2 syscalls
```

Y luego consultando la página individual de cada una de las llamadas.

La función **`read`** lee del archivo descripto por **`fd`** hasta un máximo de **`count`** bytes, los guarda a partir de la dirección de memoria **`buf`** y retorna el número de bytes leídos.
En el Fragmento de código (1) anterior, inicialmente se carga en **EAX** *(1)* el número que describe a la función que queremos llamar (el 3 es el número asociado a la función **read**). Luego pasamos como primer parámetro el valor 0 *(2), en **EBX***, que es un descriptor de archivo generalmente referido a la entrada estándar (*stdin*), el

teclado. Luego le pasamos como segundo parámetro el contenido de **EDI** *(3)*, en **ECX**, que es una dirección en donde deberá guardar los bytes leídos, y por ultimo *(4)*, en **EDX**, le pasamos el número máximo de bytes que queremos que lea, en este caso 0x1f, o sea 31. Por último llamamos a la interrupción por software numero **0x80**, que es por medio de la cual se realizan las llamadas al sistema en Linux. Al retornar la ejecución del programa, luego de haberse ejecutado la interrupción, tendremos a partir de la dirección apuntada por **EDI** los bytes leídos, y en **EAX** cuántos han sido estos.

NOTA
Esto es todo lo que hablaremos de assembler en este documento, de ahora en adelante todo código estará expresado en C.

Llamadas al sistema en C:
Veamos ahora como sería la misma llamada pero implementada en C:

```
/* read.c – Llamada al sistema read en C */
#include <unistd.h>

int main(){
        char buf[32];
        int result;

        result = read(0,buf,sizeof(buf));
        return result;
}
```

Cada vez que el intérprete de comandos (*shell*) ejecuta un programa asocia al mismo tres descriptores de archivos conocidos como la entrada estándar (***stdin***), salida estándar (***stdout***) y salida de errores (***stderr***), cuyos valores son respectivamente 0, 1 y 2. Por tanto el valor 0 pasado como argumento a la función **read** indica que lea de la entrada estándar. Ésta puede ser tanto el teclado como un archivo de texto dependiendo de como uno ejecute el programa desde el *shell*.

Por la general para realizar una llamada al sistema desde C no se necesita más que incluir las directivas del preprocesador **#include** haciendo referencia a los archivos de cabecera (*headers)* mencionados en la correspondiente página del manual de la llamada y luego simplemente invocar la función que ejecuta la llamada. En este caso <*unistd.h*>.
La *GNU C library*, junto con las demás que provee, incluye funciones que llevan a cabo la tarea de pasar los parámetros al *kernel* y llamar a la interrupción adecuada, de manera transparente para el programador.

A continuación listaremos un programa llamado *mi_date.c* que imita al conocido comando UNIX **date**, y muestra la fecha y hora actual por pantalla:

```
/*  mi_date.c - Muestra la fecha y la hora actual por pantalla  */
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

void print_time () {
     struct timeval tv;
     struct tm* ptm;
     char time_string[40];

     /*Obtenemos la fecha y hora del día y la transformamos en un estructura tm*/
     gettimeofday(&tv, NULL);
     ptm = localtime(&tv.tv_sec);

     /*Utilizando la estructura tm creamos un string con la informacion que deseamos*/
     strftime(time_string, sizeof(time_string),  "%d/%m/%Y %H:%M:%S" , ptm);

     printf( "%s\n" ,time_string);
}

int main() {
    print_time();
    return 0;
}
```

Este programa utiliza la llamada al sistema **gettimeofday** para conseguir la fecha y hora actual, que son obtenidas por el *kernel* consultando el RTC (*real time clock*) de la máquina. La función **gettimeofday** en nuestro ejemplo toma como primer parámetro un puntero a una estructura **struct timeval**, el segundo argumento es un puntero a una estructura *timezone* que aquí no es utilizada (NULL).

```
struct timeval {
    __time_t tv_sec;        /* Seconds      */
    __suseconds_t tv_usec;  /* Microseconds */
};
```

La estructura *timeval* es instanciada por el *kernel* con los valores actuales de tiempo, la misma contiene la cantidad de segundos y microsegundos (*timestamp*) que han transcurrido desde el comienzo de la llamada ***UNIX Epoch***, la medianoche del primero de enero de 1970. Como esta representación del tiempo no es muy cómoda para la gran mayoría de los humanos, utilizamos las funciones de la librería **libc** *localtime* y *strftime*, las cuales son utilizadas para ofrecer un formato más legible para la fecha y hora actual. Pueden consultarse las *man pages* de estas funciones con los comandos:

```
$ man 3 localtime
$ man 3 strftime
```

Veamos otro ejemplo de una llamada al sistema. Se trata de un programa que muestra por pantalla información acerca del sistema. La llamada al sistema que utiliza, **sysinfo**, establece los valores de una estructura de tipo **struct sysinfo** la cual se encuentra definida en ***/usr/include/linux/kernel.h***.

```
/* stats.c - Muestra estadísticas del sistema respecto al uso de memoria y los procesos
             en ejecución. */

#include <stdio.h>
#include <linux/kernel.h>
#include <linux/sys.h>
#include <sys/sysinfo.h>

int main () {
    const long minute = 60;
    const long hour   = minute * 60;
    const long day    = hour * 24;
    const double megabyte = 1024 * 1024;
    struct sysinfo si;

    /* Obtenemos estadísticas del sistema */
    sysinfo(&si);

    /* Mostramos algunos valores interesantes contenidos en la estructura sysinfo. */
    printf("Tiempo que lleva el sist. en funcionamiento: %ld dias , %ld:%02ld:%02ld\n",
            si.uptime/day,
           (si.uptime % day) / hour,
           (si.uptime % hour) / minute,
            si.uptime % minute);

    printf("Memoria RAM total: %5.1f Mb\n" , si.totalram / megabyte);
    printf("Memoria RAM libre: %5.1f Mb\n" , si.freeram / megabyte);
    printf("Cantidad de procesos corriendo: %d\n" , si.procs);

    return 0;
}
```

Tipos de llamadas al sistema:
Si bien no existe ninguna definición formal de esto, las llamadas al sistema pueden ser agrupadas en ciertos grupos según las funcionalidades que las mismas ofrecen. Sin dar más detalles enumeraremos estos grupos.

NOTA
Salvo para el caso de las llamadas asociadas al manejo de archivos, no existe en el diseño del núcleo nada que refleje explícitamente esta agrupación que haremos.

Acceso a archivos:
Estas llamadas tienen como objetivo el leer, escribir, abrir, cerrar, etc., archivos. Algunas de estas son:

open	**close**
write	**read**
lseek	**readdir**
ioctl	**fsync**
flock	**mmap**

Como ya dijimos, un archivo, bajo Linux, puede estar representando muchas cosas distintas. Puede ser un simple archivo de datos, que almacena información, puede representar un dispositivo de entrada/salida, como el mouse o el teclado, o puede ser simplemente información almacenada en la memoria que utiliza el *kernel* para guardar información, como lo son los archivos contenidos en el directorio */proc*.

Antes de poder usar cualquier archivo, un proceso necesita abrir el mismo, esto se lleva a cabo con la función **open**. La misma devuelve al proceso un descriptor de archivo (***file descriptor***) que en realidad no es más que un número que identifica unívocamente al archivo solicitado. El resto de la funciones necesitarán este descriptor para saber sobre que archivo deben ejecutarse.

La función **write** escribe a un determinado archivo y la función **read** lee del mismo. La semántica de estas funciones depende del tipo de archivo con el cual se esté trabajando.
Por ejemplo, los archivos */dev/tty1, /dev/tty2*, etc. representan consolas virtuales, como las que usamos para

trabajar de ordinario con el *bash* por ejemplo. Escribir a */dev/tty1* significará mostrar por pantalla en esa consola los caracteres que escribamos. Escribir a un archivo normal del S.O. (normal por decir de alguna manera) significará guardar información en el disco rígido. Leer del archivo */dev/mouse* (solo el root tiene permiso) devolverá, dependiendo del tipo de *mouse* con el cual trabajemos, una serie de bytes que representa las coordenadas de los movimientos del *mouse*. Leyendo el archivo */proc/interrupts* obtendremos un listado de las interrupciones que está atendiendo el sistema, y qué *driver* se encarga de atenderla a cada una, además de otros datos.

Veamos un ejemplo, el programa se llamara ***ej1.c***, el mismo tomará dos opciones por línea de comandos: el nombre del archivo, y a partir de que carácter leer. Luego mostrará por pantalla hasta 31 caracteres después del carácter seleccionado.

```
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc , char *argv[]){
    int fd,c,i=0;
    int result;
    int offset;
    char buf[32];

    /* chequeamos que se pasen los argumentos necesarios */
    if(argc < 3){
        printf("Uso: %s <file> <nro. de caracteres a mostar>\n", argv[0]);
        exit(1);
    };

    /* chequeamos que el tercer argumento (2-esimo en argv) sea un digito */
    while(c = argv[2][i]){
        if(!isdigit(c)){
            printf("error! Debe ingresar un numero de digitos ");
            printf("como segundo argumento\n");
            exit(1);
        }
        i++;
    }
    /* verificado que es un numero, convierto el argumento (char*)->(int) */
    offset = aoti(argv[2]);

    if(fd = open(argv[1],O_RDONLY)) == -1)                          /*(1)*/
        perror("error!");
    if(lseek(fd, offset, SEEK_SET) == (offset-1))                   /*(2)*/
        perror("error!");
    if((result=read(fd, buf, 31)) == -1) {                          /*(3)*/
        perror("error!");
        exit(1);
    }

    buf[result] = '\0';
    printf("char %d: %s\n", offset, buf);
    close(fd);

    return 0;
}
```

Las tres llamadas al sistema aquí utilizadas son **open**, **lseek** y **read**, la primera **(1)**, se utiliza para abrir un archivo, la segunda **(2)**, es utilizada para acomodar el puntero interno de un archivo a un valor determinado, y la tercera, **(3)** se utiliza para leer bytes desde un archivo (ver las *man pages* de cada una de las funciones para más detalles).

Ahora iremos mejorando nuestro programa poco a poco y agregándole funcionalidades. Por ejemplo las líneas:

```
    if (lseek(fd, offset,SEEK_SET) == (offset-1) )
        perror("error!");
    if ((result=read(fd,buf,31)) == -1){
        perror("error!");
        exit(1);
    }
```

Podrían reemplazarse con:

```
    if ((result=pread(fd, buf, 31, offset)) == -1){
        perror("error!");
        exit(1);
    }
```

y resultaría lo mismo dado que la función **pread** toma como cuarto argumento el valor de la posición desde la cual se debe de comenzar a leer el archivo, por tanto podemos suprimir la llamada a **lseek**.

NOTA

El valor de offset que tome **pread**() será siempre contando desde el principio del archivo, mientras que la función **lseek**() podrá acomodar este para que sea relativo tanto desde el principio, desde el final, como desde la posición actual.

Ahora podríamos hacer que lea hasta *n* caracteres desde el punto que deseemos iniciar la lectura. También haremos que en lugar usar **printf**(), utilice la llamada **write** para escribir el mensaje en pantalla.

```
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* continua en proxima pagina... */
int isDigit(char *s) {
    while(c = *s[i]){
        if(!isdigit(c)){
            printf("error!: el parametro %s no es un numero!\n", s);
            exit(1);
        }
       i++;
}

int main(int argc , char *argv[]){
    int fd;
    char buf[32];
    int result, offset, cuantos;

    /* chequeamos que se pasen los argumentos necesarios */
    if(argc < 4){
        printf("Uso: %s <archivo> <nro. de caracter de partida> \
                        <cuantos caracteres mostrar>\n", argv[0]);
        exit(1);
    };

    /* chequeamos que el 3er y 4to argumento sea un numero */
    isDigit(argv[2]);
    isDigit(argv[3]);

    offset = atoi(argv[2]);
    cuantos = atoi(argv[3]);

    if(cuantos > 31 ){
        printf("error, sobrepasa el limite del buffer\n");
        exit(1);
    }
    if((fd = open(argv[1],O_RDONLY)) == -1)
        perror("error!");
    if((result=pread(fd,buf,cuantos,offset)) == -1){
        perror("error!");
        exit(1);
    }

    buf[result] = '\0';
    /* Reemplazo de printf("caracter %d: %s\n",offset,buf); */
    {
        char aux[8];
        char mensage[48];
        strcat(mensage,"caracter ");
        sprintf(aux,"%d",offset);
        strcat(mensage,aux);
        strcat(mensage,": ");
        strcat(mensage,buf);
        aux[0] = '\012';  aux[1] = 0 ;      /* para adherir el (\n) */
        strcat(mensage,aux);
        write(1, mensage, strlen(mensage));
    }
    fsync(fd);
```

Como podemos ver, sustituir a **printf**() no es una tarea sencilla. Es verdad que esto podría haberse hecho de una manera mas óptima, pero dado que lo que nos compete es comprender y utilizar llamadas al sistema nos conformaremos con este código. También agregamos a este ejemplo una llamada a la función **fsync**, la cual sirve para sincronizar los datos contenidos en los *buffers* del sistema asociados a este archivo con los del disco rígido, o sea, actualizar los contenidos del disco.

Por último veremos el uso de la llamada al sistema **fstat** para determinar el tamaño de un archivo:

```
int file_size(int fd)
{
        struct stat pts;
        fstat(fd,&pts);
        return pts.st_size;
}
```

La estructura **struct stat** tiene la forma:

```
struct stat {
       dev_t         st_dev;      /* device */
       ino_t         st_ino;      /* inode */
       mode_t        st_mode;     /* protection */
       nlink_t       st_nlink;    /* number of hard links */
       uid_t         st_uid;      /* user ID of owner */
       gid_t         st_gid;      /* group ID of owner */
       dev_t         st_rdev;     /* device type (if inode device) */
       off_t         st_size;     /* total size, in bytes */
       unsigned long st_blksize;  /* blocksize for filesystem I/O */
       unsigned long st_blocks;   /* number of blocks allocated */
       time_t        st_atime;    /* time of last access */
       time_t        st_mtime;    /* time of last modification */
       time_t        st_ctime;    /* time of last change */
};
```

Trabajando con directorios y archivos:

Como ya hemos dicho, una de las tareas más importantes del S.O. es la de administrar el sistema de archivos. Todos sabemos que el sistema de archivos se organiza mediante el uso de un tipo particular de archivos, denominados directorios. Estos tienen como particularidad que en lugar de contener datos, contienen a otros archivos o directorios, o más bien, contienen los datos para poder hallar a estos.

En la sección anterior vimos como acceder a los archivos y como obtener información de ellos (**stat**, **fstat**, **lstat**), ahora veremos como podemos alterar sus atributos y como leer este tipo particular de archivos.

Si uno intenta utilizar la función **read** sobre un archivo que es un directorio, el sistema devolverá un error, esto se debe a que este tipo de archivos cuenta con una función propia para ser leído, esta función es **readdir**, sin embargo esta llamada al sistema esta quedando obsoleta ya que su función ha sido reemplaza por la llamada **getdents**. Sin embargo, no utilizaremos estas llamadas para leer los directorios, en su lugar usaremos *la función* que ofrece la librería estándar de C **readdir()** (más allá de que posea el mismo nombre que la llamada al sistema, esta es una función de librería que utilizando la llamada al sistema homónima brinda la misma funcionalidad).

Veamos ahora como podemos hacer para listar todos los archivos de tipo directorio pertenecientes a un directorio. Puede tomar el directorio desde la linea de comandos o, de no pasársele ningún parámetro, listará los archivos del directorio actual.

```
#include <unistd.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <sys/stat.h>
#include <string.h>

#define BUFSIZE 64

int main(int argc, char **argv) {

    DIR *dd;
    struct dirent* dirp;
    struct stat stats;
    char namebuf[BUFSIZE];
    int name_len;

    if(argc < 2)
        getcwd(namebuf, BUFSIZE-1);
    else
        strcpy(namebuf, argv[1]);

    name_len = strlen(namebuf);

    if((dd = opendir(namebuf)) == 0) {
        perror("Error1");
        exit(1);
    }

    while(dirp = readdir(dd)) {
        namebuf[name_len] = '\0';
        strcat(namebuf,"/");
        strcat(namebuf, dirp->d_name);

        if(stat(namebuf,&stats)) {
            perror("error2");
            exit(2);
        }

        if(S_ISDIR(stats.st_mode))
            printf("%s\n",namebuf);
    }
    return 0;
}
```

La primer gran diferencia que vemos aquí con los ejemplos anteriores es que para abrir el directorio usamos la función **opendir**(), esta es una función de librería, que internamente utiliza la llamada al sistema **open**, pero que esconde esto a nuestros ojos y que en lugar de retornar un *file descriptor* nos devuelve un puntero a dato del tipo DIR. Con este puntero invocaremos a la función **readdir**(), la cual devolverá un puntero a una estructura *dirent*, la cual usaremos para conocer que archivos contiene este directorio. La estructura *dirent* tiene la siguiente forma:

```
struct dirent
{
    long d_ino;               /* número de i-nodos                      */
    off_t d_off;              /* offset respecto al próximo dirent      */
    unsigned short d_reclen;  /* longitud de este dirent                */
    char d_name [NAME_MAX+1]; /* nombre de archivo (terminado en null) */
}
```

Usando el campo *d_name* podemos conocer el nombre de cada archivos que contiene el directorio. Luego, para

saber si el archivo en cuestión se trata de un directorio, usamos la función **stat**. La macro *S_ISDIR,* definida en ***/usr/include/sys/stat.h***, retorna "verdadero" si el archivo en cuestión es un directorio. Otras macros similares pueden ser consultadas a través de la página del manual de **fstat**.

Otras llamadas al sistema, útiles para modificar ciertos aspectos de los archivos, son:

- **chown** cambia los datos relacionados con el dueño de un archivo.
- **chmod** cambio los permisos de acceso a un archivo.
- **rename** cambia el nombre de un archivo.

Otras llamadas al sistema

Ahora listaremos brevemente algunas llamadas al sistema que pueden ser de utilidad para la práctica:

getrlimit y **setrlimit**
: Estas llamadas al sistema sirven para establecer tanto como para consultar los valores actuales y máximos de los recursos del sistema que el proceso que ejecuta la llamada consume, o puede llegar a consumir del sistema.

quotact
: Esta llamada sirve para manipular la cantidad de espacio de disco rígido que un usuario o grupo puede utilizar

getrusage
: Sirve para obtener estadísticas acerca del proceso que efectúa la llamada

mlock
: Sirve para evitar que un sector de memoria sea guardado en el disco debido al uso de memoria virtual y paginación.

nanosleep
: Sirve para dormir un proceso por un tiempo determinado, es sumamente preciso

sendfile
: Sirve para copiar en forma óptima datos de un archivo a otro

uname
: Sirve para obtener información útil acerca de la máquina, el nombre y versión del sistema

---

Nota final

Se recomienda fuertemente ejecutar los programas aquí expuestos y proponerse ejecicios que respondan a modificaciones de los códigos aquí escritos.
