INTRODUCCIÓN A LAS LLAMADAS AL SISTEMA OPERATIVO UNIX

Introducción de llamadas al sistema operativo UNIX

ÍNDICE

1.- Introducción

En este documento se van a analizar algunas llamadas al Sistema Operativo UNIX.

En un primer apartado, se hará un repaso de los conceptos generales de UNIX, nombrando los tipos de ficheros que existen, definiendo programas y procesos, los identificadores y los permisos asociados. Estos son conceptos que ya se vieron en las prácticas de *Introducción básica al Sistema Operativo UNIX a nivel de usuario*.

En apartados posteriores, se tratará el formato de las llamadas al sistema y toda la documentación relacionada con estas llamadas. Se clasificarán en *Llamadas de acceso a ficheros, Llamadas de control de procesos, Interrupciones software: señales,* y *Comunicaciones entre procesos*.

2.- Conceptos generales

Un sistema operativo es el programa o conjunto de programas que permiten gestionar los recursos del hardware de un sistema informático como son la memoria, dispositivos de almacenamiento de datos, terminales, etc. Los diferentes programas que se desarrollan en una máquina se basan en las peticiones de servicios a dicho sistema operativo.

UNIX como sistema operativo ofrece servicios para manejar la memoria, la planificación de los procesos, la comunicación entre los mismos, y operaciones de entrada y salida. Los procesos o programas pueden solicitar dichos servicios directamente al **KERNEL** (llamadas al sistema), o indirectamente a través de rutinas de librería. La comodidad de utilizar el lenguaje C viene del hecho de que se puede utilizar cualquier llamada al sistema a través de las bibliotecas de funciones de que se dispone.

Ficheros

Los ficheros en UNIX se organizan en lo que se conoce como **file systems** (sistemas de ficheros), que son agrupaciones de ficheros bien por razones físicas o lógicas. Hay tres tipos de ficheros en UNIX:

• fichero ordinario: conjunto de bytes en un ordenamiento secuencial. Cualquier byte de dicho ordenamiento puede ser leído y escrito. Tan solo se puede modificar el fichero añadiendo o eliminando bytes del final del mismo. Un fichero ordinario se identifica por un número denominado **i-number**. Un i-number es un índice a una tabla de **i-nodos** que se mantiene al comienzo de cada sistema de ficheros. Cada **i-nodo** contiene información del tipo: tipo de fichero, propietario, grupo, permisos, etc.

• directorio: permite utilizar nombres lógicos para referirnos a todos los ficheros. Consiste en una tabla de dos campos: el primer campo contiene el nombre de un fichero, y el segundo campo su correspondiente i-number. Cada registro de dicha tabla (nombre, i-number) se conoce como **enlace (link)**. Cuando se solicita al sistema operativo el acceso a un fichero referenciándolo por su nombre, éste busca en un directorio para encontrar el i-number. Entonces accede al inodo asociado, donde se encuentra la información asociada al fichero y las direcciones de dónde se encuentra en el disco.

• fichero especial: puede ser de dos clases, FIFO y dispositivo.

FIFO (first-in-first-out queue) es un mecanismo de intercambio de datos entre procesos.

Dispositivo (device) es cualquier dispositivo hardware.

Los ficheros especiales también tienen un i-nodo asociado, pero dicho i-nodo no apunta a ningún bloque en disco donde se contienen datos sino que se hace referencia a una tabla que usa el kernel donde se encuentran unas rutinas llamadas driver. El driver sirve para la comunicación con los terminales. Si se trata de un proceso de entrada de datos desde un dispositivo externo, los datos son pasados a un interface y es el driver el que realiza un procesamiento de éstos y entrega los datos al programa. Si por el contrario, desde el proceso se envían los datos, el driver procesa los datos de salida y los envía al interface.

Programas y procesos

Se entiende por **programa** al conjunto de instrucciones y datos que se encuentran en un fichero normal en el disco. El i-nodo asociado está marcado como ejecutable.

Un programa en ejecución se conoce como **proceso**. UNIX distingue dentro de un proceso tres regiones básicas: texto, datos y pila. Texto es el conjunto de bytes que se interpretan como instrucciones por la CPU. Los datos son tratados por el procedimiento definido en el programa y pila es la zona dinámica de la memoria de un proceso que permite la implementación de rutinas y paso de parámetros. Varios procesos pueden compartir algunas de estas regiones.

Un proceso se puede encontrar en diferentes estados, como listo para ejecutarse (ready to run), ejecutándose (running), o durmiendo (sleeping).

El **system data segment** (segmento de datos del sistema) de un proceso consiste en una serie de información que mantiene el kernel para controlar la ejecución de los procesos. Esta información (process id, parent process id, process group id, current directory, etc) se encuentra repartida en diferentes tablas del sistema.

Identificadores de procesos y grupos

Todos los procesos en UNIX tienen un número positivo que los identifica, el **process-ID (pid)**. Todos los procesos excepto algunos del sistema tienen un proceso padre. El identificador del proceso padre se conoce con el nombre de **parent process-ID (ppid)**.

Si el padre de un proceso muere (termina), éste es adoptado por el proceso **init** (pid=1).

Permisos

Cada usuario del sistema tiene asociado un número positivo llamado **user ID**. Los usuarios pueden estar organizados en grupos, cada grupo tiene una identificación conocida con el nombre de **group ID**.

Cuando un usuario entra en el sistema se leen los ficheros en donde se configuran los usuarios y los grupos y se asocian al usuario las identificaciones de usuario y grupo.

Cada fichero en su i-nodo contiene información del user ID y del group ID del propietario. Así mismo, contiene los conjuntos de permisos que ya se indicaron en la *Introducción básica al Sistema Operativo UNIX a nivel de usuario.* Estos conjuntos de permisos son: usuario, grupo y otros, y dentro de cada uno de ellos, permisos de lectura, escritura y ejecución.

3.- Formato general de las llamadas al sistema

Las llamadas al sistema se pueden usar por medio de funciones en C y todas ellas tienen un formato común, tanto en su documentación como en la secuencia de llamada.

Una llamada al sistema se invoca mediante una función retornando ésta siempre un valor con información acerca del servicio que proporciona o del error que se ha producido en su ejecución. Aunque dicho retorno puede ser ignorado, es recomendable siempre testearlo.

Retornan un valor de -1 como indicación de que se ha producido un error en su ejecución. Además existe una variable externa `errno`, donde se indica un código de información sobre el tipo de error que se ha producido. En el archivo de include `errno.h` hay declaraciones referentes a esta variable.

La variable `errno` no cambia de valor después de una llamada al sistema que retorna con éxito, por tanto es importante tomar dicho valor justo después de la llamada al sistema y únicamente cuando éstas retornan error.

El formato general de documentación de una llamada al sistema es:

```
   CREAT(2)
        Crear un fichero nuevo.
   SYNOPSIS
        #include <sys/types.h>
        #include <sys/stat.h>
        #include <fcntl.h>

        int creat(const char *pathname, mode_t mode);

   DESCRIPTION
        Creat crea un fichero regular nuevo, o prepara para
volver a escribir uno ya existente.
        .......
   ERRORS
        [ENOSPC] no hay suficiente espacio en el sistema de
ficheros.
        [EACCES] no se tiene permiso de acceso.
        ......
   RETURN  VALUE
        Si no hay error retorna el file descriptor, si hay
algún error retorna un -1 y se guarda en la variable errno la
causa del error.
   SEE  ALSO
        chmod(2), close(2),...
```

Nota: La información incluida en este manual, referente a llamadas al sistema, no pretende ser, ni mucho menos, exhaustiva ni completa. Se recomienda recurrir siempre al manual en línea (`man`) antes de emplear cualquiera de ellas. Se recuerda que la sección 2 del manual es la dedicada a llamadas al sistema.

4.- Llamadas de acceso a ficheros

En UNIX, los procesos acceden a los ficheros por medio de **file descriptors (descriptores de fichero)**. Un file descriptor es un índice a una tabla de descriptores de fichero **(file descriptor table)**, que mantiene el kernel para el proceso. Asociada a esa entrada en la tabla hay información referente al modo en que se está empleando ese fichero, como puede ser la posición en el fichero a partir de la cual se realizará la próxima operación de lectura/escritura que se ejecute sobre ese descriptor o el i-nodo del fichero.

Generalmente, cada proceso comienza con tres file descriptors, que hacen referencia al terminal del proceso:

STDIN_FILENO (0)	standard input
STDOUT_FILENO (1)	standard output
STDERR_FILENO (2)	standard error

Dentro de las llamadas al sistema para el acceso a ficheros destacan las siguientes: `open`, `read`, `write`, `close`, `lseek`, `unlink`.

A continuación se va a analizar con detalle la estructura de cada una de ellas.

4.1.- Abrir un fichero (open)

Permite abrir o crear un fichero para lectura y/o escritura. Esta llamada al sistema crea una entrada en la tabla de descriptores de fichero del proceso, retornando al mismo el descriptor asignado, el cual suele ser el primero que encuentre libre en la tabla.

```
OPEN(2)

SINOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>
     #include <fcntl.h>

     int open(const char *pathname, int flags [, mode_t
mode]);

DESCRIPCIÓN
     pathname apunta al nombre de un fichero. La llamada
open abre el fichero designado por path y devuelve el
descriptor de fichero asociado.
     flags se utiliza para indicar el modo de apertura del
fichero. Dicho modo se construye combinando los siguientes
valores:
     O_RDONLY: modo lectura.
```

O_WRONLY: modo escritura.
O_RDWR: modo lectura y escritura.
O_CREAT: si el fichero no existe, lo crea con los permisos indicados en perms.
O_EXCL: si está puesto O_CREAT y el fichero existe, la llamada da error.
O_APPEND: el fichero se abre y el offset apunta al final del mismo. Siempre que se escriba en el fichero se hará al final del mismo.
mode es opcional y permite especificar los permisos que se desea que tenga el fichero en caso de que se esté creando.

RETORNO
Retorna el file descriptor del fichero o -1 si hay error.

Existe una llamada al sistema que produce el mismo efecto que el `open` con los flags: O_RDWR | O_CREAT. Esta llamada es `creat` y su sinopsis es:

```
int creat(const char *pathname, mode_t mode);
```

4.2.- Lectura de un fichero (`read`)

La llamada `read` se utiliza para leer un número dado de bytes de un fichero. La lectura comienza en la posición señalada por el descriptor, y tras ella se incrementa ésta en el número de bytes leídos.

READ (2)

SINOPSIS

```
#include <unistd.h>

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

DESCRIPCIÓN
La llamada read lee un número de bytes dado por count del fichero al que hace referencia el descriptor de fichero fd y los coloca a partir de la dirección de memoria apuntada por buf.

RETORNO
Retorna el número de bytes leídos, 0 si encuentra el final del fichero y -1 si hay error.

4.3.- Escritura de un fichero (`write`)

La llamada `write` se emplea para escribir un número de bytes en un fichero. La escritura comienza en la posición señalada por el descriptor y tras ésta se incrementa la posición en el número de bytes escritos.

La escritura se realiza en una memoria caché del sistema y el kernel se encarga de volcar dicha memoria al fichero en disco o en el dispositivo.

Tiene el siguiente formato:

```
WRITE (2)

SINOPSIS
     #include <unistd.h>

     ssize_t write(int fd, const void *buf, size_t count);

DESCRIPCIÓN
     La llamada write escribe un número de bytes dado por
count en el fichero cuyo file descriptor viene dado por fd. Los
bytes a escribir deben encontrarse a partir de la posición de
memoria indicada en buf.

RETORNO
     Retorna el número de bytes escrito o -1 si hay un
error.
```

4.4.- Cerrar un fichero (`close`)

La llamada close deshace el enlace entre un descriptor de fichero y su fichero. La entrada en la tabla de descriptores de fichero del proceso queda disponible para volver a ser utilizada. Su descripción es la siguiente:

```
CLOSE (2)

SINOPSIS
     #include <unistd.h>

     int close(int fd);

DESCRIPCIÓN
     Close cierra un fichero.

RETORNO
     Retorna 0 si no hay error y -1 si hay algún error.
```

4.5.- Posicionamiento en un fichero (lseek)

La llamada lseek establece la posición señalada por un descriptor de fichero, la cual será empleada en la próxima llamada a read / write.

LSEEK (2)

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPCIÓN

Establece la posición señalada por el descriptor de fichero de la siguiente forma:

. si whence es SEEK_SET, el puntero al fichero apunta a la dirección de offset.

. si whence es SEEK_CUR, el puntero al fichero apunta a la dirección actual más offset.

. si whence es SEEK_END, el puntero al fichero apunta a la longitud del fichero más offset.

RETORNO

Retorna la nueva posición señalada por el descriptor de fichero si no hay error y -1 si hay algún error.

4.6.- Destruyendo entradas en los directorios (unlink)

Borra una entrada en la tabla de un directorio.

UNLINK (2)

SINOPSIS

```
#include <unistd.h>

int unlink(const char *pathname);
```

DESCRIPCIÓN

Borra una entrada en la tabla de un directorio. pathname representa un fichero ya existente cuya entrada se quiere borrar. Si se elimina la última entrada existente en algún directorio que hace referencia a un i-nodo concreto el sistema elimina el fichero de ese i-nodo.

RETORNO

Retorna 0 si no hay error y -1 si hay algún error.

Programa ejemplo: Copia de fichero

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
      int fdold, fdnew;

      if (argc!=3)
      {
           fprintf(stderr, "Se precisan 2 argumentos\n");
           exit(1);
      }

      fdold=open(argv[1], O_RDONLY);
      if (fdold==-1)
      {
      fprintf(stderr, "No se pudo abrir el fichero %s\n",
argv[1]);
      exit(1);
      }

      fdnew=creat(argv[2], 0666);
      if (fdnew==-1)
      {
      fprintf(stderr, "No se pudo crear el fichero %s\n",
argv[2]);
      exit(1);
      }

      copy(fdold, fdnew);
      exit(0);
}

copy(int old, int new)
{
      int cuenta;
      char    buffer[2048];

      while ((cuenta=read(old, buffer, sizeof(buffer)))>0)
           write(new, buffer, cuenta);
}
```

5.- Llamadas de control de procesos

5.1.- Conceptos generales

Un proceso se puede entender en parte como un programa en ejecución. Sus características generales son las siguientes:

- Un proceso consta de código, datos y pila.
- Los procesos existen en una jerarquía de árbol (varios hijos, un solo padre).
- El sistema asigna un identificador de proceso (PID) único al iniciar el proceso.
- El planificador de tareas asigna un tiempo de empleo de la CPU para el proceso según su prioridad.

5.2.- Ejecución de programas (exec)

Un programa (fichero ejecutable) es ejecutado cuando un proceso hace una llamada `exec` al sistema. El kernel sustituye los segmentos de texto y de datos del proceso que realiza la llamada por los del fichero ejecutable que se le pasa como parámetro en la llamada. El proceso continúa su ejecución en la primera línea del programa ejecutado. Es decir, el proceso sustituye el programa que está ejecutando por otro. Una vez completada con éxito la llamada `exec()` el código del antiguo programa deja de ejecutarse (desaparece ya que es sustituido por el del programa ejecutado) y se pasa al nuevo código de programa. Sin embargo el proceso sigue siendo el mismo, es decir, tiene los mismos identificadores de proceso (PID), de proceso padre (PPID) y de grupo de procesos (PGID), la misma tabla de descriptores de ficheros, mantiene el directorio actual, etc.

La llamada `exec` tiene variantes. Generalmente una de ellas es una llamada al sistema y el resto son funciones de biblioteca que permiten pasar los parámetros de forma más cómoda pero que internamente emplean la llamada al sistema

EXEC (2)

SINOPSIS

```
     #include <unistd.h>

     int execl(const char *path, const char *arg, ...);
     int execlp(const char *file, const char *arg, ...);
     int execle(const char *path, const char *arg, ..., char
* const envp[]);
     int execv( const char *path, char *const argv[]);
     int execvp( const char *file, char *const argv[]);
```

DESCRIPCIÓN

Las 6 funciones tienen la misma funcionalidad de ejecutar un fichero ejecutable.

La diferencia entre las 6 son la forma de pasar los parámetros de entrada.

Los nombres de estas funciones están compuestos por exec y uns serie de letras que tienen el siguiente significado:

l : Los argumentos para el programa (cadenas de texto) se incluyen uno a uno en los argumentos de la función, indicando el último mediante el puntero 0.

v : Los argumentos para el programa se pasan mediante un array de punteros a las cadenas de texto que son los argumentos. El último elemento del array debe ser el puntero 0.

e : Permite pasar las variables globales de entorno que deben definirse para la ejecución del nuevo programa.

p : La búsqueda del programa se hará en todos los directorios contenidos en la variable de entorno PATH.

RETORNO

Retornan -1 si hay error. Si se ejecuta con éxito no devuelve ningún valor dado que el código de programa se sustituye por el del nuevo programa y se eliminan todas las variables del programa.

Argumentos de un programa

Cuando ejecutamos un programa desde una Shell podemos especificar parámetros a los que queremos que éste tenga acceso. ej:

```
%> cp fichero1 fichero2
```

[cp es el nombre del programa, fichero1 y fichero2 son los parámetros o argumentos]

Un programa escrito en C puede estar preparado para acceder a estos argumentos. En la declaración de la función `main()` se pueden incluir varias variables:

```
main(int argc, char *argv[], char *environ[])
```

argc es un entero que cuenta el número de argumentos que posee el programa

argv es una array de punteros, los cuales hacen referencia a las cadenas de texto donde están los argumentos. El primer argumento es el nombre del ejecutable (argv[0]), es decir, el nombre del programa cuenta entre los argumentos, por lo que argc valdrá al menos 1.

environ es análogo a argv pero contiene las variables de entorno

5.3.- Crear un proceso (`fork`)

Los procesos se crean a través de la llamada al sistema `fork`. Cuando se realiza dicha llamada, el kernel duplica el entorno de ejecución del proceso que llama, dando como resultado dos procesos.

El proceso original que hace la llamada se conoce como *proceso padre,* mientras que el nuevo proceso resultado de la llamada se conoce como *proceso hijo*.

La diferencia en los segmentos de datos de ambos procesos es que la llamada `fork` retorna un 0 al proceso hijo, y un entero que representa el PID del hijo al proceso padre. Si la llamada fracasa, no se crea el proceso hijo y se devuelve -1 al proceso padre.

Una vez ejecutada con éxito la llamada `fork` y devueltos los valores de retorno ambos procesos continúan su ejecución a partir de la siguiente instrucción al fork.

```
FORK (2)

SINOPSIS
     #include <unistd.h>

     pid_t fork(void);

DESCRIPCIÓN
     fork causa la creación de un nuevo proceso copia (casi
exacta) del proceso padre.

RETORNO
     si la llamada tiene éxito retorna:
     0         al proceso hijo
     pid del hijo   al proceso padre
     si fracasa devuelve -1
```

5.3.1.- Programa ejemplo de utilización de `fork`

```
/* fork.c - Ejecución conjunta de procesos padre e hijo */
#include <stdio.h>
#include <unistd.h>
main ()
    {
    pid_t  pid;
    printf ("Ejemplo de fork.\n");
    printf ("Inicio del proceso padre. PID=%d\n", getpid ());

    pid=fork();
    if (pid == 0)
        {       /* Proceso hijo */
        printf ("Proceso hijo. PID=%d, PPID=%d\n", getpid (),
getppid ());
        sleep (1);
        }
    else
        {       /* Proceso padre */
        printf ("Proceso padre. PID=%d\n", getpid ());
        sleep (1);
        }
    printf ("Fin del proceso %d\n", getpid ());
    exit (0);
    }
```

5.4.- Espera de la terminación de un proceso (`wait`)

Cuando un proceso termina se envía un *valor de retorno* al proceso padre de éste.

Si un proceso ha creado mediante la llamada `fork` uno o varios hijos, la llamada `wait` suspende el proceso padre hasta que alguno de sus hijos termine su ejecución y devuelve dicho valor.

Si un proceso padre muere o termina antes que alguno de sus hijos éstos son *adoptados* por el proceso `init`, cuyo PID es 1. Es decir, a partir de ese momento el PPID de todos esos procesos es 1.

Durante el tiempo entre que un proceso muere y que su padre recoge ese valor de retorno el proceso hijo se considera un proceso *zombi*. El proceso zombi, aunque ya no consume tiempo de CPU sigue existiendo en la tabla de procesos de la máquina y no desaparecerá hasta que su proceso padre recoja su código de retorno.

WAIT (2)

SINOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

DESCRIPCIÓN

La llamada wait espera hasta que muera un proceso hijo. En el parámetro statusp se devuelve el status de salida del proceso hijo.

RETORNO

Retorna el pid del proceso hijo que ha terminado o -1 si hay algún error.

5.5.- Terminación de un proceso (exit)

La llamada `exit` finaliza la ejecución de un proceso indicando el status de finalización del mismo.

EXIT (2)

SINOPSIS

```
void exit (int status)

int status;
```

DESCRIPCIÓN

La llamada exit finaliza la ejecución de un proceso.

6.- Interrupciones software: señales

6.1.- Conceptos generales

Una señal es una interrupción software, un evento que debe ser procesado y que puede interrumpir el flujo normal de un programa.

Las señales en UNIX son el mecanismo que ofrece el kernel para comunicar eventos de forma asíncrona. Pero el kernel no es el único que puede enviar señales; cualquier proceso puede enviar a otro proceso una señal, siempre que tenga permiso.

Una alarma es una señal que es activada por los temporizadores del sistema.

Cuando un proceso se prepara para la recepción de una señal, puede realizar las siguientes acciones:

- Ignorar la señal
- Realizar la acción asociada por defecto a la señal
- Ejecutar una rutina del usuario asociada a dicha señal.

6.2.- Algunas señales

Nombre	Comentarios
SIGHUP	Colgar. Generada al desconectar el terminal.
SIGINT	Interrupción. Generada por teclado.
SIGILL	Instrucción ilegal. No se puede capturar.
SIGFPE	Excepción aritmética, de coma flotante o división por cero.
SIGKILL	Matar proceso. No puede capturarse, ni ignorarse.
SIGBUS	Error en el bus.
SIGSEGV	Violación de segmentación.
SIGPIPE	Escritura en una pipe para la que no hay lectores.
SIGALRM	Alarma de reloj.
SIGTERM	Terminación del programa.

6.3.- Capturar señales (subrutina `signal`)

La llamada `signal` asocia una acción determinada con una señal.

Su descripción es la siguiente:

SIGNAL (2)

SINOPSIS

```
       #include <signal.h>

       void (*signal(int signum, void (*handler)(int)))(int);
```

DESCRIPCIÓN

Especifica la respuesta de un proceso ante una señal.

signum: es el número de señal para la cual se especifica la respuesta.

handler: es el tipo de respuesta deseado:

SIG_DFL: respuesta por defecto.

SIG_IGN: ignorar.

nombre (puntero) de una función que es el handler (función que se ejecuta al recibir dicha señal).

RETORNO

Retorna el puntero de la función que había antes si no hay error y SIG_ERR si hay algún error.

Nota: Hay que destacar que la señal puede interrumpir la ejecución de una función o llamada al sistema. Al finalizar la ejecución del handler puede que continúe la ejecución de la función, se reinicie o termine con un error (depende de la función y de la implementación concreta de UNIX). Otro aspecto a tener en cuenta es que las funciones o llamadas que se empleen en el handler han de ser reentrantes, es decir, no es seguro emplear cualquier llamada en el hanlder (por ejemplo, las funciones `malloc` y `free` no son reentrantes, si se emplean en el handler y éste ha interrumpido otro `malloc` se puede corromper la memoria del programa).

7.- Comunicación entre procesos

Normalmente, las aplicaciones que se desarrollan sobre el sistema operativo UNIX constan de varios procesos ejecutándose de forma concurrente. Por lo tanto surgen necesidades a la hora de la programación:

- Compartir información entre los procesos.
- Intercambio de información entre los procesos.
- Sincronización entre los procesos.

7.1.- Conceptos generales

Un descriptor de fichero es un número entero positivo usado por un proceso para identificar un fichero abierto.

Se llama redireccionar a establecer copias del descriptor de fichero de un archivo para encauzar las operaciones de E/S hacia otro fichero.

7.1.1.- Redirección (subrutinas dup y dup2)

La función de estas llamadas al sistema es la de duplicar un descriptor de fichero. La descripción es la siguiente:

DUP(2)

SINOPSIS

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

DESCRIPCIÓN

Duplica un descriptor de fichero.

dup duplica el descriptor oldfd sobre la primera entrada de la tabla de descriptores del proceso que esté vacía.

dup2 duplica del descriptor oldfd sobre el descriptor newfd. En el cado en que éste ya hiciera referencia a un fichero, lo cierra antes de duplicar.

RETORNO

Ambas rutinas devuelven el valor del nuevo descriptor de fichero y –1 en caso de error.

7.1.1.1.- Ejemplo de programa donde se utilice la llamada dup2

```
  /* dup2.c - Redirección usando dup2 */
  /* Ejecuta el comando que se incluya como segundo argumento
redireccionando su salida estándar hacia el fichero de nombre
el primer argumento */
  #include <stdlib.h>
  #include <unistd.h>
  #include <fcntl.h>

  int main (int contargs, char *args[])
      {
      int desc_fich;

      if (contargs < 3)
         {
          printf ("Formato: %s fichero comando [opciones].\n",
args[0]);
         exit (1);
         }
      printf ("Ejemplo de redirección.\n");
        desc_fich = open (args[1], O_CREAT|O_TRUNC|O_WRONLY,
0666);
      if (desc_fich==-1)
         exit(-1);
      dup2 (desc_fich, STDOUT_FILENO);     /*   Redirige   la
salida estándar */
      close (desc_fich);    /* Cierra el descriptor, ya no es
necesario */
      execvp (args[2], &args[2]);          /*  Ejecuta  comando
*/
      exit (1);
      }
```

7.2.- Pipes

Las técnicas más comunes de comunicación entre procesos son los ficheros, las pipes y las named pipes (FIFOs), presentes desde las primeras versiones de UNIX.

Los ficheros permiten compartir gran cantidad de información a los procesos. La desventaja es la falta de eficiencia debido a la necesidad de acceder al disco.

Con el nombre de pipe o fifo se conoce a un mecanismo de comunicación entre procesos que provee UNIX con las siguientes características: es un “canal” de entrada/salida de datos en el que se puede escribir y leer, permite que 2 o más procesos envíen información a otro.

Vamos a explicar en detalle las pipes.

Las pipes o unnamed fifos sólo pueden ser empleadas entre procesos relacionados (pare-hijo, hijo-hijo). Es el tipo de comunicación que se emplea cuado en una shell se encadenan comando con el carácter “|”.

La creación de una unnamed fifo se realiza mediante la llamada `pipe`. Su descripción es la siguiente:

PIPE (2)

SINOPSIS

```
#include <unistd.h>

int pipe(int filedes[2]);
```

DESCRIPCIÓN

Crea un canal de comunicación. fildes al retorno contiene dos file descriptors, fildes[0] contiene el descriptor de lectura y fildes[1] el de escritura.

La operación de lectura en fildes[0] accede a los datos escritos en fildes[1] como en una cola FIFO (primero en llegar, primero en servirse).

RETORNO

Retorna 0 si no hay error y -1 si hay algún error.

Una vez se dispone de la pipe se puede emplear como descriptores normales, es decir, se puede escribir con `write`, leer con `read` y cerrar cualquiera de ellos con `close`. Al leer del descriptor de la pipe para lectura se obtendrá `EOF` (fin de fichero) cuando se hayan cerrado TODOS los descriptores de escritura que hagan referencia a esa pipe, sean del proceso que sean.

7.2.1.- Ejemplo de utilización de la llamada `pipe`

```
/* pipe.c - pipe entre procesos padre e hijo */
#include <stdlib.h>
#include <unistd.h>

#define LEER            0
#define ESCRIBIR        1

int main ()
    {
    int descr[2];         /* Descriptores de E y S de la pipe
*/
    int  bytesleidos;
    char mensaje[100], *frase="Veremos si la transferecia es
buena.";

    printf ("Ejemplo de pipe entre padre e hijo.\n");

    pipe (descr); /* Crea la pipe */

     if (fork () == 0) /* Crea un nuevo proceso. Este nuevo
proceso tiene una copia de las variables del padre y por tanto
tiene acceso a la pipe mediante dos entradas en SU tabla de
descriptores de fichero */
         { /* Código ejecutado por el hijo. Es el que va a
escribir. */
/* El hijo no va a leer así que cierra su descriptor de
lectura */
        close (descr[LEER]);
        write (descr[ESCRIBIR], frase, strlen(frase)+1);
/* Cierra el descriptor de escritura que ya no necesita para
que el otro proceso sepa que no va a haber más escrituras */
```

```
         close (descr[ESCRIBIR]);
         }
      else
         { /* Código que va a ejecutar el proceso padre */
  /* Cierra el descriptor de escritura dado que no lo va a
emplear. Si no lo cerrase quedaría un descriptor para escribir
en la pipe abierto, por lo que no se leería EOF nunca */
         close (descr[ESCRIBIR]);
         bytesleidos = read (descr[LEER], mensaje, 100);
         printf ("Bytes leidos: %d\n", bytesleidos);
         printf ("Mensaje: %s\n", mensaje);
         close (descr[LEER]);
         }
      }
```

7.3.- Named pipes (FIFOs)

Las named pipes permiten comunicar procesos no relacionados, existen en el sistema de ficheros, han de ser creadas utilizando `mknod` y existen hasta que se borren con `rm`.

Al igual que en las pipes, la lectura y escritura se realiza con las llamadas `read` y `write`. La apertura se realiza mediante la llamada `open`.

Como ventaja sirven para que se puedan comunicar procesos que no estén relacionados.

7.4.- System V IPCs

Se conocen como System V IPCs a tres técnicas de comunicación entre procesos que provee el UNIX System V:

• Memoria compartida: provee comunicación entre procesos permitiendo que éstos compartan zonas de memoria.

• Semáforos: dota a los procesos de un mecanismo de sincronización, generalmente para coordinar el acceso a recursos. Cuando un proceso intenta ocupar un semáforo, éste puede encontrarse en uno de los siguientes estados:

- libre: entonces el kernel permite que ocupe dicho semáforo y el proceso continúa su ejecución.

- ocupado: entonces generalmente el kernel pone el proceso en estado durmiente hasta que se libere el semáforo.

• Colas de mensajes: permiten tanto compartir información como sincronizar procesos. Un proceso envía un mensaje y otro lo recibe. El kernel se encarga de sincronizar la transmisión/recepción.

8.- Otras llamadas

8.1.- Obtener la hora del sistema

•**time** retorna el valor en segundos del tiempo transcurrido desde el 1 de Enero de 1970 a las 00:00:00 GMT

•**gettimeofday** obtiene la hora del sistema teniendo en cuenta la hora local. Trabaja con resolución de microsegundos.

8.2.- Tratamiento de errores

Existe una función de librería que permite mostrar información sobre el último error que se ha producido en un proceso como resultado de una llamada al sistema.

Su descripción es la siguiente:

PERROR (3)

SINOPSIS

```
#include <stdio.h>

void perror(const char *s);
```

DESCRIPCIÓN

La función perror produce un mensaje en la salida de error standard, describiendo el último error producido como resultado de una llamada al sistema o una llamada a una función de librería. Se imprime la cadena s y a continuación el mensaje sobre el error.

8.3.- Otras

- **chdir** Permite cambiar el directorio de trabajo del proceso
- **getenv** Permite acceder al contenido de las variables de entorno del proceso.
- **setenv** Permite modificar las variables de entorno del proceso.

Bibliografía

UNIX For Programmers And Users A Complete Guide, G. Glass, Ed. Prentice Hall, ISBN 0-13-061771-7

Advanced Programming In The UNIX Environment, W. Richard Stevens, Ed. Addison-Wesley, ISBN 0-201-56617-7

Beggining Linux Programming, N.Matthew & R.Stones, Ed.Wrox, ISBN 1-874416-68-0

UNIX Distributed Programming, C. Brown, Ed. Prentice Hall