Sistemas Operativos
2016/2017

Práctica 1

Gestión de procesos y archivos. Comunicación entre procesos: tuberías y memoria compartida

Uno de los objetivos de los sistemas operativos es hacer que el computador sea más fácil de utilizar. Para ello el sistema operativo ofrece un conjunto de operaciones que actúan a diferentes niveles (procesos, archivos, comunicación, etc.). Estas operaciones se pueden clasificar en dos grupos:

- Llamadas al sistema que proporcionan la interfaz a nivel del programador. Son operaciones básicas que facilita el sistema operativo para realizar programas. Por ejemplo, crear un proceso, abrir un archivo, crear un tubo, etc.

- Órdenes o comandos de shell que proporcionan la interfaz a nivel de usuario. Normalmente son programas más elaborados construidos a partir de las llamadas al sistema anteriores. Por ejemplo, copiar un archivo, crear un directorio, imprimir un archivo, etc.

Los sistemas operativos multiproceso permiten ejecutar varios procesos simultáneamente. Para ello deben proporcionar un conjunto de llamadas al sistema que no sólo puedan crear y eliminar procesos, sino también cambiar su imagen, esto es, cambiar su código, memoria, etc.

De la misma forma que el núcleo del sistema operativo trabaja mediante interrupciones para sincronizar-comunicar, a nivel de procesos, el sistema ofrece un mecanismo análogo basado en interrupciones software para trabajar con los procesos a nivel de usuario: las señales.

Otra de las funciones más importantes de los sistemas operativos es gestionar los recursos disponibles en el computador, repartiéndolos de forma adecuada entre los diferentes procesos que puedan estar ejecutándose en cada instante. Entre los recursos que debe gestionar se tiene, por ejemplo, la memoria principal, la memoria secundaria, los dispositivos y la CPU. El sistema de archivos es una de las partes del sistema operativo que se encarga de estudiar la gestión de la memoria secundaria.

El sistema de archivos tiene una gran importancia en el computador. Como es conocido uno de los cuellos de botella en un computador es la gestión del almacenamiento secundario y por ello el cómo implementar los archivos y directorios influirá en el rendimiento del sistema.

Para la mayoría de los usuarios, el sistema de archivos es el aspecto más visible de un sistema operativo, pues proporciona el mecanismo para el almacenamiento tanto de datos como de programas del sistema operativo y de los propios usuarios del sistema informático.

Por último, con el objetivo de que los sistemas informáticos sean eficientes y productivos, el sistema operativo debe ofrecer servicios de comunicación entre procesos. Mediante la comunicación, unos procesos pueden invocar servicios suministrados por otros, permitiendo la especialización y una mayor productividad.

En esta práctica estudiaremos algunas de las llamadas al sistema relacionadas con la creación y terminación de procesos, cambio de imagen de un proceso y gestión de señales (enviar, programar, esperar, etc.). Además, vamos a estudiar las llamadas al sistema más representativas que proporciona el sistema operativo para gestionar el sistema de archivos. Asimismo, se describirá cómo se pueden comunicar procesos empleando tuberías. Hemos incorporado al final de la práctica un anexo que recoge en una tabla las llamadas al sistema Unix. Asimismo, hemos añadido otro anexo que describe sucintamente cómo se compila y ejecuta un programa en lenguaje C que es el que se emplea para desarrollar las prácticas a nivel de programador.

CREACIÓN DE PROCESOS

LLAMADA FORK

`int fork(void)`

En Unix, un proceso es creado mediante la llamada del sistema *fork*. El proceso que realiza la llamada se denomina proceso padre (*parent process*) y el proceso creado a partir de la llamada se denomina proceso hijo (*child process*). La sintaxis de la llamada efectuada desde el proceso padre es: `pid=fork();` La llamada *fork* se realiza una sola vez, pero retorna dos valores (correspondientes a los procesos padre e hijo). Las acciones implicadas por la petición de un *fork* son realizadas por el núcleo (*kernel*) del sistema operativo. Tales acciones son las siguientes:

1. asignación de un hueco en la tabla de procesos para el nuevo proceso (hijo).
2. asignación de un identificador único (PID) al proceso hijo.
3. copia de la imagen del proceso padre (con excepción de la memoria compartida).
4. asignación al proceso hijo del estado "preparado para ejecución".
5. dos valores de retorno de la función: al proceso padre se le entrega el PID del proceso hijo, y al proceso hijo se le entrega el valor cero.

A la vuelta de la llamada *fork*, los dos procesos poseen copias iguales de sus contextos a nivel usuario y sólo difieren en el valor del PID devuelto. La indicada relación de operaciones del *fork* se ejecuta en modo núcleo en el proceso padre. Al concluir, el planificador de procesos pondrá en ejecución un proceso que puede ser:

- el mismo proceso padre: vuelta al modo usuario y al punto en el que quedó al hacer la llamada *fork*.
- el proceso hijo: éste comenzará a ejecutarse en el mismo punto que el proceso padre, es decir, a la vuelta de la llamada *fork*.
- cualquier otro proceso que estuviese "preparado para ejecución" (en tal caso, los procesos padre e hijo permanecen en el estado "preparado para ejecución").

Las dos primeras situaciones se distinguen gracias al parámetro devuelto por *fork* (cero identifica al proceso hijo, no cero al proceso padre).

La ejecución del siguiente programa `creaproc.c` y la posterior petición de información sobre procesos en ejecución permitirá comprobar el funcionamiento de la llamada *fork*.

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


int main()
{
    pid_t pid;

    pid = fork();
    switch (pid)
    {
        case -1:
            printf ("No he podido crear el proceso hijo \n");
            break;
        case 0:
```

```
            printf ("Soy el hijo, mi PID es %d y mi PPID es %d \n",
                    getpid(), getppid());
            sleep (20);
            break;
        default:
            printf ("Soy el padre, mi PID es %d y el PID de mi hijo es %d \n",
                    getpid(), pid);
            sleep (30);
    }
    printf ("Final de ejecución de %d \n", getpid());
    exit (0);
}
```

Para ejecutar el programa en modo background, y así poder verificar su ejecución con la orden `ps`, se debe añadir al nombre del programa el carácter & (ampersand):

```
$ gcc –o creaproc creaproc.c
$ creaproc &
Soy el hijo, mi PID es 944 y mi PPID es 943
Final de ejecución de 944
Soy el padre, mi PID es 943 y el PID de mi hijo es 944
Final de ejecución de 943
$ ps
  PID TTY          TIME CMD
  893 pts/1    00:00:00 bash
  943 pts/1    00:00:00 creaproc
  944 pts/1    00:00:00 creaproc
  946 pts/1    00:00:00 ps
$
```

Dos procesos vinculados por una llamada *fork* poseen zonas de datos propias, de uso privado (no compartidas). La ejecución del siguiente programa, en el cual se asigna distinto valor a una misma variable según se trate de la ejecución del proceso padre o del hijo, permitirá comprobar tal característica de la llamada *fork*.

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

int main ( ) {
     int i;
     int j;
     pid_t pid;

     pid = fork( );
     switch (pid) {
     case -1:
          printf ("\nNo he podido crear el proceso hijo");
          break;
     case 0:
          i = 0;
          printf ("\nSoy el hijo, mi PID es %d y mi variable i
(inicialmente a %d) es par", getpid(), i);
          for ( j = 0; j < 5; j ++ ) {
               i ++;
               i ++;
               printf ("\nSoy el hijo, mi variable i  es %d", i);
```

```
                };
        break;
    default:
        i = 1;
        printf ("\nSoy el padre, mi PID es %d y mi variable i
(inicialmente a %d) es impar", getpid(), i);
        for ( j = 0; j < 5; j ++ ) {
             i ++;
             i ++;
             printf ("\nSoy el padre, mi variable i  es %d", i);
             };
    };
    printf ("\nFinal de ejecucion de %d \n", getpid());
    exit (0);
}
```

En este programa, el proceso padre visualiza los sucesivos valores impares que toma su variable i privada, mientras el proceso hijo visualiza los sucesivos valores pares que toma su variable i privada y diferente a la del proceso padre.

```
$ gcc –o creaproc1 creaproc1.c
$ creaproc1 &

Soy el padre, mi PID es 956 y mi variable i (inicialmente a 1) es impar
Soy el padre, mi variable i  es 3
Soy el padre, mi variable i  es 5
Soy el padre, mi variable i  es 7
Soy el padre, mi variable i  es 9
Soy el padre, mi variable i  es 11
Final de ejecucion de 956

Soy el hijo, mi PID es 957 y mi variable i (inicialmente a 0) es par
Soy el hijo, mi variable i  es 2
Soy el hijo, mi variable i  es 4
Soy el hijo, mi variable i  es 6
Soy el hijo, mi variable i  es 8
Soy el hijo, mi variable i  es 10
Final de ejecucion de 957
$
```

Las variables descriptores, asociadas a archivos abiertos en el momento de la llamada *fork*, son compartidas por los dos procesos existentes a la vuelta del *fork*. Es decir, los descriptores de archivos en uso por el proceso padre son heredados por el proceso hijo generado.

Un proceso no sabe a priori su nombre y ni el de su padre (aunque lo tenga en sus estructuras internas). Para obtener el identificador de un proceso o el de su padre hay que acudir a las llamadas getpid() y getppid().

LLAMADA GETPID

int getpid(void)

La llamada `getpid()` devuelve el número de identificación de proceso (PID) del proceso actual. Devuelve –1 en caso de error y el PID del proceso en caso de éxito.

LLAMADA GETPPID

`int getppid(void)`

La llamada `getppid()` devuelve el número de identificación de proceso (PID) del proceso padre al proceso actual (el PID del proceso que creó al actual). Devuelve –1 en caso de error y el PID del padre en caso de éxito.

TERMINACIÓN DE PROCESOS

LLAMADA EXIT

`void exit(int estado)`

La llamada `exit()` hace finalizar un proceso con estado `estado`. Este estado es un valor entero que se retorna al padre del proceso finalizado. Para que el padre puede leer este valor, deberá realizar una llamada `wait()`.

Si un proceso finaliza sin ejecutar la llamada `exit()`, por ejemplo, al realizar un retorno de la función `main()`, su estado de finalización será indefinido.

El sistema tiene una variable de entorno, denominada `?`, donde puede recogerse el estado de finalización de la ejecución de la última orden. Para ver el estado de finalización del último hijo de shell (última orden) hay que introducir

```
$ echo $?
...
```

A lo que el sistema contestará con 0, 2, o cualquier estado posible.

LLAMADA WAIT

`int wait(int *estado)`

`int wait(NULL)`

La llamada `wait()` hace que el proceso actual quede detenido en espera hasta que muere alguno de sus hijos o recibe una señal.

`wait()` devuelve –1 en caso de error y el PID del hijo que muere en caso contrario.

Un proceso puede finalizar por dos causas: una terminación explícita mediante la ejecución de una llamada `exit()`, o bien porque otro proceso lo mató enviándole una señal.

El parámetro `estado` es un entero en el que `wait()` escribe información dividida en dos partes:

- Si el hijo termina con una llamada `exit(estado)`, en los 7 bits menos significativos `(estado & 0x7F)` escribe 0 y en los 8 bits más significativos `((estado>>8) & 0xFF)` escribe el valor `estado`.
- Si el hijo termina matado por una señal, en los 7 bits menos significativos `(estado & 0x7F)` escribe el número de señal.


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

int main ( ) {
   int estado, numero;

   switch(fork()) {
      case -1 : /* ERROR */
                perror("Error en fork");
                exit(1);
      case 0  : /* HIJO */
                numero = 13;
                printf("Soy el hijo y muero con %d...\n", numero);
                sleep(20);
                exit(numero);
      default : /* PADRE */
                wait(&estado);
                printf("Soy el padre. ");
                if ((estado & 0x7F) != 0) {
                   printf("Mi hijo ha muerto con una señal.\n");
                }
                else {
                   printf("Mi hijo ha muerto con exit(%d).\n",
                          (estado>>8) & 0xFF);
                }
                exit(0);
    }
}
```

Si compilamos y ejecutamos obtenemos lo siguiente:

```
$ gcc -o creaproc3 creaproc3.c
$ creaproc3
Soy el hijo y muero con 13...
Soy el padre. Mi hijo ha muerto con exit(13).
$ creaproc3 &
[1] 1006
Soy el hijo y muero con 13...
$ ps
```

```
UID        PID  PPID  C STIME TTY          TIME CMD
alu        893   891  0 18:06 pts/1    00:00:00 /bin/bash
alu       1006   893  0 18:34 pts/1    00:00:00 creaproc3
alu       1007  1006  0 18:34 pts/1    00:00:00 creaproc3
alu       1008   893  0 18:34 pts/1    00:00:00 ps –f
$ kill -9 1007
Soy el padre. Mi hijo ha muerto con una señal.
$
```

EJECUCIÓN DE PROCESOS

LLAMADA EXEC

```
int execl (char *path, char *arg0, char *arg1,..., NULL);
int execv (char *path, char *argv[]);
int execle(char *path, char *arg0, char *arg1,..., NULL, char *env[]);
int execve (char *path, char *argv[], char *env[]);
int execlp(char *file, char *arg0, char *arg1,..., NULL);
int execvp(char *file, char *argv[]);
```

Los procesos Unix pueden modificarse usando la llamada al sistema `exec`. A diferencia de `fork` (donde a la vuelta de la llamada existen dos procesos: el invocador o padre y el generado o hijo), la llamada `exec` produce la sustitución del programa invocador por el nuevo programa invocado. La ejecución del proceso continúa siguiendo el nuevo programa.

`fork` crea nuevos procesos, `exec` sustituye la imagen de memoria del proceso por otra nueva (sustituye todos los elementos del proceso: código del programa, datos, pila, montículo).

Las opciones de las distintas versiones de *exec* se pueden entender a través de las letras añadidas a *exec* en el nombre de las llamadas, como podemos ver en la siguiente tabla.

Letra	Significado
l	Los argumentos pasados al programa se especifican como una lista parámetros de tipo cadena, terminada con un parámetro NULL (que indica que la lista ha terminado).
v	Los argumentos pasados al programa se especifican como un vector de punteros a carácter (cadenas), terminado con un puntero nulo (que indica que el vector ha terminado).
p	Indica que para encontrar el programa ejecutable hay que utilizar la variable PATH definida, siguiendo el algoritmo de búsqueda de UNIX. Por lo tanto no es necesario especificar una vía absoluta al archivo, sino únicamente el nombre del ejecutable.
e	Indica que las variables de entorno que utilizará el programa serán las que se indique en el parámetro correspondiente mediante un vector de punteros a carácter (cadenas) terminadas en un puntero nulo, y no las variables que utiliza el proceso actual.

El nuevo programa activado mantiene el mismo PID así como otras propiedades asociadas al proceso. Sin embargo, el tamaño de memoria de la imagen del proceso cambia dado que el programa en ejecución es diferente.

Por ejemplo, si el proceso padre quiere ejecutar el comando `ls`

```
main() {
   execlp("ls", "ls", "-al", "*.c", NULL);
   perror("Error al ejecutar comando");
}
```

Si por el contrario el padre crea un hijo para que ejecute el comando `ls`

```
main() {
   printf("Este es el listado de archivos\n");
   switch(fork()) {
      case -1 : /* ERROR */
                perror("Error en fork");
                exit(1);
      case 0  : /* HIJO */
                execlp("ls","ls","-la",NULL);
                perror("Error en exec");
                exit(-1);
      default : /* PADRE */
                wait(&statusp)
                printf("Esto ha sido todo\n");
                exit(0);
   }
}
```

También podemos crear un programa y llamarlo desde otro. El código fuente del primer programa a ejecutar, `prog1.c`, es el siguiente:

```
#include <stdio.h>
#include <unistd.h>

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

     printf ("\nEjecutando el programa invocador (prog1). Sus argumentos
son: \n");
     for ( i = 0; i < argc; i ++ )
          printf ("    argv[%d] : %s \n", i, argv[i]);

     sleep( 10 );
     strcpy (argv[0],"prog2");
     if (execvp ("./prog2", argv) < 0) {
          printf ("Error en la invocacion a prog2 \n");
          exit (1);
          };
     exit (0);
}
```

El código fuente del segundo programa a ejecutar, `prog2.c`, es el siguiente:

```
#include <stdio.h>
#include <unistd.h>

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

     printf ("Ejecutando el programa invocado (prog2). Sus argumentos son:
\n");
     for ( i = 0; i < argc; i ++ )
          printf ("    argv[%d] : %s \n", i, argv[i]);
     sleep(10);
     exit (0);
}
```

Será necesario compilar ambos programas, usando la orden `gcc`. Tras ello, se estará en condición de ejecutarlos:

```
$ gcc -o prog2 prog2.c
$ gcc -o prog1 prog1.c
```

Para ejecutar el programa en modo background y así poder verificar su ejecución con la orden `ps`, se debe añadir al nombre del programa el carácter & (ampersand):

```
$ prog1 p1 p2 param3 &
Ejecutando el programa invocador (prog1). Sus argumentos son:
    argv[0] : prog1
    argv[1] : p1
    argv[2] : p2
    argv[3] : param3
$ ps -f
UID  PID PPID   C    STIME    TTY  TIME CMD
 alu 8736 9176   0 10:23:32  pts/0  0:00 prog1 p1 p2 param3
 alu 9176 6328   1 10:21:21  pts/0  0:00 -ksh
 alu 9296 9176   7 10:23:36  pts/0  0:00 ps –f
$
Ejecutando el programa invocado (prog2). Sus argumentos son:
    argv[0] : prog2
    argv[1] : p1
    argv[2] : p2
    argv[3] : param3
$ ps –f
     UID  PID PPID   C    STIME    TTY  TIME CMD
 alu 8736 9176   0 10:23:32  pts/0  0:00 prog2 p1 p2 param3
 alu 9176 6328   1 10:21:21  pts/0  0:00 -ksh
 alu 9298 9176   5 10:23:46  pts/0  0:00 ps -f
$
```

SEÑALES

El siguiente código programa al manejador de reloj mediante la llamada al sistema `alarm()` para que nos avise dentro de 5 segundos enviándonos la señal `SIGALRM`. Una vez recibida, se salta a la rutina de servicio alarma que imprime un mensaje y continúa la ejecución normal del proceso.

```
#include <signal.h>

void alarma()
{
   printf("acabo de recibir un SIGALRM\n");
}

main()
{
     signal(SIGALRM,alarma);
     printf("Acabo de programar la captura de un SIGALRM\n");
     alarm(3);
     printf("Ahora he programado la alarma en 3 seg.\n");
     pause();
     printf("Ahora continúo con la ejecución normal\n");
}
```

El sistema tiene por defecto un par de manejadores para el tratamiento de todas y cada una de las señales: SIG_DFL y SIG_IGN. El primero de ellos, llama al manejador por defecto asociado a la señal el cual realiza un exit con el número de señal que recibe, mientras que el segundo de ellos, ignora la señal que llega.

LLAMADA SIGNAL

`void (*signal(int numero, void(*funcion)(int)))(int)`

La llamada prepara a un proceso para recibir una señal, asignando una función manejadora (*handler*) a un tipo de señal. `numero` es el tipo de señal y `funcion` es una función de tipo `void` y con un parámetro entero. Devuelve un puntero a la antigua función manejadora de esa señal en caso de éxito, y `SIG_ERR` en caso de error.

Cuando un proceso recibe una señal, su ejecución se detiene y se pasa a ejecutar un manejador, que consiste en una función que se ejecuta cuando se recibe esa señal. Cuando esta función termina, el proceso continúa ejecutándose por el mismo punto en que se quedó.

Existen diversas señales, identificadas por un número entero. Cada una de ellas tiene un significado particular. Mediante el archivo <signal.h> o bien por medio del comando `kill -l` se pueden identificar las señales del sistema. Las más importantes son las siguientes:

Nombre	Número	Significado
SIGINT	2	Se ha pulsado CRTL-C
SIGQUIT	3	Se ha pulsado CRTL-/

Nombre	Número	Significado
SIGILL	4	Se ha intentado ejecutar una instrucción no válida.
SIGTRAP	5	Se ha producido una interrupción de traza.
SIGFPE	8	Se ha producido un error en una operación de punto flotante.
SIGKILL	9	Matar proceso.
SIGUSR1	10	Señal de usuario. Libre para ser reprogramada.
SIGUSR2	12	Señal de usuario. Libre para ser reprogramada.
SIGPIPE	13	Utilización de una *PIPE* que no tiene ningún proceso en el otro extremo.
SIGALRM	14	Vencimiento de alarma programada.
SIGTERM	15	Señal de terminación enviada por el sistema
SIGCHLD	17	Señal que envía un hijo a su padre cuando finaliza

Si un proceso no ha definido explícitamente una función manejadora para una determinada señal, al recibirse esa señal el proceso muere. Por tanto, antes de esperar a recibir una señal debe definirse su manejador. Sin embargo, la señal `SIGKILL` no puede ser manejada. Así, si se recibe, matará al proceso incondicionalmente.

La función manejadora es una función de tipo `void` con un parámetro de tipo entero por el cual se indica el número de la señal recibida, de manera que el mismo manejador pueda valer para distintas señales. Existen dos funciones estándar que pueden programarse como manejadores de cualquier señal (excepto `SIGKILL`):

- `SIG_DFL`: es la rutina por defecto, que mata al proceso.
- `SIG_IGN`: que hace que se ignore la señal (no hace nada, pero no mata al proceso).

La programación de una función manejadora sólo sirve una vez. Después de ejecutarse el manejador, se vuelve automáticamente a programar la función por defecto `SIG_DFL`, por lo que si queremos que sirva para más veces debemos reprogramar el manejador dentro del propio manejador. Esto ocurre en el estándar POSIX. El siguiente ejemplo lo ilustra:

```
#include <signal.h>
void alarma() {
   printf("acabo de recibir un SIGALRM\n");
}
main(){
     signal(SIGALRM,alarma);
     printf("acabo de programar la captura de un SIGALRM\n");
     alarm(3);
     printf("Ahora he programado la alarma en 3 seg.\n");
     pause();
     printf("vuelvo a programar la alarma\n");
     alarm(3);
     pause();
     printf("En POSIX esta línea nunca se ejecutaría porque me ha matado el
SIGALRM\n");
}
```

Al crearse procesos mediante `fork()` la programación de señales en el hijo se hereda de tal forma que las señales programadas con `SIG_IGN` y `SIG_DFL` se conservan y las programadas con funciones propias pasan a ser `SIG_DFL`.

La recepción de señales no se hace necesariamente en el mismo orden en que se envían. Además, no existe una memoria de señales, de manera que si llegan varias señales (iguales) a la vez, sólo se ejecutará el manejador una vez.

LLAMADA KILL

`int kill(int PID, int numero)`

La llamada `kill` permite a un proceso enviar una señal a otro proceso o así mismo a través del PID. `PID` es el PID del proceso señalado y `numero` es el tipo de señal enviada. Devuelve `−1` en caso de error y `0` en caso contrario.

Existe una limitación sobre los procesos a los que pueden enviarse señales: deben tener alguna relación de parentesco del tipo: antecesor, hermano o hijo.

Por ejemplo, el siguiente código sirve para que un proceso se suicide:

```
#include <signal.h>

main() {
   printf("Voy a suicidarme\n");
   kill(getpid(), SIGKILL);
   perror("No he muerto???");
}
```

LLAMADA ALARM

`unsigned alarm(unsigned tiempo)`

La llamada `alarm` programa una alarma, de manera que cuando hayan transcurrido `tiempo` segundos, se enviará al proceso una señal de tipo `SIGALRM`. Devuelve `0` si no había otra alarma previamente programada, o el `número de segundos` que faltaban para cumplirse el tiempo de otra alarma previamente programada.

Sólo es posible programar una alarma por proceso. Al programarse una alarma se desconecta cualquier otra que hubiera previamente.

Un ejemplo de uso de `alarm es el siguiente`: poder imprimir en el terminal durante 5 segundos.

```
#include <signal.h>

int seguir = 1; /* Variable global */

void fin(int n) {
   seguir = 0;
}
```

```
main() {
   int contador = 0;
   signal(SIGALRM, fin);
   alarm(5);
   do {
      printf("Esta es la línea %d\n", contador++);
   } while (seguir);
   printf("TOTAL: %d líneas\n", contador);
}
```

LLAMADA PAUSE

int pause()

La llamada `pause` detiene la ejecución de un proceso (mediante espera pasiva) hasta que se reciba alguna señal. Devuelve siempre `-1` y establece el error *Interrupted system call*.

Este ejemplo crea un hijo y espera su señal.

```
#include <signal.h>
main() {
   switch(fork()) {
      case -1 : /* ERROR */
                perror("Error en fork");
                exit(1);
      case 0  : /* HIJO */
                printf("Hola, soy el hijo. Espero 2 segundos...\n");
                sleep(2);
                kill(getppid(),SIGUSR1);
                printf("Soy el hijo. He señalado a mi padre. Adios.\n");
                exit(0);
      default : /* PADRE */
                printf("Hola, soy el padre y voy a esperar.\n");
                signal(SIGUSR1, SIG_IGN); /* Ignoro señal para no morir */
                pause();
                printf("Soy el padre y ya he recibido la señal.\n");
                exit(0);
    }
}
```

Una señal muy utilizada es `SIGCHLD` que es enviada por todo proceso hijo a su padre en el mismo instante que realiza `exit`. De ésta manera, el padre sabe que su hijo ha pedido terminar. Por ejemplo, el siguiente programa, una vez convertido en proceso, tiene un hijo que realiza `exit(5)`. El padre captura el 5.

```
#include <signal.h>
int status,pid;
void finhijo(){
   pid=wait(&status);
}

main(){
     signal(SIGCHLD,finhijo);
```

```
     if (fork()==0) {sleep(3); exit(5);}
     pause();
     printf("mi hijo ha muerto con estado %d\n",status/256);
     printf("ahora continúo con la ejecución\n");
}
```

GESTIÓN DE ARCHIVOS

LLAMADA OPEN

*int open(char *nombre, int operaciones [, int permisos])*

La llamada `open()` crea un archivo o abre un archivo ya creado para lectura y/o escritura. Antes de realizar la mayoría de operaciones sobre un archivo, éste debe ser previamente abierto.

El parámetro `nombre` es una cadena de caracteres que representa el nombre del archivo o dispositivo al que se quiere acceder, con las convenciones de nombres de archivo de UNIX.

El parámetro `operaciones` es un número entero que representa las operaciones que se van a realizar sobre el archivo: lectura, escritura, creación, etc. Lo habitual es confeccionar este número mediante la concatenación con operadores OR de una serie de opciones definidas mediante constantes en el archivo `<fcntl.h>`. Las más importantes son las que aparecen en la siguiente tabla:

Valor	Significado
`O_RDONLY`	Abre para lectura
`O_WRONLY`	Abre para escritura
`O_RDWR`	Abre para lectura y escritura (equivalente a `O_RDONLY|O_WRONLY`)
`O_CREAT`	Crea el archivo si no existe e ignora el parámetro si ya existe.
`O_APPEND`	Sitúa la posición al final antes de escribir

El parámetro opcional `permisos` sólo tiene sentido al utilizar `O_CREAT`, y sirve para asignar los permisos del archivo recién creado, en caso de que no existiera.

La llamada `open()` devuelve -1 en caso de error, o un entero positivo como ***descriptor*** o manejador de archivo, en caso de éxito. Este entero debe ser almacenado para acceder posteriormente a los datos del archivo, ya que el resto de operaciones sobre el mismo se realizarán a través del descriptor de archivo.

El descriptor de archivo es un índice que identifica una entrada en la ***tabla de descriptores de archivos***. La tabla de descriptores de archivos es una tabla que posee cada proceso. En ella se almacenan toda la información concerniente a los archivos (y tuberías) que se están utilizando en un proceso (sus archivos abiertos). Cada vez que se abre un archivo se utiliza una entrada de esta tabla y se devuelve como descriptor el número de entrada utilizada.

Cuando se crea un proceso, su tabla de descriptores de archivos contiene las tres primeras entradas ocupadas con otros tantos archivos abiertos, que se utilizan como entradas y salidas estándar del proceso, como se muestra en la siguiente tabla. Hay que tener en cuenta que un proceso siempre hereda la tabla de descriptores de archivos del proceso que lo creó (proceso padre).

Entrada	Archivo	
0	Entrada estándar	Teclado
1	Salida estándar	Pantalla
2	Salida de errores estándar	Pantalla
…	Archivos abiertos	…
…	Archivos abiertos	…

Cuando se emplean los caracteres de redireccionamiento sobre un proceso, realmente lo que se está haciendo es modificar su tabla de descriptores de archivos. Por ejemplo, `$ a.out > salida.txt` modifica la tabla como sigue

Entrada	Archivo	
0	Entrada estándar	Teclado
1	Salida estándar	datos.txt
2	Salida de errores estándar	Pantalla

LLAMADA CLOSE

`int close(int archivo)`

La llamada `close()` cierra un archivo abierto, cuyo descriptor es `archivo` y libera la entrada correspondiente en la tabla de descriptores de archivos. Devuelve –1 en caso de error y 0 en caso de éxito.

LLAMADA CREAT

`int creat(char *nombre, int permisos)`

La llamada `creat()` crea un nuevo archivo con el nombre indicado en el parámetro `nombre` y con los permisos indicados en el parámetro `permisos`, y además lo abre para escritura aunque los permisos no lo permitan. Si el archivo ya existe, los trunca a cero y lo abre para escritura, ignorando el parámetro de permisos.

`creat()` devuelve –1 en caso de error o el descriptor del archivo abierto en caso de éxito.

Ejemplo: creación de un archivo vacío con todos los derechos.

```
int df;

...

df = creat("nuevo.txt",0777);
if (df < 0) perror ("Error al crear");
else close(df);
```

LLAMADA READ

*int read(int archivo, char *buffer, int nbytes)*

La llamada `read()` lee una secuencia de `nbytes` octetos (bytes) del archivo o dispositivo abierto (para lectura) cuyo descriptor es `archivo` y los escribe a partir de una posición de memoria indicada por el parámetro `buffer`. Devuelve –1 en caso de error o el número de bytes realmente leídos en caso de éxito.

Es posible que el número de bytes que realmente se leen no coincida con el pedido a través del parámetro `nbytes`. Esto puede suceder cuando, por ejemplo, se intentan leer más bytes de los que realmente hay en un archivo. El hecho de que la llamada `read()` devuelva 0 no es un error, y por tanto no actualiza el valor de `errno`.

`El parámetro buffer` es un puntero a carácter (vector de caracteres) y por tanto debe apuntar a una zona de memoria suficientemente grande como para contener todos los caracteres solicitados en el parámetro `nbytes`. De lo contrario, la llamada `read` podría sobrescribir otras variables del proceso, producir un error o incluso producir un error de protección y abortar.

Después de una llamada `read()` la posición del archivo (punto del archivo desde el cual empieza a leerse o escribirse) avanza tantos octetos como caracteres (bytes) hayan sido leídos.

Ejemplo: leer un número del terminal y multiplicarlo por 2.

```
#include <fcntl.h>

main() {
   int df,tam;
   long numero;
   char buffer[10];

   df=open("/dev/tty", O_RDONLY);
   if (df<0) {
      perror("Error al abrir tty");
      exit(-1);
   }
   tam=read(df, buffer, 9); /* COMO MUCHO DE HASTA 9 DIGITOS */

   if (tam == -1)
      perror("Error de lectura");
   else {
      buffer[tam]=0; /* PONE EL FINAL DE CADENA */
      numero=atoi(buffer);
      printf("Resultado: %ld\n",numero*2);
   }
   close(df);
}
```

LLAMADA WRITE

*int write(int archivo, char *buffer, int nbytes)*

La llamada `write()` es semejante a `read()`, pero a la inversa. Escribe una secuencia de `nbytes` octetos (bytes) en el archivo o dispositivo abierto (para escritura) cuyo descriptor es `archivo`. La secuencia de datos a escribir se obtiene a partir de una posición de memoria indicada por el parámetro `buffer`. Devuelve –1 en caso de error o el número de bytes realmente escritos en caso de éxito. Es posible que el número de bytes que realmente se escribe no coincida con el pedido a través del parámetro `nbytes`. Esto puede suceder cuando, por ejemplo, se intentan escribir más bytes de los que realmente caben en el disco.

Después de una llamada `write()` la posición del archivo avanza tantos bytes como caracteres hayan sido escritos.

Ejemplo: crear un archivo con las variables de entorno.

```
#include <string.h>

main(int argc, char *argv[], char *env[]) {
   int df, cont=0;
   char buffer[128];

   df=creat("variables.txt", 0755);
   if (df<0) {
      perror("Error al crear archivo");
      exit(-1);
   }
   while (env[cont] != NULL) {
      strcpy(buffer, env[cont]);
      strcat(buffer, "\n");
      if (write(df, buffer, strlen(buffer)) != strlen(buffer)) {
         perror("Error al escribir");
         break; /* NO SIGUE */
      }
      cont++;
   }
   close(df);
}
```

LLAMADA LSEEK

`int lseek(int archivo, long desplazamiento, int modo)`

La llamada `lseek()` sirve para desplazar la posición del archivo cuyo descriptor es `archivo`. Hay tres posibilidades, seleccionadas mediante el parámetro `modo`, definidas en el archivo `<stdio.h>` y recogidas en la tabla siguiente:

Modo	Valor	Desplazamiento
SEEK_SET	0	Absoluto: `desplazamiento` es la nueva posición
SEEK_CUR	1	Relativo: la nueva posición es `desplazamiento` bytes desde la posición actual
SEEK_END	2	Relativo al fin de archivo: la nueva posición es `desplazamiento` bytes desde el final del archivo

`lseek()` devuelve –1 en caso de error y la nueva posición del archivo en caso de éxito. La nueva posición puede ser diferente de la que indica `desplazamiento` en el caso de que se excedan los límites del archivo.

LLAMADA UNLINK

`int unlink(char *archivo)`

La llamada `unlink()` elimina del disco el archivo cuyo nombre es `archivo`. Devuelve –1 en caso de error y 0 en caso de éxito.

LLAMADA DUP

`int dup(int manejador)`

La llamada `dup()` duplica una entrada de la tabla de descriptores de archivos. La entrada duplicada es la que indica el parámetro `manejador`. La nueva entrada, se sitúa en la primera entrada libre de la tabla de descriptores de archivos.

Devuelve –1 en caso de error y el número de entrada ocupada en caso de éxito (nuevo descriptor de archivo). Después de una llamada `dup()` el archivo puede ser accedido de la misma forma tanto desde el descriptor antiguo como desde el nuevo, ya que para ambos descriptores tienen las mismas características:

- Están asociados al mismo archivo o dispositivo.
- La posición del archivo es la misma para ambos.
- Comparten el mismo modo de acceso (lectura, escritura...).

Los dos descriptores deben cerrarse independientemente para liberar las entradas correspondientes en la tabla de descriptores de archivos.

LLAMADA STAT()

`int stat(char *archivo, struct stat *buffer)`

La llamada `stat()` devuelve información sobre un `archivo`: tipo de dispositivo, propietario, grupo, hora de acceso, modificación, etc. Todos estos datos están contenidos en la estructura `stat`, definida en `<sys/stat.h>`.

```
struct stat {                {
   dev_t          st_dev;       /* DISPOSITIVO */
   ino_t          st_ino;       /* INODO */
   umode_t        st_mode;      /* PROTECCIÓN */
   nlink_t        st_nlink;     /* NÚMERO DE ENLACES DUROS */
   uid_t          st_uid;       /* PID DEL PROPIETARIO */
   gid_t          st_gid;       /* GID DEL PROPIETARIO */
   dev_t          st_rdev;      /* TIPO DE DISPOSITIVO (SI DISPOSITIVO) */
   off_t          st_size;      /* TAMAÑO TOTAL EN BYTES */
   unsigned long  st_blksize;   /* TAMAÑO DE BLOQUE */
   unsigned long  st_blocks;    /* NÚMERO DE BLOQUES OCUPADOS */
   time_t         st_atime;     /* HORA DE ÚLTIMO ACCESO */
   time_t         st_mtime;     /* HORA DE ÚLTIMA MODIFICACIÓN */
   time_t         st_ctime;     /* HORA DE ÚLTIMO CAMBIO */
};
```

LLAMADAS MKDIR(), RMDIR() Y CHDIR()

```
int mkdir(char *nombre, mode_t modo)
int rmdir(char *nombre)
int chdir(char *nombre)
```

La llamada `mkdir()` crea un directorio: `nombre` es la ruta a crear y `modo` son los permisos asignados; `rmdir()` elimina un directorio (que debe estar vacío); y `chdir()` cambia el directorio actual.

En todos los casos se retorna –1 si ocurre algún error y 0 de lo contrario. `mode_t` se define en `<sys/types.h>`.

GESTIÓN DE TUBERÍAS

Una tubería (`pipe`) se puede considerar como un canal de comunicación entre dos procesos. Los mecanismos que se utilizan para manipular tuberías son los mismos que para archivos, con la única diferencia de que la información de la tubería no se almacena en el disco duro, sino en la memoria principal del sistema.

Esta comunicación consiste en la introducción de información en una tubería por parte de un proceso (similar a la escritura en un archivo de disco). Posteriormente otro proceso extrae la información de la tubería (similar a la lectura de información almacenada en un archivo de disco) de forma que los primeros datos que se introdujeron en ella son los primeros en salir. Este modo de funcionamiento se conoce como FIFO (First In, First Out; el primero en entrar es el primero en salir).

La comunicación mediante tuberías es de tipo half-duplex, es decir, en un instante dado, la comunicación solamente puede tener lugar en un sentido. Si se quiere que un proceso A pueda simultáneamente enviar y recibir información de otro B, en general se debe recurrir a crear dos tuberías, una para enviar información desde A hacia B y otra para enviar desde B hacia A.

Un tubo se puede ver como un archivo intermedio o un pseudoarchivo. A nivel del shell su implementación consiste en variar las entradas estándar sobre las tablas de descriptores de archivos de cada proceso. `$ cat | wc` genera
`cat`:

Entrada	Archivo	
0	Entrada estándar	Teclado
1	Salida estándar	pseudoarchivo
2	Salida de errores estándar	Pantalla

`wc`:

Entrada	Archivo	
0	Entrada estándar	pseudoarchivo
1	Salida estándar	Pantalla
2	Salida de errores estándar	Pantalla

Se pueden distinguir dos tipos de tuberías dependiendo de las características de los procesos que pueden tener acceso a ellas:

- Sin nombre. Solamente pueden ser utilizadas por los procesos que las crean y por los descendientes de éstos.
- Con nombre o fifo. Se utilizan para comunicar procesos entre los que no existe ningún tipo de parentesco.

En esta práctica nos centraremos en la tubería sin nombre.

LLAMADA PIPE

```
int pipe(int descriptores[2])
```

La llamada `pipe()` crea una *pipe* (tubería) sin nombre. Esta llamada admite como parámetro un vector `descriptores` con espacio para almacenar dos descriptores de archivo. El primero de ellos (`descriptores[0]`) sirve para que los procesos lean de la tubería, mientras que el segundo (`descriptores[1]`) se emplea para escribir los datos en la tubería.

Para que dos procesos puedan intercambiar información entre sí deben compartir las entradas en la tabla de descriptores de archivos, y la única forma de que esto sea posible es que mantengan algún tipo de parentesco. Al hacer `fork()` la tabla de descriptores de archivos se hereda, y por tanto, cualquier tubería creada con anterioridad también. Para que la tubería se conserve al hacer `exec()` es necesario mantener sus descriptores en algunas de las tres primeras entradas de la tabla de descriptores de archivos, ya que son las únicas que se conservan en el nuevo proceso creado con `exec()`.

Si un proceso se va a limitar a leer o a escribir sobre una tubería, puede cerrar el descriptor de la operación que no va a realizar, ya que los descriptores son un recurso limitado.

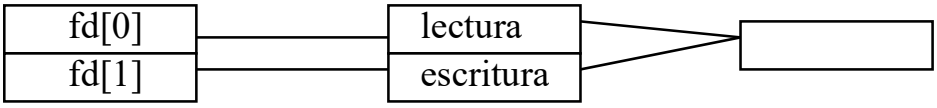
Las operaciones `read()` y `write()` tienen un comportamiento ligeramente distinto para las tubería respecto a los archivos. Una tubería funciona como una estructura de tipo cola en la que los elementos escritos en ella la van llenando, y se va vaciando cuando se lee.

El funcionamiento de la tubería sigue las siguientes pautas:

- Si un proceso intenta leer de una tubería vacía, quedará detenido hasta que haya datos que leer.

- Si un proceso intenta leer de una tubería vacía, pero todos los posibles descriptores del otro extremo de la tubería (es decir, de la parte de escritura) han sido cerrados, el proceso no quedará detenido y se retornará un final de archivo (*EOF*).
- Si un proceso intenta leer de una tubería no vacía, pero con menos datos de los pedidos en `read()`, se leerán los datos disponibles y no quedará detenido.
- Si un proceso intenta escribir en una tubería que está llena, quedará detenido hasta que haya espacio suficiente para escribir todos los datos. La escritura se completará o no comenzará; en ningún caso se escribirán menos bytes de los indicados en `write()`.
- Si un proceso intenta escribir en una tubería, pero todos los posibles descriptores del otro extremo de la tubería (es decir, de la parte de lectura) han sido cerrados, se producirá un error, el proceso recibirá una señal de tipo `SIGPIPE`, y si no está preparado para recibirla, morirá.

Con la llamada `pipe(fd)` creamos dos enlaces con el archivo `fd`, uno de lectura y otro de escritura



MEMORIA COMPARTIDA

Tal y como hemos estudiados, cuando ejecutamos la llamada `fork` se crea un nuevo proceso, idéntico al padre, en el que se duplican las variables, funciones, etc. Si el hijo actualiza una de sus variables, el proceso padre no tiene tal actualización ya que son variables distintas (son copias). Para que el padre y el hijo tengan la misma variable y se puedan comunicar es necesario emplear las llamadas al sistema que permiten gestionar la memoria compartida.

Para utilizar la memoria compartida se siguen 3 pasos:

- Obtención de un segmento de memoria compartida con su identificador en el sistema (un número entero). Se hace con la llamada `shmget`.
- Vincular el segmento de memoria compartida a través de un puntero a un determinado tipo de datos. Se usa la llamada `shmat`.
- Acceder a la memoria compartida por medio del puntero.
- Desvincular el puntero del segmento de la memoria compartida. Se usa la llamada `shmdt`.
- Eliminación del segmento de memoria compartida. Se usa la llamada `shmctl`.

A continuación se describen brevemente estas llamadas.

LLAMADA SHMGET

```
int shmget( key_t key, size_t size, int shmflg)
```

La llamada `shmget` devuelve el identificador de memoria compartida asociado al segmento de memoria. Tiene como parámetros:

- La clave que identifica al segmento de memoria compartida. Puede ser IPC_PRIVATE (el proceso y sus descendientes pueden acceder al segmento) u otra clave obtenida mediante la función `ftok` (varios procesos sin parentesco pueden acceder al segmento).
- El tamaño del segmento.
- Indicadores de permisos para acceder al recurso compartido.

Si la ejecución se realiza con éxito, entonces devolverá un valor no negativo denominado identificador de segmento compartido. En caso contrario, retornará -1 y la variable global `errno` tomará en código del error producido.

LLAMADA SHMAT

```
char *shmat( int shmid, void *shmaddr, int shmflg )
```

Esta llamada asocia o vincula el segmento de memoria compartida especificado por `shmid` (el identificador devuelto por `shmget`) al segmento de datos del proceso invocador. Presenta tres parámetros:

- El identificador obtenido mediante `shmget`,
- La dirección donde se desea que se incluya (en la práctica será NULL (0), lo cuál permite que se incluya en cualquier zona libre del espacio de direcciones del proceso).
- Una serie de identificadores de permisos (en la práctica también será NULL).

Si la llamada se ejecuta con éxito, entonces devolverá la dirección de comienzo del segmento compartido, si ocurre un error devolverá -1 y la variable global `errno` tomará el código del error producido.

LLAMADA SHMDT

```
int shmdt ( char *shmaddr)
```

Desasocia o desvincula del segmento de datos del proceso invocador el segmento de memoria compartida ubicado en la localización de memoria especificada por `shmaddr`. Si la función se ejecuta sin error, entonces devolverá 0, en caso contrario retornará -1 y `errno` tomará el código del error producido.

LLAMADA SHMCTL

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

La llamada `shmctl` permite realizar un conjunto de operaciones de control sobre una zona de memoria compartida identificada por `shmid`. El argumento `cmd` se usa para codificar la operación solicitada. Los valores más usados para este parámetro son:

- `IPC_STAT`: lee la estructura de control asociada a `shmid` y la deposita en la estructura apuntada por `buff`.
- `IPC_RMID`: elimina el identificador de memoria compartida especificado por `shmid` del sistema, destruyendo el segmento de memoria compartida y las estructuras de control asociadas.
- `SHM_LOCK`: bloquea la zona de memoria compartida especificada por `shmid`. Este comando sólo puede ejecutado por procesos con privilegios de acceso apropiados.
- `SHM_UNLOCK`: desbloquea la región de memoria compartida especificada por `shmid`. Esta operación, al igual que la anterior, sólo la podrán ejecutar aquellos procesos con privilegios de acceso apropiados.

La función `shmctl` retornará el valor 0 si se ejecuta con éxito, o -1 si se produce un error, tomando además la variable global `errno` el valor del código del error producido.

EJEMPLO DE USO

Como ejemplo vamos a ver cómo se comparte la variable entera `numero`. Para ello, tendremos como variables globales la variable `numero` y el identificador de la memoria compartida `shmid`.

```
int shmid;
int *numero=NULL;
```

Para las llamadas anteriores hay que incluir los archivos de cabecera:

```
#include <sys/types>
#include <sys/ipc.h>
#include <sys/shm.h>
```

La función `creaComp` crea el identificador de la memoria compartida y la variable que contiene.

```
int creaComp(void)
{
      if((shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|0666))==-1)
      {
            perror("Error al crear memoria compartida: ");
            return 1;
      }

      /* vinculamos el segmento de memoria compartida al proceso */
      numero=(int *) shmat (shmid,0,0);

      /*iniciamos las variables */
```

```
      *numero=0;

   return 0;
}
```

La función `borraComp` separa la variable `numero` del proceso y a continuación elimina la memoria compartida.

```
void borraComp(void)
{
   char error[100];
   if (numero!=NULL)
   {
      /* desvinculamos del proceso la memoria compartida */
     if (shmdt((char *)numero)<0)
     {
        sprintf(error,"Pid %d: Error al desligar la memoria compartida:
",getpid());
        perror(error);
        exit(3);
     }
     /* borramos la memoria compartida */
     if (shmctl(shmid,IPC_RMID,0)<0)
     {
        sprintf(error,"Pid %d: Error al borrar memoria compartida:
",getpid());
        perror(error);
        exit(4);
     }
     numero=NULL;
     }
}
```

REFERENCIAS

BIBLIOGRAFÍA

- F. M. Márquez: UNIX. Programación avanzada, Rama, 2004.
- K. A. Robbins y S. Robbins: UNIX. Programación práctica, Prentice may, 2000.

ANEXO I

TABLA DE LLAMADAS AL SISTEMA

Llamada	Categoría	Descripción
`alarm()`	Comunicación	Programa una alarma con un plazo de un cierto número de segundos.
`kill()`	Comunicación	Envía una señal a un proceso.
`pause()`	Comunicación	Detiene la ejecución indefinidamente (hasta la llegada de una señal).
`pipe()`	Comunicación	Crea una tubería y proporciona sus descriptores.

Llamada	Categoría	Descripción
`signal()`	Comunicación	Programa un manejador para una señal.
`chdir()`	Archivos	Cambia el directorio por defecto (directorio actual).
`chmod()`	Archivos	Cambia los permisos de un archivo.
`chown()`	Archivos	Cambia el propietario de un archivo.
`close()`	Archivos	Cierra un archivo.
`creat()`	Archivos	Crea un archivo y lo abre para escritura.
`dup()`	Archivos	Duplica un descriptor de archivo en la siguiente entrada libre de la tabla de archivos.
`dup2()`	Archivos	Duplica un descriptor de archivo en la entrada de la tabla de archivos que se indique.
`fstat()`	Archivos	Obtiene información sobre un archivo abierto: tamaño, fecha, permisos, etc.
`link()`	Archivos	Crea un enlace (duro).
`lseek()`	Archivos	Sitúa el puntero de lectura/escritura en cualquier posición de un archivo.
`mkdir()`	Archivos	Crea un directorio.
`mknod()`	Archivos	Crea un dispositivo especial.
`mount()`	Archivos	Monta un dispositivo en un directorio.
`open()`	Archivos	Abre un archivo y proporciona un descriptor.
`read()`	Archivos	Lee datos de un archivo.
`rmdir()`	Archivos	Elimina un directorio vacío.
`stat()`	Archivos	Obtiene información sobre un archivo: tamaño, fecha, permisos, etc.
`symlink()`	Archivos	Crea un enlace simbólico.
`umount()`	Archivos	Desmonta un dispositivo.
`unlink()`	Archivos	Elimina un enlace duro de un archivo (lo borra del disco).
`utime()`	Archivos	Modifica la fecha y hora de acceso y modificación de un archivo.
`write()`	Archivos	Escribe datos de un archivo.
`getgid()`	Otras	Proporciona el identificador de grupo del usuario actual (GID).
`getuid()`	Otras	Proporciona el identificador del usuario actual (UID).
`setgid()`	Otras	Cambia el identificador de grupo del usuario actual (GID).
`setuid()`	Otras	Cambia el identificador del usuario actual (UID).
`time()`	Otras	Proporciona el número de segundos desde el 1 de enero de 1970 a las 0 horas.
`times()`	Otras	Proporciona el tiempo consumido en la ejecución del proceso actual.
`exec`*`XX`*`()`	Procesos	Cambia el programa actual por el programa en disco indicado (ejecuta un programa).
`exit()`	Procesos	Termina el proceso actual indicando un estado de finalización.
`fork()`	Procesos	Crea un proceso hijo (desdobla el proceso actual).
`getpid()`	Procesos	Proporciona el PID del proceso actual.
`getppid()`	Procesos	Proporciona el PID del proceso padre.
`nanosleep()`	Procesos	Detiene la ejecución en espera pasiva durante un número de nanosegundos.
`sleep()`	Procesos	Detiene la ejecución en espera pasiva durante un número de segundos.
`usleep()`	Procesos	Detiene la ejecución en espera pasiva durante un número de microsegundos.
`wait()`	Procesos	Espera por la muerte de un hijo (cualquiera) y obtiene su estado de finalización.

ANEXO II

UTILIZACIÓN DEL COMPILADOR DE C

El compilador de C en los sistemas UNIX es un programa llamado `cc (gcc en Linux)` que funciona como un comando más, sin editor ni entorno. Los archivos fuente y las opciones de compilación deben pasarse como parámetro. Los errores encontrados se imprimen por la salida de errores.

El programa `cc` realiza todos los pasos de compilación en C: preproceso, compilación a objeto y enlace. Por ello es necesario incluir en la llamada al compilador todos los archivos fuente que intervienen en la compilación, así como las bibliotecas necesarias.

Habitualmente se construyen archivos `Makefile` para facilitar la tarea de compilación. Los archivos `Makefile` son archivos de texto que contienen una descripción formal de los pasos necesarios para recompilar un sistema complejo, de manera que, con introducir el comando `Make`, el compilador es invocado con los parámetros adecuados.

El funcionamiento básico del compilador consiste en invocar el comando `cc` indicando únicamente el nombre del archivo fuente:

```
$ gcc fuente.c
```

Este comando realiza la compilación del archivo `fuente.c` y produce un programa ejecutable llamado `a.out`. Este es el nombre del programa ejecutable por defecto. Para que el ejecutable generado tenga otro nombre podemos utilizar la opción `-o` como en el siguiente ejemplo:

```
$ gcc –o salida fuente.c
```

Ahora el ejecutable creado se llamará `salida`. En los ejemplos anteriores se han utilizado las bibliotecas del sistema para enlazar. Si son necesarias otras bibliotecas, podemos utilizar la opción `-l` para indicar que se incluyan, como en el siguiente ejemplo:

```
$ gcc –o salida fuente.c -lm
```

En este caso `-lm` indica que se debe enlazar con la biblioteca matemática estándar. Las principales bibliotecas estándar se recogen en la siguiente tabla:

Nombre	Uso
c	Biblioteca estándar del C. No hace falta incluirla explícitamente
m	Biblioteca estándar matemática
curses	Contiene rutinas de visualización con ventanas, etc.
l	Biblioteca para uso del *lex* y el *yacc*

Cuando el programa fuente esté compuesto por más de un archivo, deberá especificarse la lista completa de archivos que lo componen. En esta lista pueden aparecer archivos fuentes en C, ensamblador y archivos objeto. Por ejemplo:

```
$ gcc –o programa principal.c modulo1.c modulo2.c utilidades.o
```

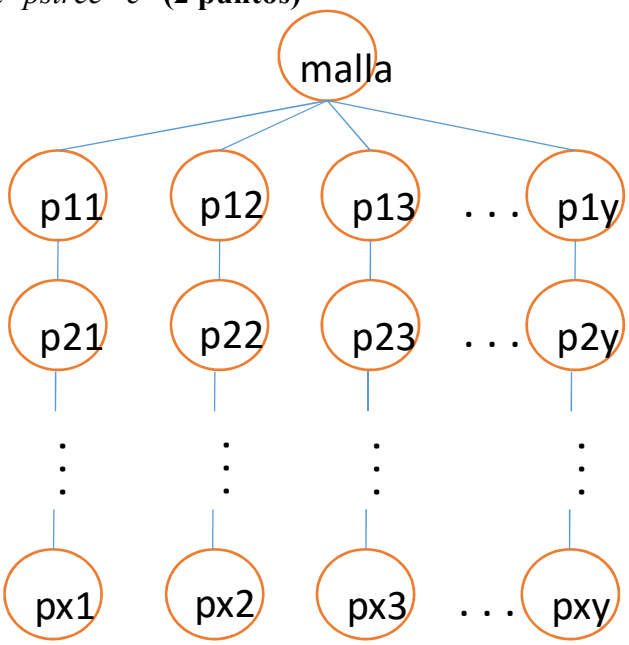
Esta instrucción compilará los archivos fuente en C `principal.c`, `modulo1.c` y `modulo2.c` junto con el código objeto de `utilidades.o` y generará un ejecutable llamado `programa`. Es muy importante especificar correctamente la extensión de cada archivo, ya que el compilador interpretará su contenido a través de la extensión. Por ejemplo, para compilar un archivo en C, obligatoriamente la extensión debe ser `.c`. Se suelen utilizar las extensiones de la siguiente tabla para nombrar los diferentes tipos de archivos:

Extensión	Tipo
`.c`	archivo fuente en C
`.h`	archivo cabecera fuente en C (para los `#include`)
`.s`	archivo fuente en ensamblador
`.o`	archivo objeto
`.a`	archivo de biblioteca

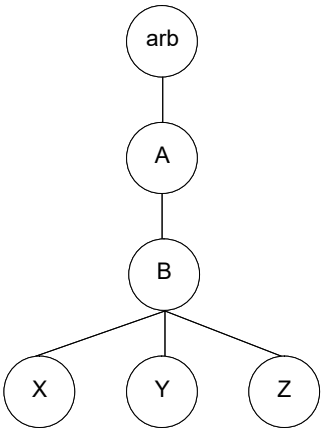
EVALUACIÓN

Ejercicio 1. Gestión básica de procesos. (**4 puntos**)

a) Realiza un programa llamado **malla.c** que produzca el siguiente árbol de procesos. El programa recibirá dos argumentos 'x' e 'y' que representan el número de filas y columnas. Para comprobar la estructura de procesos que realmente estamos creando debemos emplear la llamada al sistema "*pstree –c*" **(2 puntos)**



b) Realiza un programa llamado **ejec.c** que reciba un argumento. El programa tendrá que generar el árbol de procesos que se indica y llevar a cabo la funcionalidad que se describe a continuación. El proceso Z, transcurridos los segundos indicados por el argumento, ejecutará el comando "pstree". No se puede utilizar el comando "sleep", por lo que el proceso Z debe planificarse una alarma con los segundos indicados por el argumento. Se deberá controlar la correcta destrucción del árbol (los padres no pueden morir antes que los hijos). **(2 puntos)**



El programa debe mostrar por la salida estándar la siguiente información:

```
$ ejec 15
Soy el proceso ejec: mi pid es 751
Soy el proceso A: mi pid es 752. Mi padre es 751
Soy el proceso B: mi pid es 753. Mi padre es 752. Mi abuelo es 751
Soy el proceso X: mi pid es 754. Mi padre es 753. Mi abuelo es 752. Mi bisabuelo es 751
Soy el proceso Y: mi pid es 755. Mi padre es 753. Mi abuelo es 752. Mi bisabuelo es 751
Soy el proceso Z: mi pid es 756. Mi padre es 753. Mi abuelo es 752. Mi bisabuelo es 751
/*Tras un intervalo de 15 segundos aparecerá*/
/*Resultado del comando "pstree" */
Soy Z (756) y muero
Soy Y (755) y muero
Soy X (754) y muero
Soy B (753) y muero
Soy A (752) y muero
Soy ejec (751) y muero
```

Ejercicio 2. Comunicación entre procesos: tuberías. (**4 puntos**)

Realizar un programa llamado hacha.c que divida un archivo en varios trozos con el mismo nombre y extensión h00, h01, … teniendo en cuenta el formato y las siguientes consideraciones:

$ hacha <archivo> <tamaño>
d: divide
archivo a dividir
tamaño en bytes de los archivos divididos

- Para dividir, el proceso hacha (proceso padre) generará tantos procesos hijos como archivos se tengan que crear. Mediante tuberías el proceso padre enviará la información a los hijos y serán estos últimos los que creen los archivos de destino y escriban en ellos.

```
$ hacha at_madrid.mp3 50000
$ ls
at_madrid.mp3.h00
at_madrid.mp3.h01
at_madrid.mp3.h02
```

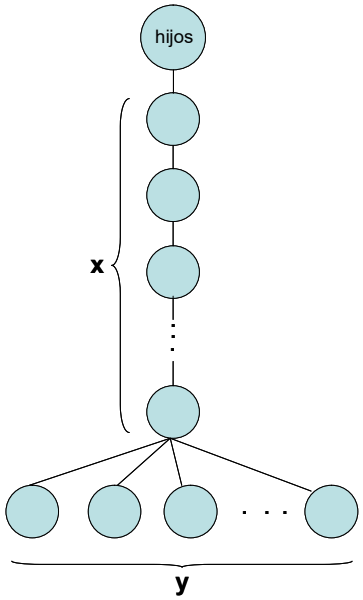
Consideraciones

- Las entradas y salidas a los archivos se realizarán con llamadas al sistema estudiadas en esta práctica (no se puede utilizar printf, scanf, etc.)
- El padre creará tantos hijos como fragmentos del archivo a realizar, siendo decisión del alumno si los hijos se lanzan secuencial o concurrentemente.
- El padre lee el archivo origen y mediante tuberías le pasa la información del archivo a los hijos.
- Cada hijo crea el archivo de destino y escribe la información que le ha pasado el padre.

Ejercicio 3. Comunicación entre procesos: memoria compartida. (**2 puntos**)

Diseña un programa llamado **hijos.c** que cree un árbol de procesos según la siguiente estructura a partir de dos parámetros

```
$hijos x y
```



El proceso **hijos** debe mostrar el siguiente mensaje:

"Soy el superpadre (pid) : mis hijos finales son: `pidx1, pidx2, pidx3, …, pidxy`"

Los procesos de la parte baja deben mostrar el siguiente mensaje**:**

"Soy el subhijo pidx1, mi padres son: `pid1, pid2, …, pidx`"

NORMAS DE ENTREGA

- La práctica es individual o por parejas.
- La entrega de la práctica será durante la semana del **17 al 21 de octubre**.