PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE INGENIERÍA

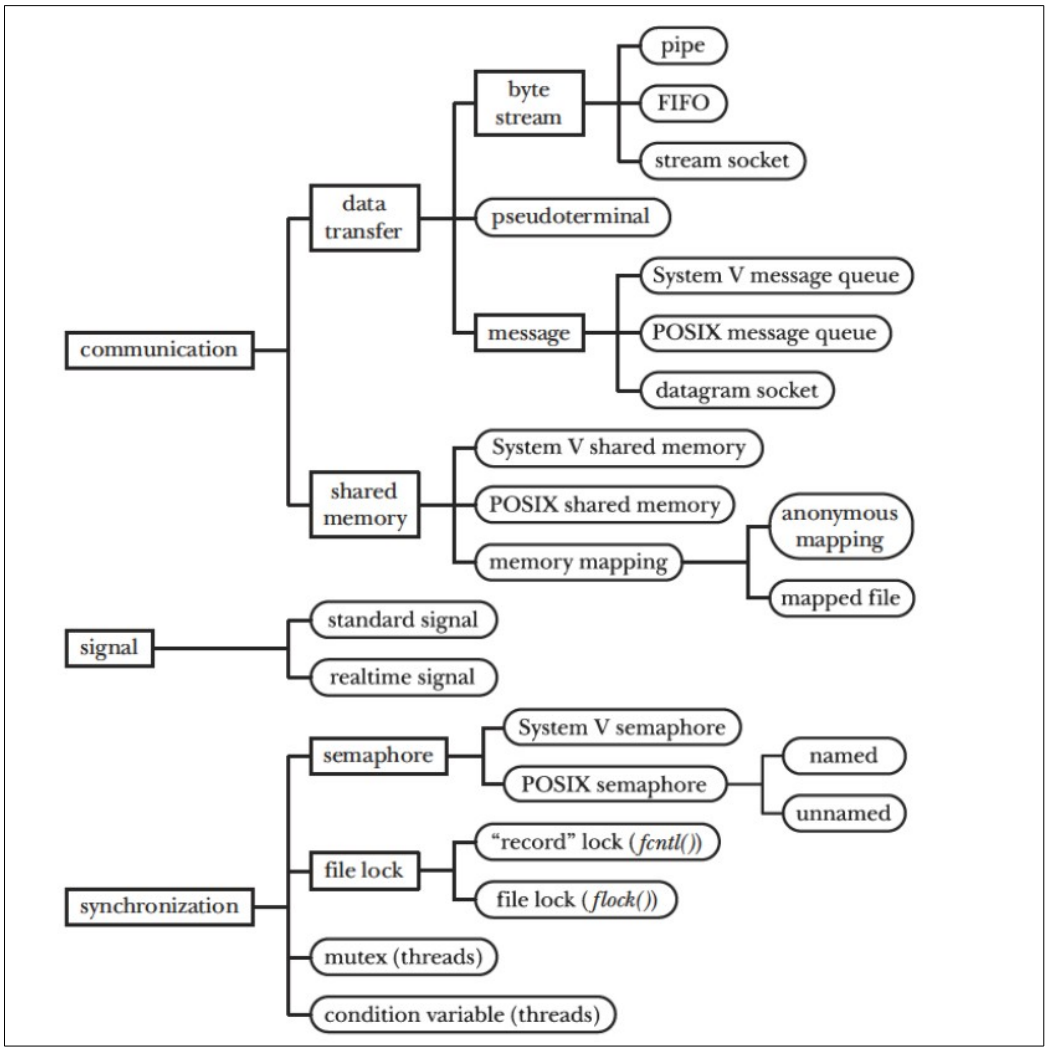
SECCIÓN DE INGENIERÍA INFORMÁTICA
SISTEMAS OPERATIVOS

INTERPROCESS COMMUNICATION (IPC)[1]

PIPES y FIFOS

1. - Descripción general de comunicación entre procesos

La figura de abajo muestra la rica variedad de facilidades de comunicación de UNIX (y también de todo UNIX Like). Nuestro propósito no es explicar cada uno de ellos, simplemente es mostrar el espectro total de estas facilidades que posee y ubicar dónde se encuentran los *pipes* y los *fifos*.



Una taxonomía de las facilidades proporcionadas por UNIX IPC

1 El presente material ha sido tomado del libro *The Linux Programming Interface by Michael Kerrisk*

Los *pipes* son los más antiguos métodos de IPC en los sistemas UNIX, habiendo aparecido en la Tercera Edición UNIX a principios de los 70s. Los *pipes* proveen una elegante solución a u requerimiento frecuente: habiendo creado dos procesos para ejecutar diferentes programas, ¿cómo puede el *shell* permitir que la salida producida por un proceso sea usado por la entrada de otro proceso? Los pipes pueden ser usados para pasar datos entre procesos emparentados (recuerde que los procesos creados forman una árbol de procesos emparentados entre ellos). Los FIFOS son una variación del concepto de *pipe*. La diferencia importante es que los FIFOs pueden ser usados entre cualquier proceso.

Los usuarios del *shell* están familiarizados con el uso de pipes en ordenes tales como la siguiente, la misma que cuenta el número de archivos en un directorio:

```
ls | wc -l
```

Con el fin de ejecutar la orden de arriba, el *shell* crea dos procesos, ejecutando *ls* y *wc*, respectivamente. Esto lo lleva acabo usando fork() y exec(), como lo explicaremos después. La siguiente figura muestra cómo los dos procesos emplean el *pipe*.
Entre otras cosas la figura está pensada para ilustrar cómo los pipes obtuvieron ese nombre. Se puede pensar de un pipe como una pieza de gasfitería que permite que los datos fluyan de un proceso a otro.

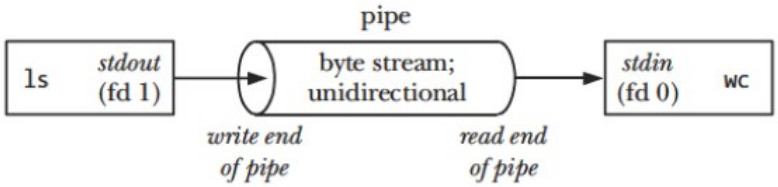


Fig. 1 Usando un *pipe* para conectar dos procesos

Un punto a notar en la figura es que ambos procesos están conectados al *pipe* de modo que el proceso que escribe (*ls*) tiene su salida estándar (descriptor de archivo 1) unido al extremo de escritura del *pipe*, mientras que el proceso que lee (*wc*) tiene su entrada estándar (descriptor de archivo 0) unido al extremo de lectura del *pipe*. Esto quiere decir que ambos procesos no son conscientes de la existencia del *pipe*; ellos solo leen y escriben desde y hacia los descriptores de archivos estándar. Por supuesto el *shell* debe de hacer algún trabajo para que todo esto se llegue a realizar. Nosotros mostraremos después el código para llevar a cabo esta tarea.

Un pipe es un flujo de bytes
Esto significa que el proceso que está leyendo de un *pipe* puede leer bloques de datos de cualquier tamaño, sin importar el tamaño de los bloques escritos por el proceso que está escribiendo. Además, los datos pasan a través del *pipe* secuencialmente, es decir, los bytes se leen de un *pipe* exactamente en el orden en que fueron escritos. En el *pipe* no es posible un acceso aleatorio de los datos usando *lseek()*.

Leyendo desde un *pipe*
Intentar leer desde un *pipe* que está vacío bloqueará el proceso, que intenta leer, hasta que al menos un byte haya sido escrito al *pipe*. Si se cierra (*close()*) el extremo de escritura del *pipe*, entonces el proceso que está leyendo del *pipe* verá un final de archivo (end-of-file) una vez que haya leído todo el resto de los datos del *pipe*. Recuerde que un final de archivo significa que la llamada al sistema *read()* devuelve 0.

Los *pipes* son unidireccionales
Los datos solo pueden viajar en una dirección en el pipe. Un extremo es usado para escritura, y el otro extremo para escritura.

Los *pipes* tienen un límite de capacidad
Un *pipe* es simplemente un *buffer* mantenido en la memoria del kernel. Este *buffer* tiene una capacidad máxima. Una vez que el *pipe* está lleno, escrituras adicionales al pipe bloquearán al proceso que escribe hasta que el lector remueva algún dato del *pipe*. A partir de Linux 2.6.11 la capacidad del *pipe* es de 65536 bytes (16 páginas de 4 KB). A partir de Linux 2.6.35 el tamaño permanece igual, pero se puede modificar a través de la llamada al sistema *fcntl()*.

En general, una aplicación nunca necesita saber la capacidad exacta de un *pipe*. Si se desea evitar que el proceso que escribe se bloquee, el proceso que lee del pipe debería ser diseñado para leer datos tan pronto como sea posible.

Creando y usando *pipes*
La llamada al sistema *pipe()* crea un nuevo *pipe*.

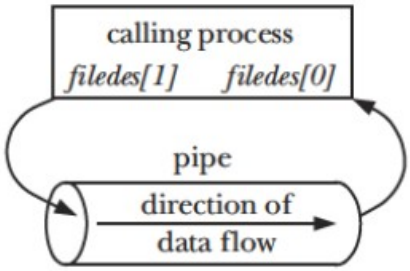
```
#include <unistd.h>

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

Returns 0 on success, or −1 on error

Una llamada exitosa a *pipe()* retorna dos descriptores de archivos en el arreglo *filedes*: uno para el extremo de lectura del *pipe* (*filedes[0]*) y otro para el extremo de escritura (*filesdes[1]*).
Como con cualquier descriptor, podemos usar las llamadas al sistema *read()* y *write()* para llevar a cabo E/S en el *pipe*. Una vez que se ha escrito en el extremo de escritura del *pipe*, los datos están inmediatamente disponibles para ser leídos en el extremo de lectura. Un *read()* de un pipe obtiene el menor entre el número de bytes solicitados y el número disponibles en el *pipe* (pero se bloquea si el *pipe* está vacío).

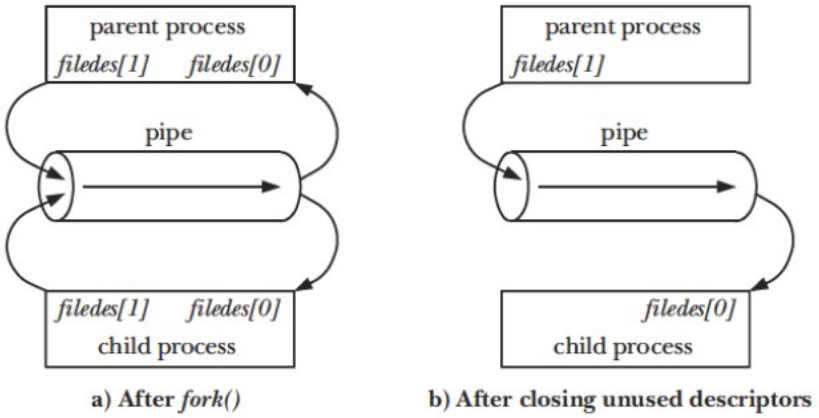
Se puede usar las funciones de *stdio* (*fprintf()*, *fscanf()*, etc) con *pipes*, pero primero se debería usar la función de librería *fdopen()* para obtener el correspondiente flujo de archivo (*FILE* *) para uno de los descriptores de *filedes*. Sin embargo cuando se hace esto se debería ser consciente de los problemas con el *buffer* (tema tratado en clase del curso).

La figura de abajo muestra la situación después que un *pipe* ha sido creado por *pipe()*, donde el proceso que invoca tiene los descriptores de archivo refiriendo a cada extremo del *pipe*.



Descriptores de archivo del proceso después de crear un pipe

Un *pipe* tiene pocos usos con un solo proceso. Lo normal es usar un *pipe* para permitir la comunicación entre dos procesos. Para conectar dos procesos usando un *pipe*, seguido a la llamada *pipe()* se invoca a *fork()*. Durante un *fork()*, el proceso hijo hereda una copia de los descriptores de archivo del padre, la nueva situación es mostrada en la parte izquierda de la figura de abajo.



Estableciendo un pipe para transferir datos de un padre a su hijo

Si bien es posible que tanto el padre como el hijo lean y escriba en el *pipe* al mismo tiempo, esto no es usual. Por tanto, inmediatamente después del *fork()*, un de los procesos cierra su descriptor de escritura en el extremo del *pipe*, y el otro cierra su descriptor de lectura en el otro extremo del *pipe*. Por ejemplo, si el padre va a enviar datos al hijo, entonces debería cerrar su descriptor de lectura para el *pipe*, *filedes[0]*, mientras que el hijo debería cerrar su descriptor de escritura para el *pipe*, *filedes[1]*, obteniéndose la situación mostrada en la parte derecha de la figura de arriba.
El código que crea esta configuración se muestra en *prog1.c*

```
  GNU nano 4.8                        prog1.c                          Modified
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
     int filedes[2];

     if (pipe(filedes) == -1)               /* Crea el pipe */
             perror("pipe");

     switch (fork()) {                      /* Creaun proceso hijo */
         case -1:
             perror("fork");

         case 0:  /* Hijo */
             if (close(filedes[1]) == -1)    /* Cierra descriptor no usado */
                 perror("close");

             /* Hijo ahora lee del pipe */
             break;
         default: /* Padre */
             if (close(filedes[0]) == -1)    /* Cierra descriptor no usado*/
                 perror("close");

             /* Padre ahora escribe en el pipe  */
             break;
      }
}
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Paste Text^T To Spell  ^_ Go To Line
```

Una razón por la que no es usual tener tanto al padre como al hijo leer de un único *pipe* es que si los dos procesos tratan simultáneamente de leer del *pipe*, no se puede asegurar cuál proceso será primero en acceder, los dos procesos compiten por los datos. Evitar tales competencias requiere el uso de algún mecanismo de sincronización. Sin embargo, si se requiere comunicación bidireccional, hay una manera simple. Simplemente se crean dos *pipes*, uno para enviar los datos en cada dirección entre los dos procesos. Si empleamos esta técnica, entonces se necesita ser cauteloso de un *deadlock* pueda ocurrir si ambos procesos se bloquean si intentan leer un pipe vacío o si ambos tratan de escribir a *pipes* que está llenos.

Si bien es posible tener múltiples procesos escribiendo a un *pipe*, lo normal es tener solo un proceso escribiendo.

Los *pipes* permiten comunicación entre procesos emparentados

Hasta ahora se ha mencionado sobre usar *pipes* para la comunicación entre procesos padre e hijo. Sin embargo, los *pipes* pueden ser usados para la comunicación entre dos (o más) procesos emparentados, siempre que el *pipe* haya sido creado por un ancestro común, antes de invocar la serie de llamadas *fork()* que conduce a la existencia de los procesos. Por ejemplo, un *pipe* puede ser usado para comunicación entre un proceso y su nieto. El primer proceso crea el *pipe*, y luego crea un hijo que a su vez crea un hijo para dar paso al nieto. Un escenario común es que el *pipe* sea usado para comunicar entre hermanos, su padre crea el *pipe*, y luego crea los dos hijos.

Cerrando descriptores de archivos de pipe no usados

Cerrar los descriptores de archivos del *pipe* no usados es más que una cuestión de asegurar que un proceso no agote su conjunto limitado de descriptores de archivos, sino que es esencial para el uso correcto de los *pipes*. A continuación se explica por qué debemos considerar cerrar los descriptores de archivos que no son usados.

El proceso que está leyendo del *pipe* cierra su descriptor de escritura para el *pipe*, de forma que, cuando el otro proceso complete su salida y cierre su descriptor de escritura, el escritor vea el *end-of-file* (una vez que haya leído cualquier dato pendiente en el *pipe*).

Si el proceso que está leyendo no cierra el extremos de escritura del *pipe*, entonces, después que el otro proceso cierre su descriptor de escritura, el lector no verá el *end-of-file*, aún después de haber leído toda la data del *pipe*. En su lugar, un *read()* debería bloquear al proceso esperando por datos, porque el *kernel* sabe que hay al menos un descriptor de escritura abierto para el *pipe*. Que este descriptor se mantenga abierto por el proceso de escritura es irrelevante; en teoría ese proceso aún puede escribir al *pipe*, aún si este es bloqueado tratando de leer. Por ejemplo, el *read()* puede ser interrumpido por un manejador de señales que escribe datos al *pipe*.

El proceso que escribe cierra su descriptor de lectura para el pipe por una razón diferente. Cuando un proceso trata de escribir a un *pipe* para el cual ningún proceso tiene abierto un descriptor de lectura, el *kernel* envía la señal `SIGPIPE` al proceso que escribe. Por defecto, esta señal mata el proceso. Un proceso puede atrapar o ignorar esta señal, en cuyo caso el *write()* en el *pipe* falla con error `EPIPE` (*broken pipe*). Recibiendo la señal `SIGPIPE` u obteniendo el error `EPIPE` es un indicador útil acerca del estado del *pipe*, y esta es la razón de por qué los descriptores no usados de lectura para el *pipe* deberían ser cerrados.

Si el proceso que está escribiendo no cierra el extremo de lectura del pipe, entonces, aún después que el otro proceso cierre el extremos de lectura del pipe, el proceso que está escribiendo aún le será posible escribir al pipe. Eventualmente, el proceso que está escribiendo llenará el pipe, y un intento más de escritura se bloqueará indefinidamente.

Una razón final para cerrar los descriptores de archivos no usados es que solo después que todos los descriptores de archivos en

Una razón final para cerrar los descriptores de archivos no utilizados es que solo después de que se cierran todos los descriptores de archivos en todos los procesos que hacen referencia a una tubería, la tubería se destruye y sus recursos se liberan para su reutilización por otros procesos. En este punto, cualquier dato no leído en el pipe se pierde.

Programa ejemplo

El siguiente es programa muy simple, donde el proceso padre envía una cadena al proceso hijo que ha creado con *fork()* inmediatamente después de crear el *pipe*. La cadena enviada a través del *pipe* es tomada de la línea de ordenes.

```
alulab@minix:~$ gcc -o simple_pipe simple_pipe.c
alulab@minix:~$ ./simple_pipe "Este es el segundo laboratorio de SO"
Este es el segundo laboratorio de SO
alulab@minix:~$
```

```c

/* simple_pipe.c */

#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define BUF_SIZE 10

int
main(int argc, char *argv[])
{
    int pfd[2];                             /* Pipe file descriptors */
    char buf[BUF_SIZE];
    ssize_t numRead;

    if (argc != 2 || strcmp(argv[1], "--help") == 0){
        fprintf(stderr,"%s string\n", argv[0]);
        exit(1);
    }

    if (pipe(pfd) == -1)                    /* Create the pipe */
        perror("pipe");

    switch (fork()) {
    case -1:
        perror("fork");

    case 0:             /* Child  - reads from pipe */
```

```
33          if (close(pfd[1]) == -1)            /* Write end is unused */
34              perror("close - child");
35
36          for (;;) {              /* Read data from pipe, echo on stdout */
37              numRead = read(pfd[0], buf, BUF_SIZE);
38              if (numRead == -1)
39                  perror("read");
40              if (numRead == 0)
41                  break;                      /* End-of-file */
42              if (write(STDOUT_FILENO, buf, numRead) != numRead)
43                  perror("child - partial/failed write");
44          }
45
46          write(STDOUT_FILENO, "\n", 1);
47          if (close(pfd[0]) == -1)
48              perror("close");
49          _exit(EXIT_SUCCESS);
50
51      default:            /* Parent - writes to pipe */
52          if (close(pfd[0]) == -1)            /* Read end is unused */
53              perror("close - parent");
54
55          if (write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]))
56              perror("parent - partial/failed write");
57
58          if (close(pfd[1]) == -1)            /* Child will see EOF */
59              perror("close");
60          wait(NULL);                         /* Wait for child to finish */
61          exit(EXIT_SUCCESS);
62      }
63  }
64
```

Usando pipes para conectar filtros

Cuando se crea un pipe, los descriptores de archivos usados para los dos extremos del pipe son los siguientes descriptores con el número más bajo disponible. Dado que, en circunstancias normales, los descriptores 0, 1 y 2 ya están en uso por un proceso, se asignarán algunos descriptores con números más altos para la tubería. Entonces ¿cómo se logrará la situación que se muestra en la Fig 1, donde dos filtros (esto es, programas que leen de *stdin* y escriben a *stdout*) son conectadas usando un *pipe*, de modo tal que la salida estándar de un programa se dirige al pipe y la entrada estándar del otro es tomada del pipe? Y en particular, ¿cómo se puede hacer sin modificar el código el código de los filtros mismos?

La respuesta es usar la técnica de duplicar los descriptores de archivo. Tradicionalmente, la siguiente serie de llamadas fueron usadas para llevar a cabo el resultado deseado:

```
int pfd[2];

pipe(pfd);   /* Allocates (say) file descriptors 3 and 4 for pipe */

/* Other steps here, e.g., fork() */

close(STDOUT_FILENO);           /* Free file descriptor 1 */
dup(pfd[1]);                    /* Duplication uses lowest free file
                                   descriptor, i.e., fd 1 */
```

El resultado final de los pasos de arriba es que la salida estándar del proceso es ligado al extremo de escritura del pipe. Un correspondiente conjunto de llamadas se pueden usar para ligar la entrada estándar de un proceso al extremo de lectura del pipe.

Observe que estos pasos se basan en que asumimos que los descriptores 0, 1 y 2 para un proceso ya están abiertos. (El *shell* normalmente asegura esto para cada programa que ejecuta) Si el descriptor de archivo 0 ya fue cerrado antes de los pasos de arriba, entonces enlazaríamos erróneamente la entrada estándar del proceso al extremo de escritura del pipe. Para evitar esta posibilidad, se puede reemplazar las llamadas *close()* y *dup()* con la siguiente llamada *dup2()*, el cual nos permite especificar el descriptor a ser enlazado al extremo del *pipe*:

```
dup2(pfd[1], STDOUT_FILENO);    /* Close descriptor 1, and reopen bound
                                   to write end of pipe */
```

Después de duplicar pfd[1], ahora se tiene dos descriptores refiriéndose al extremos de escritura del pipe: el descriptor 1 y pfd[1]. Dado que los descriptores de archivos no usados deberían ser cerrados, después de la llamada dup2(), se cierra el descriptor superfluo:

```
close(pfd[1]);
```

El código mostrado hasta ahora descansa en la confianza de que la salida estándar ha sido previamente abierta. Suponga que , antes de la llamada al pipe(), tanto la entrad estándar como la salida estándar han sido cerradas. En este caso, pipe() debería haber asignado estos dos descriptores al pipe, quizás con pfd[0] teniendo el valor 0 y pfd[1] teniendo el valor 1. consecuentemente, las anteriores llamadas dup2(9 y close() deberían ser equivalente a lo siguiente:

```
dup2(1, 1);             /* Does nothing */
close(1);               /* Closes sole descriptor for write end of pipe */
```

Por tanto, es una buena práctica de programación defensiva poner entre corchetes con una sentencia if de la siguiente forma:

```
if (pfd[1] != STDOUT_FILENO) {
    dup2(pfd[1], STDOUT_FILENO);
    close(pfd[1]);
}
```

Programa ejemplo

El siguiente programa usa la técnica descrita arriba para lograr la configuración de la Fig 1. Después de construir un *pipe*, este programa crea dos procesos hijos. El primer hijo liga su salida estándar al extremo de escritura del *pipe* y luego ejecuta (*exec*) *ls*. El segundo hijo liga su entrada estándar al extremo de lectura del *pipe* y luego ejecuta (*exec*) *wc*.

```c

/* pipe_ls_wc.c  */
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    int pfd[2];                                 /* Pipe file descriptors */

    if (pipe(pfd) == -1)                        /* Create pipe */
        perror("pipe");

    switch (fork()) {
    case -1:
        perror("fork");

    case 0:             /* First child: exec 'ls' to write to pipe */
        if (close(pfd[0]) == -1)                /* Read end is unused */
            perror("close 1");

        /* Duplicate stdout on write end of pipe; close duplicated descriptor */

        if (pfd[1] != STDOUT_FILENO) {          /* Defensive check */
            if (dup2(pfd[1], STDOUT_FILENO) == -1)
                perror("dup2 1");
            if (close(pfd[1]) == -1)
                perror("close 2");
        }

        execlp("ls", "ls", (char *) NULL);      /* Writes to pipe */
        perror("execlp ls");

    default:            /* Parent falls through to create next child */
        break;
    }

    switch (fork()) {
    case -1:
        perror("fork");

    case 0:             /* Second child: exec 'wc' to read from pipe */
        if (close(pfd[1]) == -1)                /* Write end is unused */
            perror("close 3");

        /* Duplicate stdin on read end of pipe; close duplicated descriptor */

        if (pfd[0] != STDIN_FILENO) {           /* Defensive check */
            if (dup2(pfd[0], STDIN_FILENO) == -1)
                perror("dup2 2");
            if (close(pfd[0]) == -1)
                perror("close 4");
        }

        execlp("wc", "wc", "-l", (char *) NULL);
        perror("execlp wc");

    default: /* Parent falls through */
        break;
    }

    /* Parent closes unused file descriptors for pipe, and waits for children */

    if (close(pfd[0]) == -1)
        perror("close 5");
    if (close(pfd[1]) == -1)
        perror("close 6");
    if (wait(NULL) == -1)
        perror("wait 1");
    if (wait(NULL) == -1)
        perror("wait 2");

    exit(EXIT_SUCCESS);
}
```

- 1 -

FIFOs

Semánticamente, un FIFO es similar a un *pipe*. La principal diferencia es que el FIFO tiene un nombre dentro del sistema de archivos y se abre de la misma manera que un archivo normal. Esto le permite a un FIFO que sea usado para comunicaciones entre procesos no emparentados (por ejemplo cliente servidor)

Una vez que un FIFO haya sido abierto, se usa las mismas llamadas al sistema que son usadas con pipes y otros archivos (esto es, *read()*, *write()*, y *close()*). Al igual que los pipes, un FIFO tiene un extremo de escritura y un extremo de lectura, y los datos se leen del *pipe* en el mismo orden en que se son escritos. Este hecho le da a los FIFOs su nombre: *first in*, *first out*. Los FIFOS también son conocidos como *named pipes*.

Como con los pipes, cuando todos los descriptores referentes al FIFO has sido cerrados, cualquier datos restante se descarta.

La función *mkfifo()* crea un nuevo FIFO con el nombre de ruta (*pathname*) proporcionado.

```
#include <sys/stat.h>

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

Returns 0 on success, or −1 on error

El argumento mode especifica los permisos para el nuevo FIFO. Estos permisos son especificados de la combinación deseada mediante el conector OR de constantes de la tabla de abajo.

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

Una vez que el FIFO ha sido creado, cualquier proceso puede abrirlo, sujeto a los chequeos usuales de permisos de archivos.

Abrir un FIFO tiene una semántica algo inusual. Generalmente, el único uso sensato de un FIFO es tener un proceso de lectura y un proceso de escritura en cada extremo. Por tanto, por defecto, abrir un FIFO de lectura (el *flag* `O_RDONLY` en el *open()*) se bloquea hasta que otro proceso abra el FIFO para escritura (el flag `WRONLY` del *open()*). A la inversa, abrir el FIFO para escritura ocasione que se

bloquee hasta que otro proceso abra el FIFO para lectura. En otras palabras, abrir un FIFO sincroniza los procesos de lectura y escritura. Si el extremo opuesto de un FIFO ya está abierto (quizás porque un par de procesos ya han abierto cada extremo del FIFO), entonces *open()* tiene éxito inmediatamente.

Bajo muchas implementaciones UNIX (incluyendo Linux), es posible evadir el comportamiento bloqueante cuando se abre un FIFO especificando el *flag* `O_RDWR`. En este caso, *open()* retorna inmediatamente con un descriptor de archivo que puede ser usado para leer y escribir en el FIFO. Esta técnica debería se evitada, en principio por razones de portabilidad. En circunstancias donde se necesite evitar bloquearse cuando se abre un FIFO, el *flag* `O_NONBLOCK` de *open()* provee un método estándar para hacerlo.

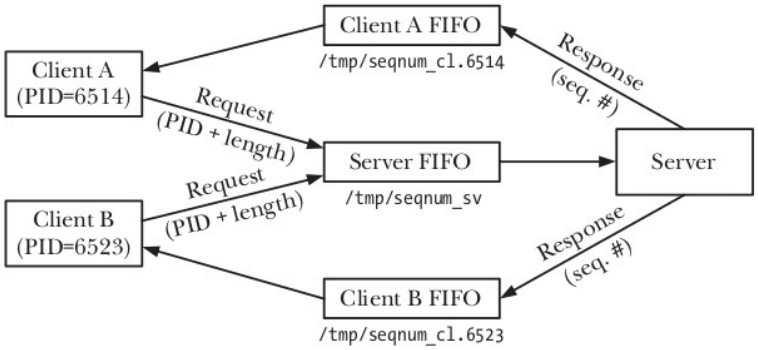
En este laboratorio no usaremos `O_NONBLOCK`.

Una aplicación Cliente-Servidor usando FIFOs
A continuación se presentará una simple aplicación cliente-servidor que emplea FIFOS para IPC. El servidor provee el servicio trivial de asignar un número secuencial a cada cliente que le solicita el servicio.

En la aplicación ejemplo, todos los clientes envían sus solicitudes al servidor usando un único FIFO. El archivo de cabecera *fifo_sqnum.h* define el nombre conocido (*/tmp/seqnum_sv*) que el servidor usa para su FIFO. Este nombre es fijo, de modo que todos los clientes conozcan cómo contactar el servidor. (En este ejemplo se crea el FIFO en */tmp*, sin embargo en aplicaciones reales no es aconsejable usar este directorio por razones de vulnerabilidad.)

Sin embargo, no es posible emplear un único FIFO para enviar todas las respuestas a todos los clientes, debido a que múltiples clientes estarían compitiendo por el FIFO, y posiblemente lea los mensajes de los otros en lugar del suyo propio. Por tanto cada cliente crea un único FIFO que el servidor usa para repartir las respuestas a cada cliente, y el servidor necesita saber cómo encontrar cada FIFO del cliente. Una posible forma de llevar a cabo esto es que cada cliente genere su propia ruta de nombre FIFO, y luego pase la ruta de nombre al servidor como parte del mensaje de solicitud. Alternativamente, el servidor y el cliente pueden acordar una convención para construir la ruta de nombre del FIFO cliente, y, como parte de su mensaje de solicitud, el cliente puede pasar la información requerida para construir la ruta de nombre especifico para este cliente. Esta última solución será usada en es te ejemplo. Cada nombre FIFO del cliente es construido de una plantilla (`CLIENT_FIFO_TEMPLATE`) consistiendo de una ruta de nombre conteniendo el *pid* del cliente. La inclusión del pid del proceso provee una forma fácil de generar nombres únicos para cada cliente.

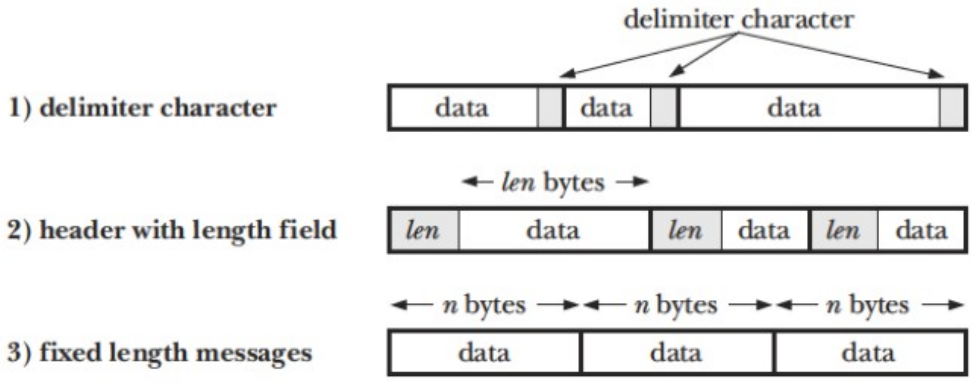
La figura de abajo muestra cómo esta aplicación usa FIFOs para la comunicación entre los procesos clientes y el servidor.



Hay que recordar que los datos en los pipes y en los FIFOS son flujos de bytes; no existe límites entre los mensajes que se envían. Esto significa que cuando múltiples mensajes están siendo repartidos a un único proceso, tal como el servidor en nuestro ejemplo, entonces tanto el que envía como el que recibe debe acordar en alguna convención para separar los mensajes. Varios enfoques son posibles:

a) Terminar el mensaje con un carácter delimitador, tal como el cambio de línea. En cuyo caso el delimitador nunca puede ser parte del mensaje.
b) Incluir una cabecera de tamaño fijo que indique la longitud del campo en cada mensaje especificando el número de bytes en el componente restante de longitud variable del mensaje.
c) Usar mensajes de longitud fija, y tener al servidor siempre leyendo mensajes de tamaño fijo.

Estas tres técnicas se ilustran en el dibujo de abajo:



Programa servidor
El servidor lleva acabo los siguientes pasos:

a) Crea el FIFO del servidor y lo abre para lectura. El servidor debe de estar corriendo antes que cualquier cliente, de modo que el FIFO servidor exista cuando algún cliente intente leerlo. El *open()* del servidor se bloquea hasta que en el otro extremo se abra el FIFO servidor para escritura.

b) Abre el FIFO servidor una vez más, esta vez para escritura. Esto nunca se bloqueará, debido a que el FIFO ya ha sido abierto para escritura. Este segundo *open()* es conveniente para asegurar que el servidor no vea el *end-of-file* si todos los clientes cierran el extremo de escritura del FIFO.

c) Ignora la señal `SIGPIPE`, de modo que si el servidor intenta escribir a un cliente FIFO que no tiene un lector, entonces, entonces en lugar de que se le envíe una señal `SIGPIPE` (el cual mataría el proceso por defecto), este recibe un error `EPIPE` de la llamada al sistema *write()*.

d) Ingresa a un lazo que lee y responde a cada solicitud de cliente que llega. Para enviar la respuesta, el servidor construye el nombre del FIFO cliente y abre ese FIFO.

e) Si el servidor encuentra un error al abrir el FIFO cliente, este abandona la solicitud de ese cliente.

Este es un ejemplo de un *servidor iterativo*, en el cual el servidor lee y maneja cada solicitud del cliente antes de ir a atender el siguiente cliente. Este enfoque es adecuado cuando el servidor puede procesar y responder muy rápidamente las solicitudes del cliente sin retardo. Una alternativa es diseñar un *servidor concurrente*, en el cual el proceso servidor principal emplea un proceso hijo (o hilo) para manejar cada solicitud del cliente.

```c
/* fifo_seqnum_server.c */

#include <signal.h>
#include "fifo_seqnum.h"

int
main(int argc, char *argv[])
{
    int serverFd, dummyFd, clientFd;
    char clientFifo[CLIENT_FIFO_NAME_LEN];
    struct request req;
    struct response resp;
    int seqNum = 0;                     /* This is our "service" */
    char msgError[100];

    /* Create well-known FIFO, and open it for reading */

    umask(0);                           /* So we get the permissions we want */
    if (mkfifo(SERVER_FIFO, S_IRUSR | S_IWUSR | S_IWGRP) == -1
            && errno != EEXIST)
        sprintf(msgError,"mkfifo %s", SERVER_FIFO),perror(msgError);
    serverFd = open(SERVER_FIFO, O_RDONLY);
    if (serverFd == -1)
        sprintf(msgError,"open %s", SERVER_FIFO),perror(msgError);

    /* Open an extra write descriptor, so that we never see EOF */

    dummyFd = open(SERVER_FIFO, O_WRONLY);
    if (dummyFd == -1)
        sprintf(msgError,"open %s", SERVER_FIFO),perror(msgError);

    /* Let's find out about broken client pipe via failed write() */

    if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        perror("signal");

    for (;;) {                          /* Read requests and send responses */
        if (read(serverFd, &req, sizeof(struct request))
                != sizeof(struct request)) {
            fprintf(stderr, "Error reading request; discarding\n");
            continue;                   /* Either partial read or error */
        }

        /* Open client FIFO (previously created by client) */

        snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
                (long) req.pid);
        clientFd = open(clientFifo, O_WRONLY);
        if (clientFd == -1) {           /* Open failed, give up on client */
            printf("open %s", clientFifo);
            continue;
        }

        /* Send response and close FIFO */

        resp.seqNum = seqNum;
        if (write(clientFd, &resp, sizeof(struct response))
                != sizeof(struct response))
            fprintf(stderr, "Error writing to FIFO %s\n", clientFifo);
        if (close(clientFd) == -1)
            printf("close");

        seqNum += req.seqLen;           /* Update our sequence number */
    }
}
```

- 1 -

Programa cliente

El cliente lleva a cabo los siguientes pasos:

a) Crea un FIFO para ser usado como receptor de las respuestas del servidor. Esto se hace antes de enviar la solicitud, con el fin de asegurar que el FIFO exista al momento que el servidor intenta abrirlo para enviar la respuesta al cliente.

b) Se construye un mensaje para el servidor conteniendo el *pid* del cliente y un número (tomado opcionalmente desde la línea de ordenes) especificando la longitud de la secuencia que el cliente desea que el servidor le asigne. (Si no se proporciona, la longitud de la secuencia es 1)

c) Abre el FIFO servidor y envía el mensaje al servidor.

d) Abre el FIFO cliente, lee e imprime la respuesta del servidor.

El único detalle a notar es el manejador de salida, establecido con *atexit()*, el cual asegura que el FIFO del cliente es borrado cuando el proceso sale.

```c
/* fifo_seqnum_client.c */

#include "fifo_seqnum.h"

static char clientFifo[CLIENT_FIFO_NAME_LEN];

static void             /* Invoked on exit to delete client FIFO */
removeFifo(void)
{
    unlink(clientFifo);
}

int
main(int argc, char *argv[])
{
    int serverFd, clientFd;
    struct request req;
    struct response resp;
    char msgError[100];

    if (argc > 1 && strcmp(argv[1], "--help") == 0) {
        printf("Usage: %s [seq-len...]\n", argv[0]);
        exit(1);
    }
    /* Create our FIFO (before sending request, to avoid a race) */

    umask(0);                   /* So we get the permissions we want */
    snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
            (long) getpid());
    if (mkfifo(clientFifo, S_IRUSR | S_IWUSR | S_IWGRP) == -1
                && errno != EEXIST)
        sprintf(msgError,"mkfifo %s", clientFifo),perror(msgError);

    if (atexit(removeFifo) != 0)
        perror("atexit");

    /* Construct request message, open server FIFO, and send message */

    req.pid = getpid();
    req.seqLen = (argc > 1) ? atoi(argv[1]) : 1;

    serverFd = open(SERVER_FIFO, O_WRONLY);
    if (serverFd == -1)
        sprintf(msgError,"open %s", SERVER_FIFO),perror(msgError);

    if (write(serverFd, &req, sizeof(struct request)) !=
            sizeof(struct request))
        perror("Can't write to server");

    /* Open our FIFO, read and display response */

    clientFd = open(clientFifo, O_RDONLY);
    if (clientFd == -1)
        sprintf(msgError,"open %s", clientFifo),perror(msgError);

    if (read(clientFd, &resp, sizeof(struct response))
            != sizeof(struct response))
        perror("Can't read response from server");

    printf("%d\n", resp.seqNum);
    exit(EXIT_SUCCESS);
}
```

Ambos programas hacen uso del archivo de cabecera *fifo_seqnum.h* mostrado a continuación:

```c
/* fifo_seqnum.h */

#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>

#define SERVER_FIFO "/tmp/seqnum_sv"
                                /* Well-known name for server's FIFO */
#define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%ld"
                                /* Template for building client FIFO name */
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
                                /* Space required for client FIFO pathname
                                   (+20 as a generous allowance for the PID) */

struct request {                /* Request (client --> server) */
    pid_t pid;                  /* PID of client */
    int seqLen;                 /* Length of desired sequence */
};

struct response {               /* Response (server --> client) */
    int seqNum;                 /* Start of sequence */
};
```

A continuación se muestra un ejemplo de lo que se espera cuando se ejecuta el servidor y los clientes.

```
alulab@minix:~/Documents$ gcc -o fifo_seqnum_client fifo_seqnum_client.c
alulab@minix:~/Documents$ gcc -o fifo_seqnum_server fifo_seqnum_server.c
alulab@minix:~/Documents$ ./fifo_seqnum_server &
[1] 9154
alulab@minix:~/Documents$ ./fifo_seqnum_client 3
0
alulab@minix:~/Documents$ ./fifo_seqnum_client 2
3
alulab@minix:~/Documents$ ./fifo_seqnum_client
5
alulab@minix:~/Documents$
```

Daemons

Un demonio es un programa que se ejecuta como un proceso en segundo plano (o de fondo, sin una terminal o interfaz de usuario), comúnmente esperando que ocurran eventos y ofreciendo servicios. Un buen ejemplo es un servidor *web* que espera una solicitud para entregar una página, o un servidor *ssh* esperando a que alguien intente iniciar sesión. Si bien estas son aplicaciones completas, hay demonios cuyo trabajo no es tan visible. Los demonios son para tareas como escribir mensajes en un archivo de registro (por ejemplo, *syslog*, *metalog*) o para mantener la precisión del reloj de su sistema (por ejemplo, *ntpd*). Para obtener más información, véase *daemon(7)*. (Tomado de *https://wiki.archlinux.org/index.php/Daemons_(Espanol)*

Sobre el nombre usted puede encontrar interesante leer los siguientes enlaces:

The jergon file. *Eric S. Raymond*. "*daemon*"
"*Take Our Word for it*" Fernando Corbató

Los pasos para conseguir que un proceso llegue a ser un daemon, son los siguientes:

a) El proceso padre crea un hijo y termina si todo ha salido bien. Debido a que el proceso padre ha terminado el proceso hijo está ahora ejecutándose en *background* adoptado por *systemd.*

b) El proceso hijo invoca *setsid()* para iniciar una nueva sesión y se libere de cualquier asociación con una terminal de control.

c) Si el demonio nunca abre ningún dispositivo terminal a partir de entonces, no es necesario que nos preocupemos de que el demonio vuelva a adquirir una terminal de control. Si el demonio puede después abrir un dispositivo de terminal, entonces debemos tomar los pasos para asegurarnos que el dispositivo no llegue a ser convierta en la terminal de control. Una manera sencilla de hacer esto es hacer un segundo *fork()* después de la llamada *setsid()*, y nuevamente el padre termina (*exit()*) y el nieto continúa, de esta forma el proceso nunca podrá adquirir una terminal de control.

d) Limpiar el *umask* del proceso, para asegurar que, cuando el demonio cree archivos y directorios, ellos tengan los permisos requeridos.

e) Cambiar el directorio de trabajo actual, normalmente al directorio raíz (en nuestro caso será un directorio con permisos apropiados, considerando que nuestro demonio será eliminado después de cada prueba).

f) Cerrar todos los descriptores de archivos abiertos que el demonio ha heredado de su padre.

g) Después de haber cerrado los descriptores de archivos 0, 1 y 2, un demonio normalmente abre */dev/null* y usa *dup2()* (o similar) para que todos estos descriptores se refieran a este dispositivo.

A continuación se ha desarrollado la función *becomeDaemon()* contenida en el archivo *become_daemon.c* y una programa, con nombre *test_become_daemon.c* que prueba dicha función. Adicionalmente hay un archivo de cabecera *become_daemon.h*

```c
/* become_daemon.h */

#ifndef BECOME_DAEMON_H             /* Prevent double inclusion */
#define BECOME_DAEMON_H

/* Bit-mask values for 'flags' argument of becomeDaemon() */

#define BD_NO_CHDIR           01    /* Don't chdir("/") */
#define BD_NO_CLOSE_FILES     02    /* Don't close all open files */
#define BD_NO_REOPEN_STD_FDS  04    /* Don't reopen stdin, stdout, and
                                       stderr to /dev/null */
#define BD_NO_UMASK0         010    /* Don't do a umask(0) */

#define BD_MAX_CLOSE  8192          /* Maximum file descriptors to close if
                                       sysconf(_SC_OPEN_MAX) is indeterminate */

int becomeDaemon(int flags);

#endif
```

```c
/* become_daemon.c */
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include "become_daemon.h"


int                                     /* Returns 0 on success, -1 on error */
becomeDaemon(int flags)
{
    int maxfd, fd;

    switch (fork()) {                   /* Become background process */
    case -1: return -1;
    case 0:  break;                     /* Child falls through... */
    default: _exit(EXIT_SUCCESS);       /* while parent terminates */
    }

    if (setsid() == -1)                 /* Become leader of new session */
        return -1;

    switch (fork()) {                   /* Ensure we are not session leader */
    case -1: return -1;
    case 0:  break;
    default: _exit(EXIT_SUCCESS);
    }

    if (!(flags & BD_NO_UMASK0))
        umask(0);                       /* Clear file mode creation mask */

    if (!(flags & BD_NO_CHDIR))
        chdir("/");                     /* Change to root directory */

    if (!(flags & BD_NO_CLOSE_FILES)) { /* Close all open files */
        maxfd = sysconf(_SC_OPEN_MAX);
        if (maxfd == -1)                /* Limit is indeterminate... */
            maxfd = BD_MAX_CLOSE;       /* so take a guess */

        for (fd = 0; fd < maxfd; fd++)
            close(fd);
    }

    if (!(flags & BD_NO_REOPEN_STD_FDS)) {
        close(STDIN_FILENO);            /* Reopen standard fd's to /dev/null */

        fd = open("/dev/null", O_RDWR);

        if (fd != STDIN_FILENO)         /* 'fd' should be 0 */
            return -1;
        if (dup2(STDIN_FILENO, STDOUT_FILENO) != STDOUT_FILENO)
            return -1;
        if (dup2(STDIN_FILENO, STDERR_FILENO) != STDERR_FILENO)
            return -1;
    }

    return 0;
}

```

- 1 -

```c
/* test_become_daemon.c */

#include "become_daemon.h"
#include <unistd.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    becomeDaemon(0);

    /* Normally a daemon would live forever; we just sleep for a while */

    sleep((argc > 1) ? atoi(argv[1]) : 20);

    exit(EXIT_SUCCESS);
}
```

Al ejecutarlo en una terminal, se tiene la siguiente salida

```
alulab@minix:~/Documents$ gcc -o test_become_daemon test_become_daemon.c become_daemon.c
alulab@minix:~/Documents$ ./test_become_daemon
alulab@minix:~/Documents$ ps -p 9807 -o "pid ppid pgid sid tty command"
  PID  PPID  PGID   SID TT       COMMAND
 9807   995  9806  9806 ?        ./test_become_daemon
```

Todo lo anterior, se puede resumir en una sola orden: *daemon()*. Sin embargo esta función de la biblioteca estándar de C no es POSIX. En Linux se encuentra disponible, usted puede encontrar información en el manual en línea (desde una termina: *man daemon*)

```
DAEMON(3)                  Linux Programmer's Manual                 DAEMON(3)

NAME
       daemon - run in the background

SYNOPSIS
       #include <unistd.h>

       int daemon(int nochdir, int noclose);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       daemon():
           Since glibc 2.21:
                _DEFAULT_SOURCE
           In glibc 2.19 and 2.20:
                _DEFAULT_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)
           Up to and including glibc 2.19:
                _BSD_SOURCE || (_XOPEN_SOURCE && _XOPEN_SOURCE < 500)

DESCRIPTION
       The  daemon()  function is for programs wishing to detach themselves from the con-
       trolling terminal and run in the background as system daemons.

       If nochdir is zero, daemon() changes the process's current  working  directory  to
       the  root  directory  ("/");  otherwise, the current working directory is left un-
       changed.

 Manual page daemon(3) line 1 (press h for help or q to quit)
```

TAREA

1) Escriba un programa para que un proceso padre cree dos hijos. Usted debe crear un pipe para que los dos procesos hermanos se intercambien mensajes. Es decir ambos deben ser capaces de enviar y leer mensajes, sin que haya problemas de concurso de competencia.

2) Haciendo uso de FIFOS escriba un programa que lleve a cabo el modelo cliente-servidor. Haciendo uso de *fork()* implemente la idea de un *servidor concurrente*.

3) Modifique el programa anterior para que el servidor se ejecute como un demonio. Haga uso de la función *daemon(3)* de la biblioteca de C