

PRÁCTICA 5

Comunicación entre procesos usando C para linux

| Estudiante | Escuela | Asignatura |
|---|--|---------------------|
| Rodrigo Infanzón Acosta rinfanzona@ulasalle.edu.pe | Carrera Profesional de Ingeniería de Software | Sistemas Operativos |

| Informe | Tema | Duración |
|---------|--|----------|
| 06 | Comunicación entre procesos usando C para linux | 04 horas |

| Semestre académico | Fecha de inicio | Fecha de entrega |
|--------------------|-----------------|------------------|
| 2024 - B | 11/10/24 | 18/10/24 |

Índice

| | |
|---|----|
| 1. Objetivos | 1 |
| 2. Recursos y herramientas utilizados | 2 |
| 3. URL de repositorio Github | 2 |
| 4. Marco teórico | 2 |
| 5. Ejercicios propuestos | 7 |
| 5.1. Ejercicio 1 | 7 |
| 5.2. Ejercicio 2 | 8 |
| 5.3. Ejercicio 3 | 10 |
| 5.4. Ejercicio 4 | 12 |
| 5.4.1. Named Pipes (FIFO) | 12 |
| 5.4.2. Sockets | 12 |
| 5.4.3. Shared Memory (Memoria Compartida) | 12 |
| 5.4.4. Message Queues | 13 |
| 5.4.5. Signals (Señales) | 13 |

1. Objetivos

- El alumno comprenderá y analizará el funcionamiento interno de los sistemas operativos desde la utilización de comandos hasta la programación de procesos.
- El alumno deberá de probar, analizar y explicar el comportamiento de los diferentes medios de comunicación que existen entre procesos para un intercambio de datos de forma segura. A nivel del sistema operativo.

2. Recursos y herramientas utilizados

- Sistema operativo utilizado: Windows 10 pro 22H2 de 64 bits SO. 19045.4170.
- Hardware: Ryzen 7 5700X 4.0 GHz, RAM 32 GB DDR4 3200 MHz, RTX 4060 Asus Dual.
- Virtual Box 7.0.20-163906-Win
- Git 2.44.0.
- Sistema invitado utilizado: Fedora Linux 40 Cinnamon Spin
- Conocimientos básicos en Git.
- Conocimientos básicos en sistemas operativos.
- Conocimientos básicos en Linux.
- C (gcc).

3. URL de repositorio Github

URL para la práctica 5 en el repositorio GitHub:

- <https://github.com/RodrigoStranger/sistemas-operativos-24b/tree/main/Practica%205>

4. Marco teórico

Los procesos en Linux pueden comunicarse a través de memoria compartida o envío de mensajes.

Un mecanismo de comunicación disponible en Linux son los pipes.

Los pipes se generan mediante la función `pipe()`.

La función `pipe()` crea una "tubería" (canal) de comunicación.

Al llamar a `pipe()`, se obtienen dos descriptores de fichero:

- Uno para la lectura.
- Otro para la escritura.

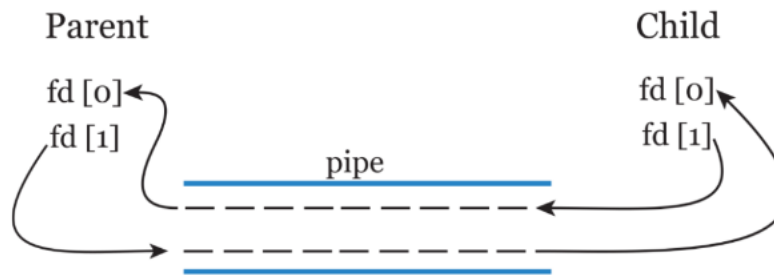
Se puede leer datos desde un extremo (con `read()`) que han sido escritos desde el otro extremo (con `write()`).

La ventaja de crear la tubería antes de `fork()` es que el proceso hijo copia el contenido del proceso padre.

Esto incluye la copia de los descriptores de la tubería, permitiendo que ambos procesos (padre e hijo) puedan comunicarse.

Si ocurre un error al invocar `pipe()`, la función retorna `-1`.

```
1  int descriptorTuberia[2];
2  int pid;
3  pipe(descriptorTuberia);
4  pid = fork();
```



No es recomendable que ambos procesos intenten escribir en la misma tubería, al menos salvo que posean mecanismos de sincronización. De ocurrir esta situación, los datos podrían mezclarse y resultarían en datos erróneos. En el caso de la lectura, si el proceso que ha escrito un dato intenta leer la tubería antes que el proceso hijo, recogerá su propio mensaje. Si ambos intentan leer a la vez, recogerán cada uno trozos del mensaje. Por ello, es mejor utilizar la tubería para que un proceso solo envíe datos y sea el otro sea el único encargado de recibirlos. Si queremos una comunicación bidireccional, es mejor abrir dos tuberías (llamar dos veces a la función `pipe()`).

Para evitar equivocaciones, y sobre todo para no mantener recursos ocupados innecesariamente, el proceso padre debe cerrar el lado de la tubería que no vaya a utilizar. El proceso hijo tiene su propia copia de ese lado, así que aunque el proceso padre lo cierre, para él sigue abierto y utilizable. El proceso hijo debe de cerrar el otro extremo.

En el siguiente ejemplo cerraremos el descriptor de lectura en el proceso padre y el de escritura en el proceso hijo. Una vez hecho esto, el padre puede escribir por `descriptorTuberia[1]` y el hijo leer por `descriptorTuberia[0]`, como si se tratara de un fichero normal. Obviamente ambos procesos deben estar de acuerdo en qué estructuras de datos se van a enviar:

```
1 //Proceso Padre
2 close(descriptorTuberia[0]);
3 ...
4 write(descriptorTuberia[1], buffer, length);
5 ...
6
7 //Proceso Hijo
8 close(descriptorTuberia[1]);
9 ...
10 read(descriptorTuberia[0], buffer, length);
11 ...
```

En el caso del proceso hijo, con la función `read()`, lo dejara bloqueado hasta que el proceso padre envíe su dato. Además, en caso de estar listo a leer los datos con la función `read()`, debemos de asegurar mediante un bucle de lectura completa de todos los datos enviados, ya que podemos obtener una cantidad parcial de la información enviada. Una tubería tiene en realidad dos descriptors de fichero: uno para el extremo de escritura y otro para el extremo de lectura. Como los descriptors de fichero de UNIX son simplemente enteros, un pipe o tubería no es más que un array de dos enteros:

```
1 int tuberia[2];
```

Para crear la tubería se emplea la función `pipe()`, que abre dos descriptors de fichero y almacena su valor en los dos enteros que contiene el array de descriptors de fichero. El primer descriptor de fichero es abierto como `O_RDONLY`, es decir, sólo puede ser empleado para lecturas. El segundo se abre como `O_WRONLY`, limitando su uso a la escritura. De esta manera se asegura que el pipe sea de un solo sentido: por un extremo se escribe y por el otro se lee, pero nunca al revés:

```
1 int tuberia[2];
2 pipe(tuberia);
```

Una vez creado un pipe, se podrán hacer lecturas y escrituras de manera normal, como si se tratase de cualquier fichero. Sin embargo, no tiene demasiado sentido usar un pipe para uso propio, sino que se suelen utilizar para intercambiar datos con otros procesos. Como ya sabemos, un proceso hijo hereda todos los descriptores de ficheros abiertos de su padre, por lo que la comunicación entre el proceso padre y el proceso hijo es bastante cómoda mediante una tubería. Para asegurar la unidireccionalidad de la tubería, es necesario que tanto padre como hijo cierren los respectivos descriptores de ficheros.

El proceso padre y su hijo comparten datos mediante una tubería:

```
1 #include <sys/types.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4 #include <stdio.h>
5
6 #define SIZE 512
7
8 int main( int argc, char **argv )
9 {
10     pid_t pid;
11     int p[2], readbytes;
12     char buffer[SIZE];
13
14     pipe( p );
15
16     if ( (pid=fork()) == 0 )
17     {
18         // hijo
19         close( p[1] ); /* cerramos el lado de escritura del pipe */
20
21         while( (readbytes=read( p[0], buffer, SIZE )) > 0 )
22             write( 1, buffer, readbytes );
23
24         close( p[0] );
25     }
26     else
27     {
28         // padre
29         close( p[0] ); /* cerramos el lado de lectura del pipe */
30
31         strcpy( buffer, "Esto llega a través de la tubería\n" );
32         write( p[1], buffer, strlen( buffer ) );
33
34         close( p[1] );
35
36     }
37     waitpid( pid, NULL, 0 );
38     exit( 0 );
39 }
40
```

Al ejecutar el código nos produce el siguiente error:

```
debian@debian: ~  
Archivo Editar Pestañas Ayuda  
debian@debian:~$ cd Documentos/Practica4  
bash: cd: Documentos/Practica4: No existe el fichero o el directorio  
debian@debian:~$ nano ejemplo1.c  
debian@debian:~$ gcc ejemplo1.c -o Ejemplo1  
ejemplo1.c: In function 'main':  
ejemplo1.c:31:9: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]  
31 |         strcpy( buffer, "Esto llega a traves de la tuberia\n" );  
    |         ^~~~~~  
ejemplo1.c:31:9: warning: incompatible implicit declaration of built-in function 'strcpy'  
ejemplo1.c:5:1: note: include '<string.h>' or provide a declaration of 'strcpy'  
4 | #include <stdio.h>  
+++ |+#include <string.h>  
5 |  
ejemplo1.c:32:30: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]  
32 |         write( p[1], buffer, strlen( buffer ) );  
    |                               ^~~~~~  
ejemplo1.c:32:30: warning: incompatible implicit declaration of built-in function 'strlen'  
ejemplo1.c:32:30: note: include '<string.h>' or provide a declaration of 'strlen'  
ejemplo1.c:38:5: warning: implicit declaration of function 'waitpid' [-Wimplicit-function-declaration]  
38 |         waitpid( pid, NULL, 0 );  
    |         ^~~~~~  
ejemplo1.c:39:5: warning: implicit declaration of function 'exit' [-Wimplicit-function-declaration]  
39 |         exit( 0 );  
    |         ^~~~~  
ejemplo1.c:39:5: warning: incompatible implicit declaration of built-in function 'exit'  
ejemplo1.c:5:1: note: include '<stdlib.h>' or provide a declaration of 'exit'  
4 | #include <stdio.h>  
+++ |+#include <stdlib.h>  
5 |  
debian@debian:~$
```

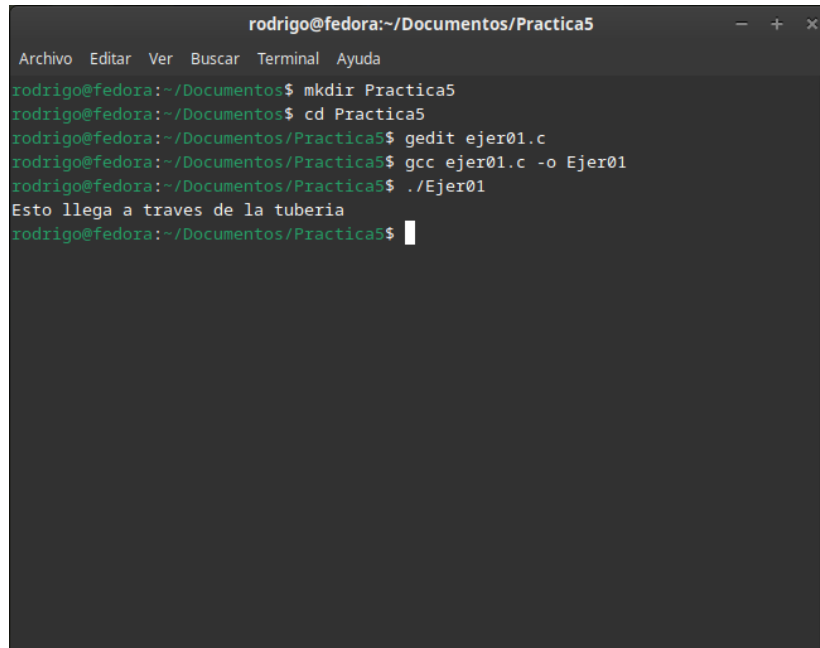
El código carece de las cabeceras necesarias para algunas funciones, como `string.h` y `stdlib`. Además, no se verificaban los posibles fallos en las llamadas a `pipe()` y `fork()`, lo que podría llevar a errores silenciosos. También faltaba un manejo adecuado del proceso hijo mediante `waitpid()`.

Código corregido:

```
1 #include <sys/types.h>  
2 #include <fcntl.h>  
3 #include <unistd.h>  
4 #include <stdio.h>  
5 #include <string.h> // para strcpy y strlen  
6 #include <stdlib.h> // para exit  
7 #include <sys/wait.h> // para waitpid  
8  
9 #define SIZE 512  
10  
11 int main(int argc, char **argv)  
12 {  
13     pid_t pid;  
14     int p[2], readbytes;  
15     char buffer[SIZE];  
16  
17     // Verificar si la creacin del pipe fue exitosa  
18     if (pipe(p) == -1) {  
19         perror("Error al crear el pipe");  
20         exit(1);  
21     }  
22  
23     pid = fork();  
24  
25     // Verificar si la bifurcacin del proceso fue exitosa  
26     if (pid == -1) {  
27         perror("Error en fork");  
28         exit(1);  
29     }  
30 }
```

```
30
31 if (pid == 0) // Proceso hijo
32 {
33     close(p[1]); // Cerramos el lado de escritura del pipe
34
35     // Leer datos del pipe y escribirlos en stdout
36     while ((readbytes = read(p[0], buffer, SIZE)) > 0)
37         write(1, buffer, readbytes);
38
39     close(p[0]); // Cerramos el lado de lectura del pipe
40 }
41 else // Proceso padre
42 {
43     close(p[0]); // Cerramos el lado de lectura del pipe
44
45     // Enviamos el mensaje al proceso hijo a travs del pipe
46     strcpy(buffer, "Esto llega a traves de la tuberia\n");
47     write(p[1], buffer, strlen(buffer));
48
49     close(p[1]); // Cerramos el lado de escritura del pipe
50
51     // Esperamos a que el proceso hijo termine
52     waitpid(pid, NULL, 0);
53 }
54
55 exit(0);
56 }
```

Corremos el código:



```
rodrigo@fedora:~/Documentos/Practica5
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos$ mkdir Practica5
rodrigo@fedora:~/Documentos$ cd Practica5
rodrigo@fedora:~/Documentos/Practica5$ gedit ejer01.c
rodrigo@fedora:~/Documentos/Practica5$ gcc ejer01.c -o Ejer01
rodrigo@fedora:~/Documentos/Practica5$ ./Ejer01
Esto llega a traves de la tuberia
rodrigo@fedora:~/Documentos/Practica5$
```

5. Ejercicios propuestos

5.1. Ejercicio 1

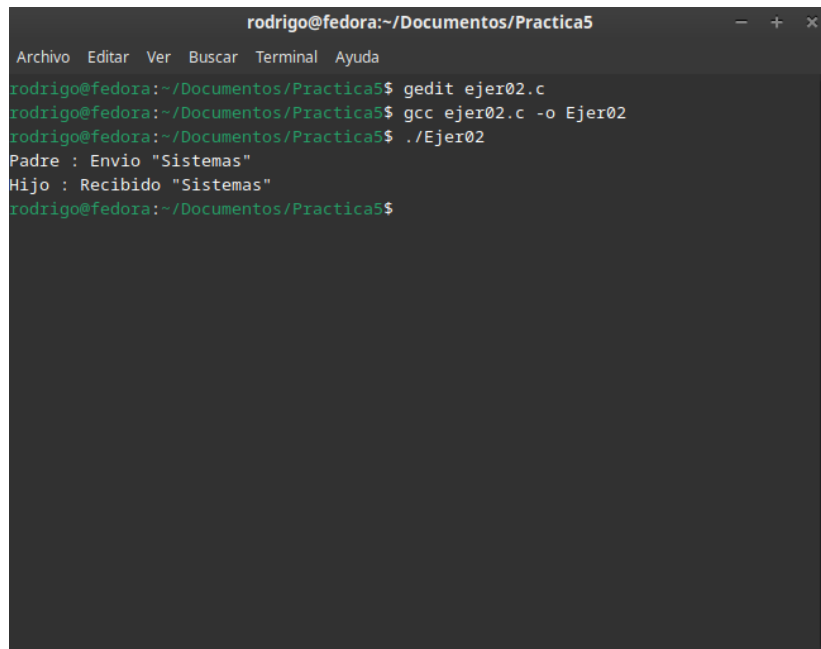
Analice y describa la actividad que realiza el siguiente código. Explique cómo sucede el proceso de envío de información del proceso padre al proceso hijo:

```
1  #include <sys/types.h>
2  #include <wait.h>
3  #include <unistd.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  int main() {
9      pid_t idProceso;
10     int estadoHijo;
11     int descriptorTuberia[2];
12     char buffer[100];
13
14     if (pipe(descriptorTuberia) == -1) {
15         perror("No se puede crear Tuberia");
16         exit(-1);
17     }
18
19     idProceso = fork();
20
21     if (idProceso == -1) {
22         perror("No se puede crear proceso");
23         exit(-1);
24     }
25
26     if (idProceso == 0) {
27         close(descriptorTuberia[1]);
28         read(descriptorTuberia[0], buffer, 9);
29         printf("Hijo : Recibido \"%s\"\n", buffer);
30         exit(0);
31     }
32
33     if (idProceso > 0) {
34         close(descriptorTuberia[0]);
35         printf("Padre : Envio \"Sistemas\"\n");
36         strcpy(buffer, "Sistemas");
37         write(descriptorTuberia[1], buffer, strlen(buffer) + 1);
38         wait(&estadoHijo);
39         exit(0);
40     }
41
42     return(1);
43 }
```

Análisis: el código ilustra la comunicación entre procesos utilizando tuberías. El proceso padre envía un mensaje al proceso hijo a través de un canal de comunicación, que el hijo lee y muestra en pantalla. Este ejemplo demuestra cómo se puede intercambiar información entre procesos mediante este método.

Actividades que realiza:

- El programa crea un pipe para que dos procesos (padre e hijo) puedan comunicarse.
- El proceso padre se encarga de enviar un mensaje al proceso hijo a través del pipe.
- El proceso hijo lee los datos enviados por el padre desde el pipe.
- Ambos procesos cierran los descriptores del pipe que no utilizan: el padre cierra el lado de lectura y el hijo cierra el lado de escritura.
- El proceso hijo imprime el mensaje recibido y luego termina su ejecución.
- El proceso padre espera la finalización del hijo usando la función `wait()` antes de finalizar su propia ejecución.



```
rodrigo@fedora:~/Documentos/Practica5
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica5$ gedit ejer02.c
rodrigo@fedora:~/Documentos/Practica5$ gcc ejer02.c -o Ejer02
rodrigo@fedora:~/Documentos/Practica5$ ./Ejer02
Padre : Envio "Sistemas"
Hijo : Recibido "Sistemas"
rodrigo@fedora:~/Documentos/Practica5$
```

Envío de información: el proceso padre escribe el mensaje (Sistemas) en el pipe a través del descriptor de escritura, y el proceso hijo lo recibe leyendo desde el descriptor de lectura del pipe, imprimiéndolo en pantalla tras la recepción.

5.2. Ejercicio 2

Analice y describa la actividad que realiza el siguiente código. Explique como sucede el proceso de envío de información del proceso padre al proceso hijo:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <errno.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 #define LEER 0
8 #define ESCRIBIR 1
```



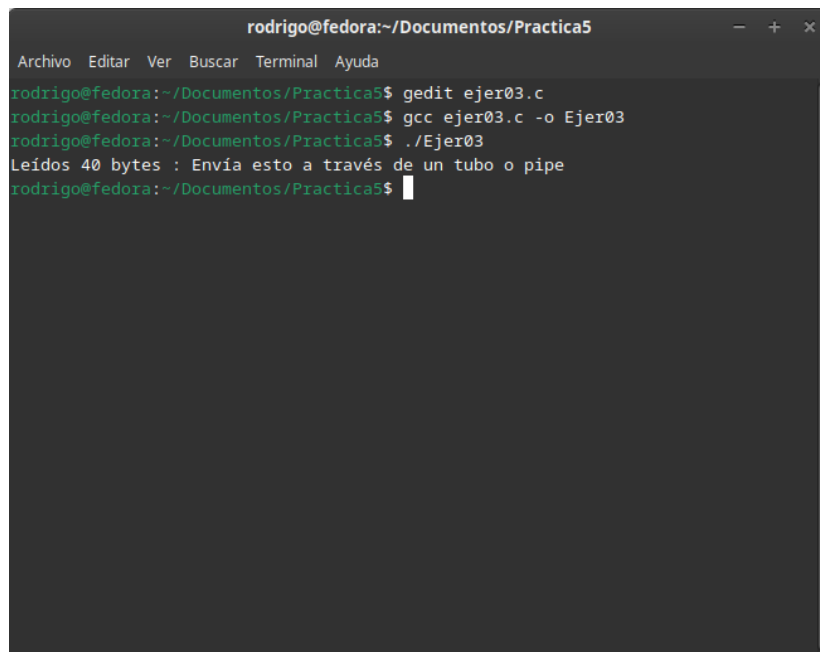
```
9
10 char *frase = "Envia esto a través de un tubo o pipe";
11 extern int errno;
12
13 int main() {
14     int fd[2], bytesLeidos;
15     char mensaje[100];
16     int control;
17
18     // se crea la tubería
19     control = pipe(fd);
20     if (control != 0) {
21         perror("pipe:");
22         exit(errno);
23     }
24
25     // se crea el proceso hijo
26     control = fork();
27     switch(control) {
28         case -1:
29             perror("fork:");
30             exit(errno);
31
32         case 0:
33             close(fd[LEER]);
34             write(fd[ESCRIBIR], frase, strlen(frase) + 1);
35             close(fd[ESCRIBIR]);
36             exit(0);
37
38         default:
39             close(fd[ESCRIBIR]);
40             bytesLeidos = read(fd[LEER], mensaje, 100);
41             printf("Leídos %d bytes : %s\n", bytesLeidos, mensaje);
42             close(fd[LEER]);
43     }
44
45     exit(0);
46 }
```

Análisis: el código implementa un mecanismo de comunicación entre un proceso padre y un proceso hijo utilizando una tubería (pipe). El padre envía una cadena de texto al hijo a través de este pipe, y el hijo recibe y muestra el mensaje.

Actividades que realiza:

- Se incluyen las librerías necesarias para el funcionamiento del programa, como `stdio.h`, `stdlib.h`, `unistd.h`, entre otras.
- Se define una cadena `frase` que contiene el mensaje: (Envía esto a través de un tubo o pipe).
- Se crean dos descriptores de archivo en un arreglo `fd[2]` para la tubería y se declara una variable `mensaje` para almacenar los datos leídos.
- Se crea la tubería llamando a `pipe(fd)`, que establece un canal de comunicación entre el proceso padre y el hijo.

- Se crea un proceso hijo utilizando `fork()`, que genera dos procesos: el padre y el hijo.
- Si el proceso es el hijo (cuando `fork()` devuelve 0), se cierra el descriptor de lectura de la tubería y se envía el mensaje a través del descriptor de escritura.
- Si el proceso es el padre, se cierra el descriptor de escritura de la tubería y se lee el mensaje enviado por el hijo a través del descriptor de lectura.
- El padre imprime el número de bytes leídos y el mensaje recibido del hijo.
- Finalmente, ambos procesos cierran sus respectivos descriptores de la tubería antes de finalizar su ejecución.



```
rodrigo@fedora:~/Documentos/Practica5
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica5$ gedit ejer03.c
rodrigo@fedora:~/Documentos/Practica5$ gcc ejer03.c -o Ejer03
rodrigo@fedora:~/Documentos/Practica5$ ./Ejer03
Leídos 40 bytes : Envía esto a través de un tubo o pipe
rodrigo@fedora:~/Documentos/Practica5$
```

Envío de información: el proceso padre escribe el mensaje en el descriptor de escritura de la tubería (`fd[ESCRIBIR]`), mientras que el proceso hijo lo recibe leyendo desde el descriptor de lectura (`fd[LEER]`), lo cual permite que el mensaje sea transferido y procesado entre ambos procesos.

5.3. Ejercicio 3

Elabore un programa propio que emplee la comunicación entre procesos (padre e hijo) utilizando pipes:

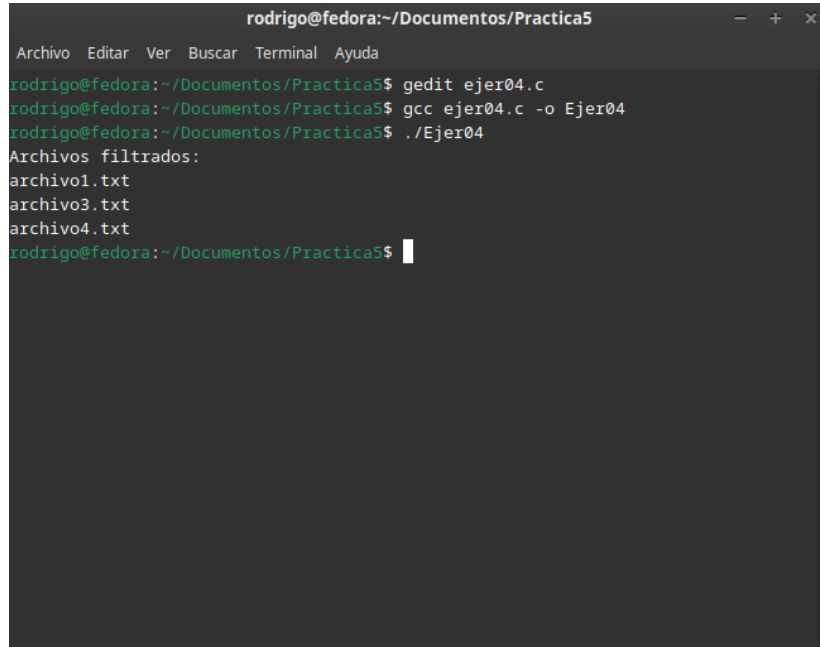
Propuesta:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/wait.h>
6
7 int main() {
8     int fd[2]; // Crear la tubera
9     pid_t pid;
10    char buffer[1024];
```

```
11
12 // Lista simulada de archivos (como si fuera la salida de 'ls')
13 char archivos[] = "archivo1.txt\narchivo2.doc\narchivo3.txt\narchivo4.txt\n";
14
15 // Crear la tubera
16 if (pipe(fd) == -1) {
17     perror("Error al crear la tubera");
18     exit(1);
19 }
20
21 pid = fork();
22
23 if (pid < 0) {
24     perror("Error al crear el proceso hijo");
25     exit(1);
26 }
27
28 if (pid == 0) {
29     // Proceso hijo - Filtra archivos .txt (simula 'grep ".txt"')
30     close(fd[1]); // Cierra el extremo de escritura (fd[1])
31
32     // Leer la lista de archivos desde la tubera
33     read(fd[0], buffer, sizeof(buffer));
34     close(fd[0]); // Cierra el extremo de lectura
35
36     // Filtrar los archivos que contienen ".txt"
37     printf("Archivos filtrados:\n");
38     char *linea = strtok(buffer, "\n");
39     while (linea != NULL) {
40         if (strstr(linea, ".txt") != NULL) {
41             printf("%s\n", linea); // Imprime solo los archivos .txt
42         }
43         linea = strtok(NULL, "\n");
44     }
45
46 } else {
47     // Proceso padre - Enva la lista de archivos (simula 'ls')
48     close(fd[0]); // Cierra el extremo de lectura (fd[0])
49
50     // Enva la lista de archivos al hijo
51     write(fd[1], archivos, strlen(archivos) + 1);
52     close(fd[1]); // Cierra el extremo de escritura
53
54     wait(NULL); // Asegurate de incluir el encabezado <sys/wait.h>
55 }
56
57 return 0;
58 }
```

El código utiliza una tubería (pipe) para establecer comunicación entre el proceso padre y el proceso hijo. El padre envía una lista de archivos simulando la salida de un comando ls, y el hijo lee esta lista a través de la tubería, filtrando y mostrando únicamente los archivos que tienen la extensión .txt. Este mecanismo ilustra cómo los procesos pueden colaborar para procesar información en un entorno multitarea utilizando IPC (comunicación entre procesos).

Código en ejecución:



```
rodrigo@fedora:~/Documentos/Practica5
Archivo Editar Ver Buscar Terminal Ayuda
rodrigo@fedora:~/Documentos/Practica5$ gedit ejer04.c
rodrigo@fedora:~/Documentos/Practica5$ gcc ejer04.c -o Ejer04
rodrigo@fedora:~/Documentos/Practica5$ ./Ejer04
Archivos filtrados:
archivo1.txt
archivo3.txt
archivo4.txt
rodrigo@fedora:~/Documentos/Practica5$
```

5.4. Ejercicio 4

Investigue cómo se pueden enviar datos de un proceso padre a un proceso hijo y viceversa:

Existen diversas maneras de enviar datos entre un proceso padre y un proceso hijo, así como entre procesos en general. A continuación, se describen algunos de los métodos más comunes de comunicación interprocesos (IPC):

5.4.1. Named Pipes (FIFO)

- **Descripción:** similar a los pipes, pero permite la comunicación entre procesos que no tienen relación directa (no necesitan ser padre e hijo).
- **Uso:** se crean utilizando la llamada al sistema `mkfifo`. Los procesos pueden escribir y leer desde el FIFO como si fuera un archivo.

5.4.2. Sockets

- **Descripción:** proporcionan una interfaz para comunicación entre procesos, incluso en diferentes máquinas.
- **Uso:** pueden ser utilizados para la comunicación bidireccional entre procesos. Los sockets pueden ser de tipo TCP o UDP dependiendo de si se necesita conexión o no.

5.4.3. Shared Memory (Memoria Compartida)

- **Descripción:** permite que varios procesos accedan a la misma región de memoria.
- **Uso:** se utiliza en situaciones donde se necesita compartir grandes cantidades de datos entre procesos de manera eficiente. Los procesos deben sincronizar el acceso a la memoria compartida para evitar condiciones de carrera.

5.4.4. Message Queues

- **Descripción:** permiten que los procesos envíen mensajes entre sí en una cola.
- **Uso:** se utilizan para la comunicación asincrónica entre procesos, permitiendo que los procesos envíen mensajes sin necesidad de que el receptor esté listo para leer en el momento de la escritura.

5.4.5. Signals (Señales)

- **Descripción:** son notificaciones que se envían entre procesos para indicar eventos.
- **Uso:** aunque no se utilizan para enviar datos directamente, se pueden emplear para notificar a un proceso que debe realizar una acción (como leer o escribir datos).