Práctica 4: Pipes y Fifos

75.59 - Técnicas de Programación Concurrente I

Ejercicios

1. Escribir un programa que utilice una tubería para permitir a un proceso padre, leer un mensaje de su proceso hijo.

2. Escribir un programa que ejecute dos programas, conectando la salida estándar del primero con la entrada estándar del segundo. Suponer que ninguno de los dos programas es invocado sin opciones y que los nombres de los programas están listados en la línea de comandos.

3. Estudiar cómo crear una tubería con nombre (named pipe) o fifo, utilizando el comando mknod.

4. Escribir un programa tal que:

   a) Utiliza un proceso lector y uno escritor

   b) Se ejecuta un único proceso lector que crea una tubería con nombre llamada "miPipe"

   c) Lee y muestra líneas de la tubería hasta que la misma es cerrada por todos los procesos escritores

   d) Uno o más procesos escritores se ejecutan, cada uno de los cuales abre la tubería "miPipe" y manda tres mensajes a la misma

   e) Si la tubería no existe cuando el proceso escritor intenta abrirla, el escritor vuelve a intentarlo cada segundo hasta que pueda abrirla

   f) Una vez enviados todos los mensajes, el escritor cierra el pipe y hace `exit()`

5. Estudiar la utilización de `lseek()` en un pipe.

6. ¿Es posible que dos procesos que se comunican vía un pipe sufran un abrazo mortal (deadlock)? Justificar la respuesta.

7. Estudiar si es posible utilizar un pipe como vía de comunicación bidireccional entre dos procesos, en algún ambiente UNIX o Linux. Hing: FreBSD, Solaris.

8. Se tiene el archivo de texto de nombre "archivoDatos", compuesto por las siguientes palabras, una por cada renglón y en el orden indicado: {ruso, jamaiquino, chino, inglés, árabe, francés, ruso, peruano, argentino, irlandés, español}. Escribir un programa que lea de este archivo y escriba en otro, de manera que el principal invoca a la siguiente función, para ordenar las palabras almacenadas en el archivo de entrada y explicar por qué no funciona correctamente:

```
1  void Malordenar ( void ) {
2          int pfd[2], fd;
3          ssize_t nread;
4          pid_t pid;
5          char buf[512];
6
7          pipe ( pfd );
8          pid = fork ();
9          if ( pid == 0 ) { /*hijo*/
10                 dup2(pfd[0], STDIN_FILENO);
11                 close ( pfd[0] );
12                 dup2 ( pfd[1],STDOUT_FILENO );
```

```
13                  close ( pfd[1] );
14                  execlp ( "sort","sort", (char *)NULL );
15          }
16
17          /*padre*/
18          fd = open ( "archivoDatos",O_RDONLY );
19          while ( true ) {
20                  nread = read(fd, buf, sizeof(buf));
21                  if ( nread == 0 )
22                          break;
23                  write ( pfd[1],buf,nread );
24          }
25          close ( fd );
26          close ( pfd[1] );
27          while ( true ) {
28                  nread= read(pfd[0],buf, sizeof(buf));
29                  if ( nread == 0 )
30                          break;
31                  write ( STDOUT_FILENO,buf,nread );
32          }
33          close ( pfd[0] );
34          waitpid ( pid,NULL,0 );
35          return;
36  }
```

Nota: a este código le falta, además, todo lo necesario para manejo de errores.

Apuntes

Pipes y Fifos

75.59 - Técnicas de Programación Concurrente I

Facultad de Ingeniería - Universidad de Buenos Aires

Resumen

Introducción

Mecanismos de comunicación unidireccional entre procesos

- Pipes: *unnamed pipes*
  - Los procesos que se comunican deben tener relación entre sí, por ejemplo, compartir el mismo padre o ser padre e hijo
- Fifos: *named pipes*
  - Los procesos que se comunican pueden no tener relación entre sí
  - Residen dentro del *filesystem* en forma de archivos

Creación, uso y destrucción de pipes (I)

Creación de un pipe

- Función *pipe()*

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

- Parámetros:
  - pipefd: vector de descriptores: 0 es el de lectura y 1 el de escritura (el proceso debe cerrar uno de los dos y utilizar el otro)
- Retorna:
  - 0 en caso de éxito
  - -1 en caso de error, seteando la variable externa *errno*

Creación, uso y destrucción de pipes (II)

Escritura en el pipe

- Función *write()*

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

Lectura del pipe

- Función *read()*

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

Destrucción del pipe

- El proceso debe cerrar el descriptor que dejó abierto

Reasignación de entrada y salida estándar (I)

Existen funciones que permiten redireccionar la entrada y salida estándar de un proceso

- Función *dup()*

```
int dup ( int oldfd );
```

  - Crea un file descriptor como copia de *oldfd* utilizando el menor descriptor libre
  - Retorna:
    - El file descriptor copiado en caso de éxito
    - -1 en caso de error, seteando la variable externa *errno*

- Función *dup2()*

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

  - Copia el file descriptor *oldfd* en el descriptor *newfd*, cerrando primero *newfd* si es necesario
  - Retorna:
    - El file descriptor copiado en caso de éxito
    - -1 en caso de error, seteando la variable externa *errno*

Reasignación de entrada y salida estándar (II)

- Reasignación de la entrada estándar

```
dup2 ( fd_obtenido,0 );
```

- Reasignación de la salida estándar

```
dup2 ( fd_obtenido,1 );
```


Creación, uso y destrucción de fifos (I)

Creación de un fifo

- Función *mknod()*

```
int mknod ( const char* pathname,mode_t mode,dev_t dev );
```

- Parámetros:
  - pathname: nombre del archivo que representa el fifo
  - mode: S_IFIFO |permisos
  - dev: se ignora
- Retorna:
  - 0 en caso de éxito
  - -1 en caso de error, seteando la variable externa *errno*

Desde la línea de comandos, ejecutar el comando `mknod`

Creación, uso y destrucción de fifos (II)

Escritura en el fifo

- Apertura del archivo con *open()*

```
int fd = open ( nombre,O_WRONLY );
```

- Escritura con *write()*

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

Lectura del fifo

- Apertura del archivo con *open()*

```
int fd = open ( nombre,O_RDONLY );
```

- Lectura con *read()*

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


Creación, uso y destrucción de fifos (III)

Destrucción del fifo

- Cierre del archivo abierto

```
int close ( int fd );
```

- Eliminación del archivo si no está referenciado por ningún proceso

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

Elimina la entrada de directorio para un archivo; el archivo ya no se puede acceder por su nombre.

Pipes y Fifos

75.59 - Técnicas de Programación Concurrente I

Facultad de Ingeniería - Universidad de Buenos Aires

Resumen

Introducción

Mecanismos de comunicación unidireccional entre procesos

- Pipes: *unnamed pipes*
  - Los procesos que se comunican deben tener relación entre sí, por ejemplo, compartir el mismo padre o ser padre e hijo
- Fifos: *named pipes*
  - Los procesos que se comunican pueden no tener relación entre sí
  - Residen dentro del *filesystem* en forma de archivos

Creación, uso y destrucción de pipes (I)

Creación de un pipe

- Función *pipe()*

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

- Parámetros:
  - pipefd: vector de descriptores: 0 es el de lectura y 1 el de escritura (el proceso debe cerrar uno de los dos y utilizar el otro)
- Retorna:
  - 0 en caso de éxito
  - -1 en caso de error, seteando la variable externa *errno*

Creación, uso y destrucción de pipes (II)

Escritura en el pipe

- Función *write()*

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

Lectura del pipe

- Función *read()*

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

Destrucción del pipe

- El proceso debe cerrar el descriptor que dejó abierto

Reasignación de entrada y salida estándar (I)

Existen funciones que permiten redireccionar la entrada y salida estándar de un proceso

- Función *dup()*

```
int dup ( int oldfd );
```

  - Crea un file descriptor como copia de *oldfd* utilizando el menor descriptor libre
  - Retorna:
    - El file descriptor copiado en caso de éxito
    - -1 en caso de error, seteando la variable externa *errno*

- Función *dup2()*

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

  - Copia el file descriptor *oldfd* en el descriptor *newfd*, cerrando primero *newfd* si es necesario
  - Retorna:
    - El file descriptor copiado en caso de éxito
    - -1 en caso de error, seteando la variable externa *errno*

Reasignación de entrada y salida estándar (II)

- Reasignación de la entrada estándar

```
dup2 ( fd_obtenido,0 );
```

- Reasignación de la salida estándar

```
dup2 ( fd_obtenido,1 );
```


Creación, uso y destrucción de fifos (I)

Creación de un fifo

- Función *mknod()*

```
int mknod ( const char* pathname,mode_t mode,dev_t dev );
```

- Parámetros:
  - pathname: nombre del archivo que representa el fifo
  - mode: S_IFIFO |permisos
  - dev: se ignora
- Retorna:
  - 0 en caso de éxito
  - -1 en caso de error, seteando la variable externa *errno*

Desde la línea de comandos, ejecutar el comando `mknod`

Creación, uso y destrucción de fifos (II)

Escritura en el fifo

- Apertura del archivo con *open()*

```
int fd = open ( nombre,O_WRONLY );
```

- Escritura con *write()*

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

Lectura del fifo

- Apertura del archivo con *open()*

```
int fd = open ( nombre,O_RDONLY );
```

- Lectura con *read()*

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


Creación, uso y destrucción de fifos (III)

Destrucción del fifo

- Cierre del archivo abierto

```
int close ( int fd );
```

- Eliminación del archivo si no está referenciado por ningún proceso

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

Elimina la entrada de directorio para un archivo; el archivo ya no se puede acceder por su nombre.

Observación

- El comportamiento default de pipes y fifos es *bloqueante*
- Se pueden utilizar en forma *no bloqueante* aunque **no se recomienda**
  - Pipes:
    fcntl `( descriptor,F_SETFL,O_NONBLOCK );`
  - Fifos: utilizar la constante O_NONBLOCK en *open()*

Comportamiento bloqueante y no bloqueante

Operación	Estado del pipe/fifo	Retorno	
		Bloqueante	No bloqueante
abrir para lectura	FIFO abierto para escritura	OK	OK
	FIFO no abierto para escritura	se bloquea hasta que el FIFO sea abierto para escritura	OK
abrir para escritura	FIFO abierto para lectura	OK	OK
	FIFO no abierto para lectura	se bloquea hasta que el FIFO sea abierto para lectura	Error `ENXIO`
leer pipe o fifo vacío	abierto para escritura	Se bloquea hasta que haya datos o sea cerrado para escritura	Error `EAGAIN`
	no abierto para escritura	Retorna 0 (EOF)	Retorna 0 (EOF)
escribir en el pipe o fifo	abierto para lectura	si bytes $\leq$ `PIPE_BUF` $\Rightarrow$ atómica	si bytes $\leq$ `PIPE_BUF` $\Rightarrow$ atómica si bytes $>$lugar_libre $\Rightarrow$ error `EAGAIN` si bytes $\leq$ lugar_libre $\Rightarrow$ OK
	no abierto para lectura	Señal `SIGPIPE`	Señal `SIGPIPE`

Bibliografía

- Manuales del sistema operativo
- *The Design of the Unix Operating System*, Maurice Bach
- *Unix Network Programming, Interprocess Communications*, W. Richard Stevens, segunda edición

Fuentes de los ejemplos

Listado 1: Ejemplo 1

```
#ifdef EJEMPLO_1

#include <string.h>
#include <iostream>
#include <stdlib.h>
#include <sys/wait.h>

#include "Pipes/Pipe.h"


int main () {

        static const int BUFFSIZE = 100;

        Pipe canal;
        int pid = fork ();

        if ( pid == 0 ) {

                // lector
                char buffer[BUFFSIZE];

                std::cout << "Lector: esperando para leer..." << std::endl;
                ssize_t bytesLeidos = canal.leer ( static_cast<void*>(buffer),BUFFSIZE );
                std::string mensaje = buffer;
                mensaje.resize ( bytesLeidos );

                std::cout << "Lector: lei el dato [" << mensaje << "] (" << bytesLeidos << "
                    bytes) del pipe" << std::endl;
                std::cout << "Lector: fin del proceso" << std::endl;

                canal.cerrar ();
                exit ( 0 );

        } else {

                // escritor
                std::string dato = "Hola mundo pipes!!";
                sleep ( 5 );
                canal.escribir ( static_cast<const void*>(dato.c_str()),dato.size() );

                std::cout << "Escritor: escribi el dato [" << dato << "] en el pipe" << std::
                    endl;
                std::cout << "Escritor: fin del proceso" << std::endl;

                // espero a que termine el hijo
                wait ( NULL );

                canal.cerrar ();
                exit ( 0 );
        }
}

#endif
```

Listado 2: Ejemplo 2

```
#ifdef EJEMPLO_2

#include <string.h>
#include <iostream>
#include <stdlib.h>
#include <sys/wait.h>

#include "Pipes/Pipe.h"


int main () {

        static const int BUFFSIZE = 100;

        Pipe canal;
        int pid = fork ();

        if ( pid == 0 ) {

```

```
20                  // lector
21                  char buffer[BUFFSIZE];
22                  canal.setearModo ( Pipe::LECTURA );
23
24                  std::cout << "Lector: esperando para leer . . ." << std::endl;
25
26                  // se redirige la entrada estandar
27                  dup2 ( canal.getFdLectura(),0 );
28                  // se lee el dato del pipe
29                  std::cin.get ( buffer,BUFFSIZE );
30
31                  std::cout << "Lector: lei el dato [" << buffer << "] del pipe" << std::endl;
32                  std::cout << "Lector: fin del proceso" << std::endl;
33
34                  canal.cerrar ();
35                  exit ( 0 );
36
37          } else {
38
39                  // escritor
40                  std::string dato = "Hola mundo pipes (segundo ejemplo)!!";
41                  canal.setearModo ( Pipe::ESCRITURA );
42
43                  sleep ( 5 );
44
45                  std::cout << "Escritor: escribo el dato [" << dato << "] en el pipe" <<std::
                        endl;
46
47                  // se redirige la salida estandar
48                  dup2 ( canal.getFdEscritura(),1 );
49                  // se escribe el dato en el pipe
50                  std::cout << dato << std::endl;
51
52                  // esperar a que el hijo termine
53                  wait ( NULL );
54
55                  canal.cerrar ();
56                  exit ( 0 );
57          }
58  }
59
60  #endif
```

Listado 3: Ejemplo 3

```
1   #ifdef EJEMPLO_3
2
3   #include <iostream>
4   #include <stdlib.h>
5   #include <sys/wait.h>
6
7   #include "Fifos/FifoLectura.h"
8   #include "Fifos/FifoEscritura.h"
9
10
11  int main () {
12
13          static const int BUFFSIZE = 100;
14          static const std::string ARCHIVO_FIFO = "/tmp/archivo_fifo";
15
16          pid_t pid = fork ();
17
18          if ( pid == 0 ) {
19                  FifoLectura canal ( ARCHIVO_FIFO );
20                  char buffer[BUFFSIZE];
21
22                  canal.abrir();
23                  std::cout << "[Lector] A punto de leer del fifo" << std::endl;
24                  ssize_t bytesLeidos = canal.leer(static_cast<void*>(buffer),BUFFSIZE);
25
26                  std::string mensaje = buffer;
27                  mensaje.resize ( bytesLeidos );
28                  std::cout << "[Lector] Lei el dato del fifo: " << mensaje << std::endl;
29
30                  canal.cerrar();
31                  std::cout << "[Lector] Fin del proceso" << std::endl;
32                  exit ( 0 );
33          } else {
34                  FifoEscritura canal ( ARCHIVO_FIFO );
35                  std::string mensaje = "Hola mundo fifo!!";
36
```

```
37                  canal.abrir();
38                  sleep ( 5 );
39                  canal.escribir ( static_cast<const void*>(mensaje.c_str()),mensaje.length() );
40                  std::cout << "[Escritor] Escribi el mensaje " << mensaje << " en el fifo" <<
                        std::endl;
41
42                  // esperar a que el hijo termine
43                  wait ( NULL );
44
45                  canal.cerrar ();
46                  canal.eliminar ();
47
48                  std::cout << "[Escritor] Fin del proceso" << std::endl;
49                  exit ( 0 );
50          }
51
52  }
53
54  #endif
```

Listado 4: Clase Pipe

```
1   #ifndef PIPE_H_
2   #define PIPE_H_
3
4   #include <unistd.h>
5   #include <fcntl.h>
6
7   class Pipe {
8
9   private:
10          int descriptores[2];
11          bool lectura;
12          bool escritura;
13
14  public:
15          static const int LECTURA = 0;
16          static const int ESCRITURA = 1;
17
18          Pipe();
19          ~Pipe();
20
21          void setearModo ( const int modo );
22
23          ssize_t escribir ( const void* dato,const int datoSize );
24          ssize_t leer ( void* buffer,const int buffSize );
25
26          int getFdLectura () const;
27          int getFdEscritura () const;
28
29          void cerrar ();
30  };
31
32  #endif /* PIPE_H_ */
```

Listado 5: Clase Pipe

```
1   #include "Pipe.h"
2
3   Pipe :: Pipe() : lectura(true), escritura(true) {
4           pipe ( this->descriptores );
5           /*fcntl ( this->descriptors[0],F_SETFL,O_NONBLOCK );
6           fcntl ( this->descriptors[1],F_SETFL,O_NONBLOCK );*/
7   }
8
9   Pipe::~Pipe() {
10  }
11
12  void Pipe :: setearModo ( const int modo ) {
13          if ( modo == LECTURA ) {
14                  close ( this->descriptores[1] );
15                  this->escritura = false;
16
17          } else if ( modo == ESCRITURA ) {
18                  close ( this->descriptores[0] );
19                  this->lectura = false;
20          }
21  }
22
23  ssize_t Pipe :: escribir ( const void* dato,int datoSize ) {
```

```
24          if ( this->lectura == true ) {
25                  close ( this->descriptores[0] );
26                  this->lectura = false;
27          }
28
29          return write ( this->descriptores[1],dato,datoSize );
30  }
31
32  ssize_t Pipe :: leer ( void* buffer,const int buffSize ) {
33          if ( this->escritura == true ) {
34                  close ( this->descriptores[1] );
35                  this->escritura = false;
36          }
37
38          return read ( this->descriptores[0],buffer,buffSize );
39  }
40
41  int Pipe :: getFdLectura () const {
42          if ( this->lectura == true )
43                  return this->descriptores[0];
44          else
45                  return -1;
46  }
47
48  int Pipe :: getFdEscritura () const {
49          if ( this->escritura == true )
50                  return this->descriptores[1];
51          else
52                  return -1;
53  }
54
55  void Pipe :: cerrar () {
56          if ( this->lectura == true ) {
57                  close ( this->descriptores[0] );
58                  this->lectura = false;
59          }
60
61          if ( this->escritura == true ) {
62                  close ( this->descriptores[1] );
63                  this->escritura = false;
64          }
65  }
```

Listado 6: Clase Fifo

```
1   #ifndef FIFO_H_
2   #define FIFO_H_
3
4   #include <string>
5   #include <sys/types.h>
6   #include <sys/stat.h>
7   #include <fcntl.h>
8   #include <unistd.h>
9
10  class Fifo {
11  public:
12          Fifo(const std::string nombre);
13          virtual ~Fifo();
14          virtual void abrir() = 0;
15          void cerrar();
16          void eliminar() const;
17
18  protected:
19          std::string nombre;
20          int fd;
21  };
22
23  #endif /* FIFO_H_ */
```

Listado 7: Clase Fifo

```
1   #include "Fifo.h"
2
3   Fifo::Fifo(const std::string nombre) : nombre(nombre), fd(-1) {
4           mknod ( static_cast<const char*>(nombre.c_str()),S_IFIFO|0666,0 );
5   }
6
7   Fifo::~Fifo() {
8   }
9
```

```
10 void Fifo::cerrar() {
11         close ( fd );
12         fd = -1;
13 }
14
15 void Fifo::eliminar() const {
16         unlink ( nombre.c_str() );
17 }
```

Listado 8: Clase FifoLectura

```
1  #ifndef FIFOLECTURA_H_
2  #define FIFOLECTURA_H_
3
4  #include "Fifo.h"
5
6  class FifoLectura : public Fifo {
7  public:
8          FifoLectura(const std::string nombre);
9          ~FifoLectura();
10
11         void abrir();
12         ssize_t leer(void* buffer,const ssize_t buffsize) const;
13 };
14
15 #endif /* FIFOLECTURA_H_ */
```

Listado 9: Clase FifoLectura

```
1  #include "FifoLectura.h"
2
3  FifoLectura::FifoLectura(const std::string nombre) : Fifo(nombre) {
4  }
5
6  FifoLectura::~FifoLectura() {
7  }
8
9  void FifoLectura::abrir() {
10         fd = open ( nombre.c_str(),O_RDONLY );
11 }
12
13 ssize_t FifoLectura::leer(void* buffer,const ssize_t buffsize) const {
14         return read ( fd,buffer,buffsize );
15 }
```

Listado 10: Clase FifoEscritura

```
1  #ifndef FIFOESCRITURA_H_
2  #define FIFOESCRITURA_H_
3
4  #include "Fifo.h"
5
6  class FifoEscritura : public Fifo {
7  public:
8          FifoEscritura(const std::string nombre);
9          ~FifoEscritura();
10
11         void abrir();
12         ssize_t escribir(const void* buffer,const ssize_t buffsize) const;
13 };
14
15 #endif /* FIFOESCRITURA_H_ */
```

Listado 11: Clase FifoEscritura

```
1  #include "FifoEscritura.h"
2
3  FifoEscritura::FifoEscritura(const std::string nombre) : Fifo(nombre) {
4  }
5
6  FifoEscritura::~FifoEscritura() {
7  }
8
9  void FifoEscritura::abrir() {
10         fd = open ( nombre.c_str(),O_WRONLY );
11 }
12
13 ssize_t FifoEscritura::escribir(const void* buffer,const ssize_t buffsize) const {
```

```
14          return write ( fd,buffer,buffsize );
15  }
```