TP V – Tuberías - Pipes

Objetivo: Desarrollar una serie de programas que permitan comprobar distintas funcionalidades vinculadas con la gestión de tuberías (pipes) y su aplicación.

Introducción

Las tuberías o pipes son una de las más viejas formas de comunicación entre procesos (IPC, inter process comunication) provistas por Unix, sin embargo, ellas tienen dos limitaciones:

1. Son half-duplex, es decir, el flujo de datos es unidireccional.
2. Pueden ser usados entre procesos que tengan un ancestro en común. Normalmente una tubería es creada por un proceso padre, quien llama a `fork()`, opcionalmente llama a `exec()` y luego la tubería es utilizada para comunicar padre con hijo.

1. La empresa X posee 10 sucursales, este proceso (`tp51`) le permite al usuario ingresar por teclado un código de sucursal (número comprendido entre 0 y 9) y muestra por pantalla el total de ventas de dicha sucursal (el dato de venta es proveniente de otro sistema que será simulado utilizando la función `random()`). Si el usuario ingresa un código de sucursal inválido, el programa devolverá -1 como total de ventas para dicha sucursal. La interacción con el programa podría ser la siguiente:

```
grchere@debian:~/gccwork/src/pipes$ ./tp51
1
suc[1]=180.428938
grchere@debian:~/gccwork/src/pipes$ ./tp51
99
suc[99]=-1.000000
grchere@debian:~/gccwork/src/pipes$ ./tp51
hola
suc[0]=180.428938
grchere@debian:~/gccwork/src/pipes$ ./tp51
-1
suc[-1]=-1.000000
grchere@debian:~/gccwork/src/pipes$
```

Sugerencia: para el ingreso por teclado se recomienda utilizar la función `read()`, sobre el descriptor de archivo STDIN_FILENO quien representa la entrada estandard.Compile el programa como tp51 (tp51.c).

2. La empresa X esta muy conforme con su programa interactivo que muestra los totales de ventas por sucursal, dicho programa (`tp51`) es muy viejo y fue desarrollado por el consultor Y. No se tienen los fuentes de dicho programa ni tampoco se sabe cómo accede a la información de los totales de ventas. Ahora la empresa X lo contrata a Ud. para que realice un programa que obtenga los totales de ventas de cada una de las sucursales y los muestre en pantalla, de forma tal, que la salida de este programa también pueda ser redirigida a un archivo. Un ejemplo de ejecución de este programa podría ser:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55
suc[0]=185.698899
suc[1]=48.488474
suc[2]=17.118268
suc[3]=201.864525
suc[4]=171.574494
```

```
suc[5]=33.204262
suc[6]=109.972127
suc[7]=80.015225
suc[8]=49.890014
suc[9]=20.500763
grchere@debian:~/gccwork/src/pipes$

Otra posibilidad es redirigiendo la salida:

grchere@debian:~/gccwork/src/pipes$ ./tp55 > salida.txt
grchere@debian:~/gccwork/src/pipes$ cat salida.txt
suc[0]=169.214392
suc[1]=31.285207
suc[2]=107.657237
suc[3]=185.225414
suc[4]=47.221871
suc[5]=124.386318
suc[6]=94.252531
suc[7]=63.131976
suc[8]=140.837584
suc[9]=110.428725
grchere@debian:~/gccwork/src/pipes$
```

Para lograr esto, se pueden utilizar tuberías o pipes. La idea sería tomar al programa `tp51` como una "caja negra" de la cual sólo sabemos sus entradas posibles y por ende, sus salidas posibles[1]. Podemos crear un nuevo programa, el cual hará un `fork()`, luego un `exec()` de `tp51` (quien se transformará en hijo de este nuevo programa) y creará una tubería entre el nuevo programa y su hijo, simulando una interacción que será "no humana", sino entre procesos: el padre "escribirá" la entrada del hijo y "leerá" de la salida del hijo. De esta forma, se podrán obtener los datos de cada una de las sucursales, juntarlos y por último imprimirlos todos juntos.
Iremos paso a paso, primero crearemos el programa `tp52` que hará un `fork()` y quedará esperando hasta la finalización del hijo (Ej: `waitpid(pid,&status,WUNTRACED);`). Introducimos algunos mensajes:

```
grchere@debian:~/gccwork/src/pipes$ ./tp52
main(): hago fork()!
hijo!
padre esperando hijo!
fin padre esperando hijo!
grchere@debian:~/gccwork/src/pipes$
```

Sugerencia: copie[2] código del TP II (proyecto shell primitivo), observe la parte del código en donde se hace un `fork()`, `exec()`, etc.
Compile el programa como `tp52` (`tp52.c`).

3. Copie `tp52.c` en `tp53.c`. Modifique `tp53.c`, ahora, luego de hacer el `fork()`, en la parte de código correspondiente al hijo, vamos a agregar una llamada a `execvp()` para reemplazar la imagen del proceso actual por la del proceso `tp51`, para ello podemos usar un código como este:

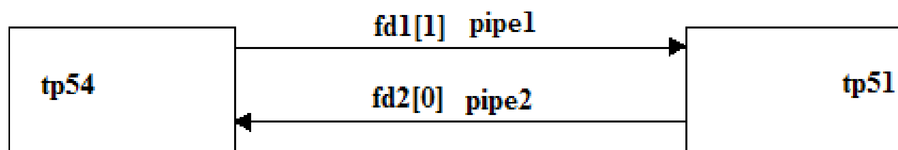
[1] Acorde con el Modelo Determinista de la programación.
[2] Aquí se debería utilizar el término "reutilizar", pero ello implicaría la creación y compartición de librerías. Una síntesis de ello esta tratado en la sección "Compilación y Catalogación de Librerías" del apunte " Trabajando con el Compilador GNU C/C++ GCC" que se encuentra en /Aputes/ProgramacionI-2008-gcc.doc del cd-rom de programación de esta asignatura.

```
char *prog[] = {"./tp51","./tp51",NULL};
if (execvp(prog[0], prog) == -1) {
   printf("main():Error en execvp()\n");
}
```

Ahora el proceso hijo se transformará en `tp51`, provocando el ingreso por teclado y la impresión del total de ventas de la sucursal ingresada por teclado:

```
grchere@debian:~/gccwork/src/pipes$ ./tp53
main(): hago fork()!
hijo!
padre esperando hijo!
2
suc[2]=180.428938
fin padre esperando hijo!
grchere@debian:~/gccwork/src/pipes$
```

4. Copie `tp53.c` en `tp54.c`. Hemos logrado que `tp53` ejecute a `tp51`, no obstante, el problema es que todo el proceso continúa siendo interactivo, ahora necesitamos establecer una comunicación entre procesos y sin intervención "humana". Podemos establecer un pipe de comunicación entre `tp54` y `tp51`, adicionalmente, el proceso hijo deberá conectar un extremo del pipe con su stdin (entrada estandard, descriptor `STDIN_FILENO`) y otro extremo del pipe con su stdout (salida estandard, descriptor `STDOUT_FILENO`).
La descripción anterior nos indica que `tp54` y `tp51` actúan como *coprocesos*, en donde uno de los procesos corre en background y conecta su stdin y stdout a otro proceso que lo leerá y escribirá según corresponda.
Trabajando con pipes unidireccionales (half-duplex) el esquema sería:



La función `pipe()` nos permite crear un pipe representado por un arreglo de enteros que contiene dos descriptores de archivos[3]: uno sobre el cual podemos grabar (posición 1 del arreglo) y otro sobre el cual podemos leer (posición 0 del arreglo). Necesitamos dos pipes para comunicar estos procesos. La función `pipe()` devuelve un valor < 0 cuando hay un error:

```
   int fd1[2],fd2[2];
   // crear pipes
   if ( pipe(fd1) < 0 || pipe(fd2) < 0 ) {
      fprintf(stderr,"main():Error creando pipes!\n");
      exit(1);
   }
```

[3] Los descriptores de archivos son datos de tipo entero, por lo tanto, nada prohibe que podamos almacenarlos en un arreglo de enteros.

Hay dos extremos de los pipes que no serán utilizados por el padre: `fd1[0]` (`tp54` no pretende leer del stdin de `tp51`) y `fd2[1]` (`tp54` no pretende grabar en el stdout de `tp51`). Por lo tanto estos extremos pueden ser cerrados por el padre:

```
   close(fd1[0]);
   close(fd2[1]);
```

Idem anterior, hay dos extremos que no serán utilizados por el hijo: `fd1[1]` (`tp51` no pretende escribir en su stdin!) y `fd2[0]` (`tp51` no pretende leer de su stdout!). Por lo tanto estos extremos pueden ser cerrados por el hijo:

```
   close(fd1[1]);
   close(fd2[0]);
```

El hijo deberá conectar su stdin con el pipe `fd1[0]` y su stdout con el pipe `fd2[1]`, para ello podemos usar la función `dup2()`, ya utilizada en el proyecto del shell:

```
...
   if (fd1[0] != STDIN_FILENO) {
      if ( dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
fprintf(stderr,"ejecutoHijo():dup2 error en stdin\n");
      close(fd1[0]);
   }
   if (fd2[1] != STDOUT_FILENO) {
      if ( dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
fprintf(stderr,"ejecutoHijo():dup2 error en stdout\n");
      close(fd2[1]);
   }
...
```

Una vez creados los pipes y conectados de la forma apropiada, sólo queda leer y escribir de ellos, tarea a ser realizada por el proceso padre. Para leer y escribir se pueden utilizar las funciones `read()` y `write()` puesto que el contenido de los arreglos `fd1` y `fd2` son descriptores de archivos[4]. Por ejemplo, para pedir y obtener el total de ventas de la sucursal 2:

```
...
   strcpy(buffer,"2\n"); // solicito sucursal 2
   int l = strlen(buffer);
   int n = write(fd1[1],buffer,l);
   if ( n != l ) printf("pidoVentas:grabe %d y esperaba grabar %d en
suc=2\n",n,l);
   n = read(fd2[0],buffer,MAX_BUFFER);
   buffer[n]='\0';  // no olvidar finalizar string con \0
   printf("main():lei de prog. hijo: [%s]",buffer);
...
```

Luego de enviar y recibir los datos, el proceso padre continua con la espera de la finalización del hijo, compilamos y probamos `tp4.c`, podríamos obtener la siguiente salida:

```
grchere@debian:~/gccwork/src/pipes$ ./tp54
```

[4] Las funciones open(), read(), write(), seek(), close() trabajan con descriptores de archivos (tipo de dato: int); mientras que las funciones fopen(), fread(), fwrite(), fseek(), fclose() trabajan con streams (tipo de dato: FILE *).

```
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[1]=180.428938
]
padre esperando hijo!
fin padre esperando hijo!
grchere@debian:~/gccwork/src/pipes$
```

Puede observarse que no hay una interacción "humana" sino entre procesos. En este caso, `tp54` solicita el código de la sucursal `1` (envía `"1\n"`) y recibe como respuesta `"suc[1]=180.428938\n"`; exactamente los caracteres que ingresa el operador como entrada de `tp51` y los caracteres que `tp51` emite como salida.

5. Copie `tp54.c` en `tp55.c`. Ahora debemos agregar un loop de 0 a 9 para solicitar las ventas de las 10 sucursales. El diseño general del programa es el siguiente:

inicio loop sucursal	
creo pipes	
hago fork()	
proceso padre	proceso hijo
cierro pipes innecesarios	cierro pipes innecesarios
envio codigo sucursal	conecto stdin
recibo total de ventas	conecto stdout
imprimo total de ventas	exec de tp51
espero finalización de hijo	
fin loop sucursal	

Una salida posible del programa `tp55`:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[0]=63.149707
]padre esperando hijo!
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[1]=31.934346
]padre esperando hijo!
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[2]=109.752318
]padre esperando hijo!
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[3]=185.506887
]padre esperando hijo!
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[4]=155.639549
]padre esperando hijo!
```

```
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[5]=125.171518
]padre esperando hijo!
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[6]=94.452528
]padre esperando hijo!
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[7]=173.261329
]padre esperando hijo!
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[8]=142.342423
]padre esperando hijo!
fin padre esperando hijo!
main(): hago fork()!
hijo!
main():lei de prog. hijo: [suc[9]=112.486605
]padre esperando hijo!
fin padre esperando hijo!
grchere@debian:~/gccwork/src/pipes$
```

Ahora modificamos `tp55`, quitamos los mensajes que estan de mas:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55
suc[0]=185.698899
suc[1]=48.488474
suc[2]=17.118268
suc[3]=201.864525
suc[4]=171.574494
suc[5]=33.204262
suc[6]=109.972127
suc[7]=80.015225
suc[8]=49.890014
suc[9]=20.500763
grchere@debian:~/gccwork/src/pipes$
```

Redirigiendo la salida:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55 > salida.txt
grchere@debian:~/gccwork/src/pipes$ cat salida.txt
suc[0]=169.214392
suc[1]=31.285207
suc[2]=107.657237
suc[3]=185.225414
suc[4]=47.221871
suc[5]=124.386318
suc[6]=94.252531
suc[7]=63.131976
suc[8]=140.837584
suc[9]=110.428725
grchere@debian:~/gccwork/src/pipes$
```

Volvemos a modificar `tp55`, debemos capturar la señal `SIGPIPE` la cual será lanzada cuando grabamos en un pipe en donde su lector ya ha terminado. Implementamos la captura, en caso de producirse la señal `SIGPIPE`, enviamos mensaje de error y retornamos `2` al shell, comprobamos nuevamente el funcionamiento de `tp55`:

```
grchere@debian:~/gccwork/src/pipes$ ./tp55
suc[0]=189.557790
suc[1]=158.801467
suc[2]=127.814084
suc[3]=97.586370
suc[4]=65.977004
suc[5]=144.760503
suc[6]=6.487983
suc[7]=83.616891
suc[8]=159.872828
suc[9]=21.453911
grchere@debian:~/gccwork/src/pipes$
```

En cuanto al uso de pipes

Lo visto en esta práctica es sumamente elemental en cuanto al tema pipes, también existen implementaciones full-duplex de pipes, la posibilidad de duplicar streams usando pipes, el uso de FIFO's o "named pipes" que también pueden ser un medio muy propicio para la programación client-server. En esta práctica se optó por un camino algo más largo pero más detallado para la explicación y comprensión de los pipes, pero cabe aclarar que existen las funciones `popen()` y `pclose()` que, si bien tienen algunas limitaciones, son muy prácticas y fáciles de usar para la implementación de coprocesos.

Bibliografía

- Stevens, Richard, "Advanced Programming in the UNIX Environment", Addison-Wesley Professional Computing Series, 1993, ISBN 0-201-56317-7
- LooseSandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, Ulrich Drepper, "The GNU C Library Reference Manual", Free Software Foundation, 2007, Boston, USA.

Espero que esta práctica haya sido de vuestro agrado y le permita comprender mejor los mecanismos básicos de manejo de pipes y coprocesos como otra forma de comunicacion entre procesos provista por el sistema operativo.