Contenido

8. Comunicación mediante tuberías

8.1 Comunicación entre procesos

La comunicación entre procesos va a habilitar mecanismos para que los procesos puedan intercambiarse datos y sincronizarse. Hasta ahora hemos visto dos formas bastantes elementales para que dos procesos se comuniquen: el envío de señales para la sincronización y el uso de archivos ordinarios. Las señales no deben considerarse parte de la forma habitual de comunicar dos procesos y su uso debe restringirse a la comunicación de eventos o situaciones excepcionales. Los archivos ordinarios tampoco son la forma más eficiente de comunicarse, ya que se ve involucrado el acceso a disco que suele ser 3 o 4 órdenes de magnitud más lento que el acceso a memoria.

Los mecanismos que vamos a estudiar a continuación pretenden dar soluciones más eficientes a las necesidades de comunicación empleando como canal de transmisión la memoria principal. Esto va a suponer una mayor velocidad de transferencia de datos.

A la hora de comunicar dos procesos, vamos a considerar dos situaciones claramente diferentes:

- Que los procesos se están ejecutando bajo el control de una misma computadora.
- Que los procesos se están ejecutando en computadoras separadas.

La primera situación no va a aportar especial dificultad, ya que lo que hemos estado realizando hasta ahora va en esa línea. Así, para comunicar dos o más procesos a nivel local, vamos a estudiar un mecanismo clásico como son las tuberías, y después vamos a pasar a ver las facilidades de IPC

de UNIX. Estas facilidades están diseñadas con una misma filosofía y engloban tres mecanismos de comunicación: semáforos, memoria compartida y filas de mensajes.

El segundo escenario es más complejo, porque se ven involucradas las redes de computadoras y la comunicación entre ellas. Hoy en día este tema tiene mucho interés, porque el proceso distribuido es una tecnología con gran uso. La idea de tener varias computadoras interconectadas y que cualquier usuario pueda manejarlas como si se tratasen de una sola, pero con una potencia cercana a la suma de todas ellas, no sólo es interesante como ejercicio teórico, sino que ya tiene implementación real.

Como introducción al tema de la comunicación entre computadoras remotas, vamos a exponer la interfaz que suministra el UNIX de Berkeley para comunicar procesos. Esta interfaz se compone de una serie de llamadas que manejan un nuevo tipo de archivo conocido como conector (socket) y que va a actuar como canal de comunicación entre procesos. Aunque un conector es tratado sintácticamente como un archivo; semánticamente, no lo es. Esto significa que no vamos a tener los problemas de velocidad inherentes al acceso a disco.

Las tuberías son una de las primeras formas de comunicación implantadas en UNIX y muchos sistemas se ofrecen hoy día con esta facilidad.

Una tubería se puede considerar como un canal de comunicación entre dos procesos, y las hay de dos tipos: tuberías con nombre (`fifos`) y tuberías sin nombre.

8.2 Tuberías sin Nombre

Las tuberías sin nombre se crean con la llamada pipe y sólo el proceso que hace la llamada y sus descendientes pueden utilizarla. `pipe` tiene la siguiente declaración:

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

Si la llamada funciona correctamente, devolverá el valor 0 y creará una tubería sin nombre; en caso contrario, devolverá -1 y en `errno` estará el código del error producido.

La tubería creada la vamos a poder manejar a través del arreglo `filedes`. Los dos elementos de `filedes` se comportan como dos descriptores de archivo y los vamos a usar para escribir y leer de la tubería. Al escribir en `filedes[0]` vamos a extraer los datos de ella. Naturalmente, `filedes[1]` se comporta como un archivo de solo escritura y `filedes[0]` como un archivo de sólo lectura.

Como el núcleo trata la tubería igual que a un archivo del sistema, al crearla debe asignarle un nodo-i. También le asigna un par de descriptores de archivo (`filedes[0]` y `filedes[1]`) y reserva las correspondientes entradas en la tabla de archivos del sistema y en la tabla de descriptores del proceso.

Todo eso facilita el manejo de la tubería, ya que al recibir el mismo tratamiento que un archivo, vamos a poder escribir y leer de ella con las llamadas `write` y `read`, llamadas que empleamos para los archivos ordinarios, directorios, archivos especiales y también tuberías.

Los descriptores de archivo se heredan de padres a hijos tras la llamada a `fork` o a `exec`. Así, para que se comuniquen padre e hijo mediante una tubería, la abriremos en el padre y tanto padre como hijo podrán compartirla.

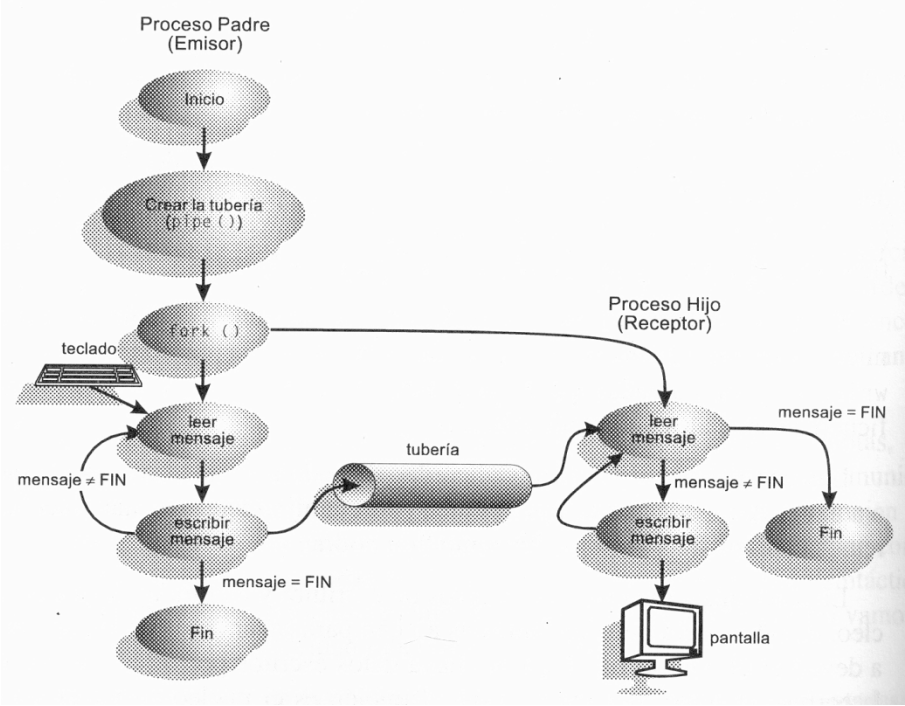
La sincronización entre los accesos de escritura y lectura la lleva a cabo el kernel, de tal manera que, las llamadas a `read` para sacar datos de la tubería, no van a devolver el control hasta que no haya datos escritos por otro proceso mediante la correspondiente llamada a `write`. También es el kernel el que se encarga de gestionar la tubería para dotarla de un mecanismo de acceso tipo `FIFO` (`first-in, first-out`) y así, el proceso receptor sacará los datos en el mismo orden en que los escriba el proceso emisor.

Los datos escritos en la tubería se gestionan en el buffer caché sin que llegue al disco, por lo que, al producirse la transferencia a través de memoria, las tuberías constituyen un mecanismo de comunicación mucho más rápido que el uso de archivo ordinarios. El tamaño de una tubería; es decir, el bloque de datos más grande que podemos escribir en ella depende del sistema, pero se garantiza que no va a ser inferior a 4,096 byte.

Cuando la tubería está llena, las llamadas a `write` quedan desbloqueadas hasta que no se saquen suficientes datos de la tubería como para escribir el bloque deseado.

Como ejemplo de aplicación, vamos a ver la forma de enviar datos desde un proceso emisor a un proceso receptor a través de una tubería sin nombre (ver la siguiente figura). Este ejemplo es el mismo que vimos para ilustrar la sincronización mediante señales de dos procesos. Ahora

veremos que sólo tenemos que preocuparnos por los aspectos de envío y recepción, ya que la sincronización es algo que resuelve el kernel.




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

#define MAX 256

void limpia(char *cadena) {
	char *p = strchr(cadena, '\n');
	if (p) {
		*p = '\0';
	}
}

int main(int argc, char * argv[]) {
	int tuberia[2];
	int pid;
	char mensaje[MAX];

	if (pipe(tuberia) < 0) {
		perror("pipe");
		return -1;
	}

	if ((pid = fork()) < 0) {
		perror("fork");
		return -1;
	} else if (pid == 0) {
		while (read(tuberia[0], mensaje, MAX) > 0 &&
			strcmp(mensaje, "FIN") != 0) {
			limpia(mensaje);
```

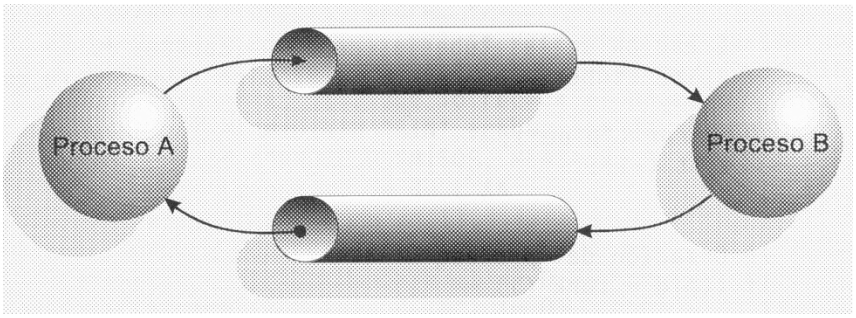
```
                fprintf(stdout, "PROCESOR RECEPTOR, MENSAJE: %s\n", mensaje);
                strcpy(mensaje,"");
        }
        close(tuberia[0]);
        close(tuberia[1]);
        return 0;
    } else {
        do {
            fprintf(stdout, "PROCESO EMISOR, MENSAJE: ");
            fgets(mensaje, MAX, stdin);
            limpia(mensaje);
            write(tuberia[1], mensaje, strlen(mensaje) + 1);
        } while (strcmp(mensaje, "FIN") != 0);
        close(tuberia[0]);
        close(tuberia[1]);
        return 0;
    }
}
```

8.3 Comunicación Bidireccional

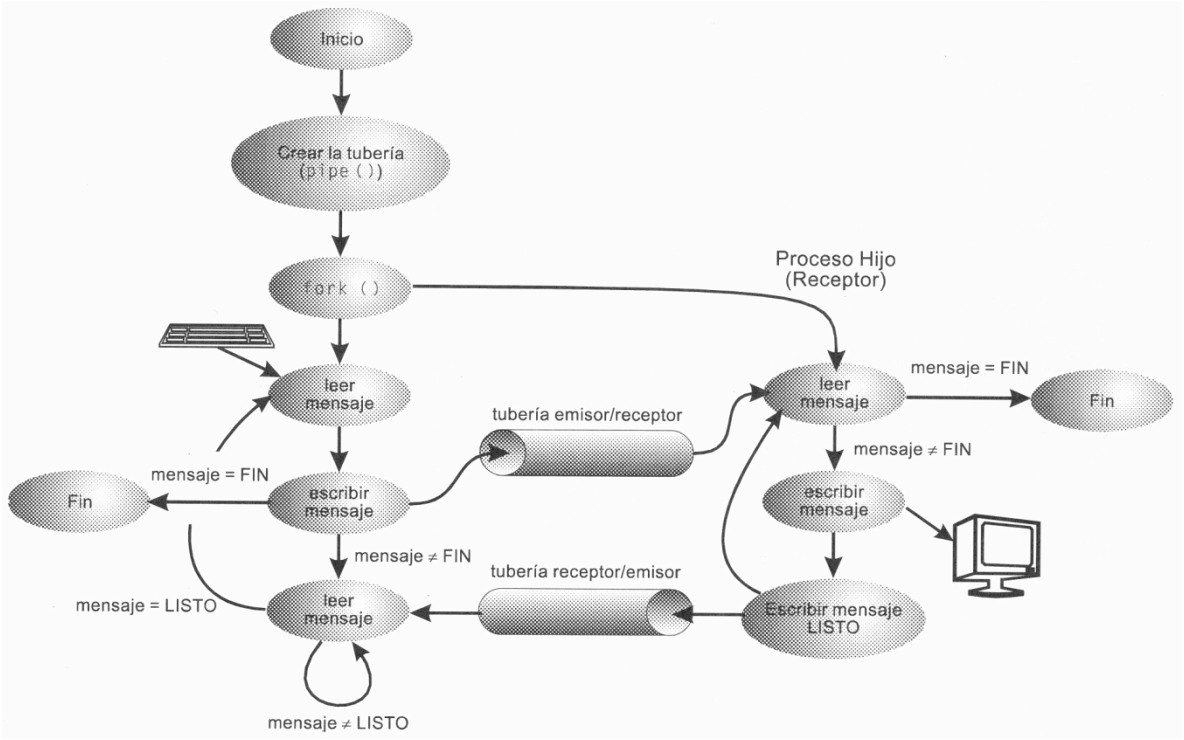
Uno de los problemas que presenta el programa del apartado anterior es la falta de comunicación entre el proceso hijo y el padre, de tal forma que el proceso emisor puede pedirnos que introduzcamos más mensajes antes de que el proceso receptor haya presentado el mensaje que acabamos de enviarle. Para solucionar este problema tenemos que implementar algún tipo de protocolo entre los procesos.

Para implementar esta comunicación, necesitamos otra tubería que sirva de canal entre el proceso receptor y el emisor. Podríamos sentirnos tentados a aprovechar una sola tubería como canal bidireccional, pero esto plantea problemas de sincronismo y tendríamos que ayudarnos de señales o semáforos para controlar el acceso a la tubería. En efecto, si un proceso escribe en la tubería un mensaje para otro proceso y se pone a leer de ella la respuesta que le envía éste, puede darse el caso de que lea el mensaje que él mismo envió.

Lo mejor es valernos de dos tuberías (como se muestra en la siguiente figura), una lleva los mensajes que van del proceso A al proceso B, y la otra lleva los mensajes en sentido contrario.



A continuación, podemos ver el código correspondiente al mismo ejemplo del tema anterior, pero con un protocolo entre los procesos emisor y receptor, de tal forma que el proceso emisor no va a efectuar el envío de otro mensaje hasta que el receptor haya procesado totalmente el último mensaje que recibió (ver la siguiente figura).




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

#define MAX 256

void limpia(char *cadena) {
	char *p = strchr(cadena, '\n');
	if (p) {
		*p = '\0';
	}
}

int main(int argc, char *argv[]) {
	int tuberia_em_re[2];
	int tuberia_re_em[2];
	int pid;
	char mensaje[MAX];

	if (pipe(tuberia_em_re) < 0 || pipe(tuberia_re_em) < 0) {
		perror("pipe");
		return -1;
	}

	if ((pid = fork()) < 0) {
		perror("fork");
		return -1;
	} else if (pid == 0) {
		while(read(tuberia_em_re[0], mensaje, MAX) > 0 &&
			strcmp(mensaje, "FIN") != 0) {
			fprintf(stdout, "PROCESO RECEPTOR, MENSAJE: %s\n", mensaje);

			strcpy(mensaje, "LISTO");
```

```
                write(tuberia_re_em[1], mensaje, strlen(mensaje) + 1);
        }
        close (tuberia_em_re[0]);
        close (tuberia_em_re[1]);
        close (tuberia_re_em[0]);
        close (tuberia_re_em[1]);
        return 0;
    } else {
        do {
            fprintf(stdout, "PROCESO EMISOR, MENSAJE: ");
            fgets(mensaje, MAX, stdin);
            limpia(mensaje);
            write(tuberia_em_re[1], mensaje, strlen(mensaje) + 1);
            if (strcmp(mensaje, "FIN") == 0) {
                break;
            }
            do {
                strcpy(mensaje, "");
                read(tuberia_re_em[0], mensaje, MAX);
            } while (strcmp(mensaje, "LISTO") != 0);
        } while (1);
        close (tuberia_em_re[0]);
        close (tuberia_em_re[1]);
        close (tuberia_re_em[0]);
        close (tuberia_re_em[1]);
        return 0;
    }
}
```

8.4 Tuberías en los intérpretes de líneas de comando

Hoy en día, casi todos los intérpretes de líneas de comando ofrecen la posibilidad de redirigir tanto la entrada como la salida de la ejecución de un programa.

Normalmente, los programas utilizan los archivos estándar para efectuar la entrada/salida. Estos programas pueden servirnos para trabajar con otros archivos sin necesidad de modificar su código, pero para ello el intérprete de línea de comandos que nos comunica con el sistema operativo debe contemplar la redirección. La redirección hacia otros archivos se le indica al intérprete mediante los caracteres "<" y ">" (redirección de entrada y salida, respectivamente).

Otra forma de redirigir es mediante las tuberías. Con ellas, lo que conseguimos es que la salida de un programa se convierta en entrada para otro. Esto es importante a la hora de aprovechar programas estándar, que realizan funciones sencillas, para construir otros que realizan funciones más complejas.

Como ejemplo, vamos a ver cómo podemos sacar a pantalla el listado de un directorio en orden alfabético inverso. Sabemos que la orden `ls` muestra por la salida estándar (pantalla) el contenido de un directorio. Sabemos también que la orden `sort -r` lee líneas de la entrada estándar y las muestra por la salida estándar en orden alfabético inverso. Mediante una tubería vamos a lograr que la salida de `ls` se redirija hacia la entrada de `sort` y el resultado será un listado del directorio en el orden deseado. La orden por ejecutar es:

```
$ ls | sort -r
```

Tanto `ls` como `sort` son dos programas que se van a ejecutar en paralelo o concurrentemente y que se van a comunicar a través de una tubería sin nombre.

Para ilustrar cómo se puede llevar a cabo esto, vamos a ver un programa que codifica la parte de un intérprete de línea de comando encargado de crear tuberías y comunicar procesos a través de ellas (ver la siguiente figura). Este programa se llama `tuberia` y la forma de invocarlo es:

```
$tuberia programa1 programa2
```

El resultado de la ejecución de la línea anterior es el mismo que el de la línea:

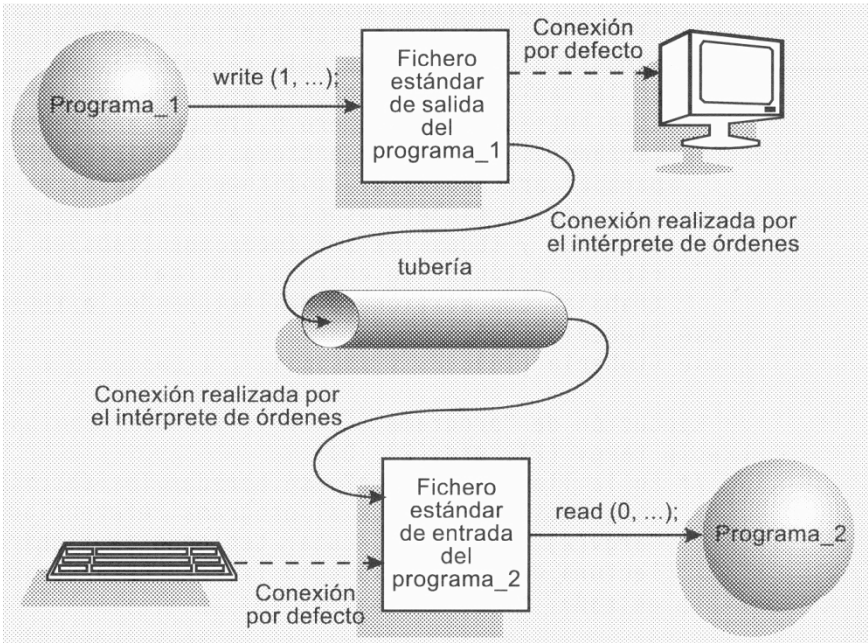
```
$ programa1 | programa2
```

El código del programa tubería es el siguiente:

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

int main(int argc, char *argv[]) {
	int tuberia[2];
	int pid;
	if (argc < 3) {
		fprintf(stderr, "forma de uso: %s programa1 programa2\n", argv[0]);
		return -1;
	}

	if (pipe(tuberia) < 0) {
		perror(argv[0]);
		return -1;
	}

	if ((pid = fork()) < 0) {
		perror(argv[0]);
		return -1;
	} else if (pid == 0) {
		close(0); dup(tuberia[0]);
		close(tuberia[0]);
		close(tuberia[1]);
		execlp(argv[2], argv[2], 0);
	} else {
		close(1); dup(tuberia[1]);
		close(tuberia[0]);
		close(tuberia[1]);
		execlp(argv[1], argv[1], 0);
	}
	return 0;
}
```

Para que un programa pueda formar parte de una tubería tal y como las manejan los intérpretes de línea de comando, ha de cumplir dos requisitos:

1. La entrada de datos al programa se debe realizar a través del archivo estándar de entrada (descriptor de archivo número 0).
2. La salida de datos se debe realizar a través del archivo estándar de salida (descriptor de archivo número 1).



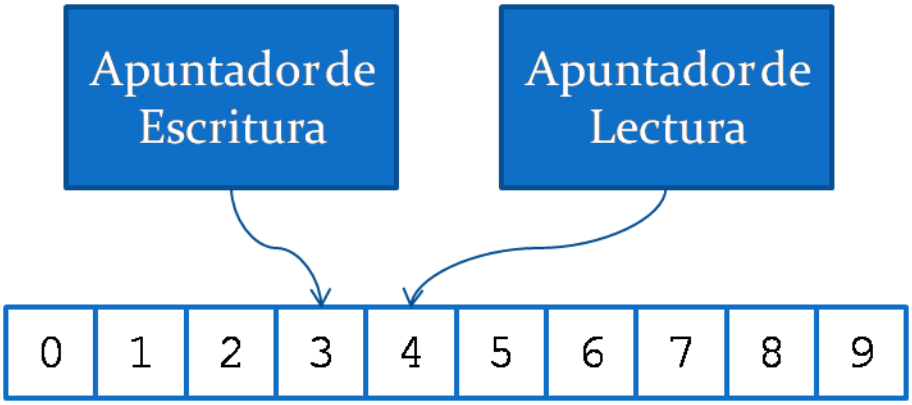
8.5 Tuberías con Nombre

Por medio de las tuberías sin nombre podemos comunicar procesos relacionados entre sí. El proceso que crea la tubería y sus descendientes tiene acceso a la misma. Para los procesos que no guardan ninguna relación de parentesco, no sirven los canales abiertos mediante tuberías sin nombre.

Una tubería con nombre es un archivo con una semántica idéntica a la de una tubería sin nombre, pero ocupa una entrada en un directorio y se accede a él a través de una ruta.

Un proceso puede abrir una tubería con nombre mediante una llamada a open, de la misma forma que abre un archivo ordinario. Así, para comunicar dos procesos mediante una tubería con nombre, uno de ellos debe abrir la tubería para escribir en ella y el otro para leer. La llamada a open tiene un comportamiento ligeramente distinto, según se trata de abrir una tubería con nombre o un archivo ordinario. Así, cuando un proceso abre una tubería con nombre para escribir en ella, se pone a dormir hasta que no haya otro proceso que la abra para leer de ella. Cuando es el proceso lector el primero en abrir la tubería, se pone a dormir hasta que algún proceso la abre para escribir. Esto tiene sentido, ya que no valdría para nada escribir en una tubería cuando nadie va a recoger esos datos.

Otra diferencia que hay entre las tuberías con nombre y los archivos ordinarios, es que para las primeras, el kernel sólo emplea los bloques directos de direcciones de su nodo-i, por lo que la cantidad total de bytes que se pueden enviar en una tubería con nombre en una sola operación de escritura está limitada.



Para controlar los accesos de escritura y lectura de la tubería, el kernel emplea dos apuntadores. Los bloques directos de direcciones del nodo-i son manejados como si fuesen nodos de una cola circular, de tal forma que cuando el apuntador de escritura llega al último de los bloques, empieza por el primero, y lo mismo ocurre para el apuntador de lectura. Ambos apuntadores son gestionados de tal forma que el acceso a la tubería es del tipo FIFO, al igual que ocurriría con las tuberías sin nombre.

Para poder abrir una tubería con nombre, ésta debe existir. Hay dos formas de crearla: desde la línea de comandos, mediante una llamada a `mknod`, y desde un programa mediante una llamada a la función `mknod`.

Para crear en nuestro directorio de trabajo actual una tubería de nombre fifo1, usando `mknod`, debemos escribir:

```
$ /etc/mknod fifo1 p
```

Para crear esa misma tubería con la función `mknod`, debemos incluir en nuestro programa unas líneas parecidas a las siguientes:

```
if (mknod("fifo1", S_IFIFO | permisos, 0) < 0) {
      /* rutina para el tratamiento del error */
}
```

`permisos` es la máscara, ya conocida, de permisos asociados a un archivo.

Como primer ejemplo de aplicación de las tuberías con nombre, vamos a escribir la pareja de programas `llamar_a` y `responder_a`. Estos programas van a permitir que dos usuarios se comuniquen mediante el intercambio de mensajes. Supongamos que el usuario `usr1` desea comunicarse con `usr2`; entonces deberá escribir:

```
$ llamar_a usur2
```

Y `usr2` recibirá en pantalla el siguiente mensaje:

```
LLAMADA PROCEDENTE DEL USUARIO usr1
REPONDER TECLEANDO: reponder_a usr1
```

Si `usr2` desea responder, tendrá que escribir:

```
$ responder_1 usur1
```

Para iniciar la comunicación.

Una vez iniciada la conversación, los dos usuarios se van a enviar mensajes alternativamente, inicia el envío `usr1`. Cada mensaje va a constar de una serie de líneas de texto, finalizando con la línea clave "`cambio`". Esta línea servirá para pasarle el turno al otro usuario. Cuando alguno de los usuarios envíe la línea "`corto`", la conversación terminará.

La comunicación se va a llevar a cabo a través de dos tuberías con nombre creadas por el programa `llamar_a` en el directorio `/usr/tmp`. Estass tuberías tendrán los nombres: `/usr/tmp/fifo_usur1_usr2` y `/usr/tmp/fifo_usur2_usr1`. Una de ellas es para los mensajes que vayan de `usr1` a `usr2`; la otra, para los mensajes que circulen en sentido inverso.

El código del programa `llamar_a` es el siguiente:

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <utmp.h>

#define MAX 256

#define EQ(str1, str2) (strcmp(str1, str2) == 0)

int fifo_12, fifo_21;
char nombre_fifo_12[MAX], nombre_fifo_21[MAX];

char mensaje[MAX];

void limpia(char *cadena) {
	char *p = strchr(cadena, '\n');
	if (p) {
		*p = '\0';
	}
}

void fin_de_transmision(int sig) {
	sprintf(mensaje, "corto");
	write(fifo_12, mensaje, strlen(mensaje) + 1);
	fprintf(stdout, "FIN DE TRANSMISIóN.\n");
	close(fifo_12);
	close(fifo_21);
}

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

```
	char terminal[MAX], *logname;
	struct utmp *utmp1;

	if (argc != 2) {
		fprintf(stderr, "forma de uso: %s usuario\n", argv[0]);
		return -1;
	}

	while((utmp1 = getutent()) != NULL &&
		strncmp(utmp1->ut_user, argv[1], 8) != 0);
	if (utmp1 == NULL) {
		fprintf(stdout, "EL USUARIO %s NO ESTá EN SESIóN.\n", argv[1]);
		return 0;
	}

	sprintf(terminal, "/dev/%s", utmp1->ut_line);
	if ((tty = open(terminal, O_WRONLY)) < 0) {
		perror(terminal);
		return -1;
	}

	logname = (char *) getenv("LOGNAME");
	sprintf(mensaje,	"\n\t\tLLAMADA	PROCEDENTE	DEL	USUARIO
%s\07\07\07\n\t\tREPONDER ESCRIBIENDO: reponder_a %s\n\n", logname, logname);
	write(tty, mensaje, strlen(mensaje) + 1);
	close(tty);

	signal(SIGINT, fin_de_transmision);
	sprintf(nombre_fifo_12, "/usr/tmp/%s_%s", logname, argv[1]);
	sprintf(nombre_fifo_21, "/usr/tmp/%s_%s", argv[1], logname);
	unlink(nombre_fifo_12);
	unlink(nombre_fifo_21);
	if (mknod(nombre_fifo_12, S_IFIFO | 0666, 0) < 0 ||
		mknod(nombre_fifo_21, S_IFIFO | 0666, 0) < 0) {
		perror(argv[0]);
		return -1;
	}

	if ((fifo_12 = open(nombre_fifo_12, O_WRONLY)) < 0 ||
		(fifo_21 = open(nombre_fifo_21, O_WRONLY)) < 0) {
		perror(argv[0]);
		return -1;
	}

	fprintf(stdout, "LLAMADA ATENDIDA.\07\07\07\n");
	do {
		fprintf(stdout, "<===");
		fgets(mensaje, MAX, stdin);
		limpia(mensaje);
		write(fifo_12, mensaje, strlen(mensaje) + 1);
		if (EQ(mensaje, "cambio")) {
			do {
				read(fifo_21, mensaje, MAX);
				fprintf(stdout, "===> %s\n", mensaje);
			} while (!EQ(mensaje, "cambio"));
		}
	} while (!EQ(mensaje, "corto"));

	fprintf(stdout, "EOT\n");
```

```
	close(fifo_12);
	close(fifo_21);
	return 0;
}
```

El programa `responder_a` puede quedar como sigue:

```
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

#define MAX 256

#define EQ(str1, str2) (strcmp(str1, str2) == 0)

int fifo_12, fifo_21;
char nombre_fifo_12[MAX], nombre_fifo_21[MAX];

char mensaje[MAX];

void limpia(char *cadena) {
	char *p = strchr(cadena, '\n');
	if (p) {
		*p = '\0';
	}
}

void fin_de_transmision(int sig) {
	sprintf(mensaje, "corto");
	write(fifo_12, mensaje, strlen(mensaje) + 1);
	fprintf(stdout, "FIN DE TRANSMISIóN.\n");
	close(fifo_12);
	close(fifo_21);
}

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

	if (argc != 2) {
		fprintf(stderr, "forma de uso: %s usuario\n", argv[0]);
		return -1;
	}

	logname = (char *) getenv("LOGNAME");
	signal(SIGINT, fin_de_transmision);
	sprintf(nombre_fifo_12, "/usr/tmp/%s_%s", argv[1], logname);
	sprintf(nombre_fifo_21, "/usr/tmp/%s_%s", logname, argv[1]);
	if ((fifo_12 = open(nombre_fifo_12, O_WRONLY)) < 0 ||
		(fifo_21 = open(nombre_fifo_21, O_WRONLY)) < 0) {
		perror(argv[0]);
		return -1;
	}

	do {
		fprintf(stdout, "===> ");
		fflush(stdout);
```

```
            read(fifo_12, mensaje, MAX);
            fprintf(stdout, "%s\n", mensaje);
            if (EQ(mensaje, "cambio")) {
                  do {
                        fprintf(stdout, "<=== ");
                        fgets(mensaje, MAX, stdin);
                        write(fifo_21, mensaje, strlen(mensaje) + 1);
                  } while (!EQ(mensaje, "cambio"));
            }
      } while (!EQ(mensaje, "corto"));

      fprintf(stdout, "EOT\n");
      close(fifo_12);
      close(fifo_21);
      return 0;
}
```

Lo normal es que a la hora de usar una tubería con nombre sólo haya un programa que escriba en ella y otro que lea de ella, pero eso no es una imposición. Puede haber más de un proceso escribiendo en la tubería y varios procesos leyendo de ella, y el número de procesos escritores no tiene por qué coincidir con el de lectores. En estas situaciones, todos los procesos que utilizan la misma tubería deben implementar mecanismos para coordinar su uso.

8.6 Comunicación FULL-DUPLEX

En los casos estudiados hasta ahora, la comunicación entre procesos es de tipo half-duplex. Esto se debe a que no hemos visto técnicas para conseguir que un proceso lea simultáneamente de dos o más dispositivos. Así, en el código del programa que simula las conversaciones por radio, un programa actúa de hablante y el otro de oyente. Esto está impuesto por la secuencia del código, porque cuando el hablante lee un mensaje del teclado, no puede estar leyendo de su tubería asociada para ver si su interlocutor le envía algo. De igual forma, el oyente lee de la tubería correspondiente, pero en ese instante no atiende al teclado.

La comunicación full-duplex da solución a estos inconvenientes, permitiendo que un proceso pueda actuar como emisor o receptor en cualquier instante, sin tener que sujetarse al protocolo “stop & wait”. Un ejemplo de este tipo de comunicación es una conversación telefónica, en la que los interlocutores pueden hablar o escuchar cuando les apetezca e incluso pueden hablar ambos a la vez. Otro tema es que decidan respetar sus turnos para poder entenderse, pero eso no viene impuesto por el canal, ni por el tipo de comunicación.

Para simular las comunicaciones, full-duplex tenemos que conseguir que un proceso pueda leer de forma simultánea datos procedentes de más de una fuente. Vamos a describir tres técnicas que persiguen este objetivo:

- Interrogación periódica (`polling`).
- Lectura conducida por eventos.
- Multiplexación mediante `select`.

A continuación, vamos a estudiar con más detalle cada una de estas técnicas.

8.6.1. Interrogación periódica

La técnica de interrogación periódica consiste en hacer un recorrido periódico preguntando sobre el estado de las distintas funciones. En el momento que se encuentra una con datos, se procede a leerlos y pasamos a interrogar sobre el estado de la siguiente. Esta técnica implica disponer de funciones que informen sobre el estado de la fuente; por ejemplo, si hay datos en una tubería o en un archivo. Estas funciones no suelen estar disponibles; por tanto, la solución está en interrogar la propia función de lectura. Sabremos que no hay datos cuando la función no pueda leer.

Para impedir que la función de lectura se quede bloqueada en el caso de que no haya datos que leer (puede ser el caso de las tuberías vacías), será necesario reprogramar el acceso al archivo o dispositivo para permitir una lectura no bloqueante. Esto se puede hacer abriendo el dispositivo con el indicador `O_NDELAY` activado o activándolo con una llamada `fcntl`.

La periodicidad del interrogatorio debe calcularse lo suficientemente lenta como para que no sature de trabajo al CPU, y lo suficientemente rápida como para que el usuario no note demoras en la comunicación. Habrá que llegar a una solución de compromiso que estará condicionada por la naturaleza estadística de las peticiones de comunicación.

Uno de los inconvenientes que presenta la lectura por interrogación periódica es el tiempo que se pierde interrogando por el estado de las distintas fuentes de entrada. Por otro lado, las prestaciones del procedimiento van a estar estrechamente relacionadas al período de muestreo que se seleccione. Si el régimen de generación de datos y el período de muestreo son muy altos, puede darse el caso de que alguna tubería o alguna memoria intermedia de entrada/salida se sature y se pierdan datos.

8.6.2 Lectura Conducida por Eventos

En una lectura conducida por eventos habrá varios procesos encargados de atender a las distintas fuentes de datos. En concreto, habrá un proceso por cada fuente. El proceso principal estará parado la mayor parte del tiempo y sólo cuando alguno de sus procesos servidores le comunique que se ha producido un evento, tomará el control y leerá del dispositivo correspondiente. Una de las formas de comunicar eventos que tienen los procesos servidores, es mediante el envío de señales al proceso principal.

Supongamos, como en los ejemplos anteriores, que un proceso recibe información de una tubería y del teclado. Este proceso puede crear dos procesos servidores, uno que va a leer constantemente de la tubería y el otro que va a hacerlo del teclado. Cuando alguna de las fuentes tiene datos, su proceso correspondiente los va a leer y acto seguido se lo va a comunicar a su proceso principal mediante una señal. Por ejemplo, el proceso que lee de la tubería puede enviar la señal `SIGUSR1`, y el que lee del teclado, la señal `SIGUSR2`. Estos procesos, a su vez, escribirán los datos en otra tubería que los comunica con el principal. Las rutinas de tratamiento de las señales anteriores van a leer de esta tubería, con lo que el proceso principal va a tener acceso a los datos.

El uso de señales distintas sirve para discernir las distintas fuentes, pero no resulta operativo cuando el número de fuentes es elevado. Esto es así porque el número de señales reservadas para el usuario es sólo de dos. Para eliminar este inconveniente, podemos recurrir a incluir una cabecera en el conjunto de datos que los procesos servidores envían al proceso principal. Esta

cabecera puede constar de dos campos: la procedencia del mensaje (tubería o teclado), codificada mediante un número entero, y la longitud de la cadena de caracteres transmitida. Así, el proceso principal puede discernir los tipos de mensajes que se encuentran en su tubería de comunicación con los procesos servidores. Con este mecanismo, lo que hemos hecho ha sido implementar un protocolo de comunicación mediante mensajes.

En la figura anterior podemos ver la estructura que tendría el programa de llamadas telefónicas anterior. Los procesos A y B están asociados, respectivamente, a cada uno de los interlocutores. Dando servicio al proceso A, estarán los procesos A1 y A2. El primero de ellos se encarga de leer del teclado y el segundo lee de la salida de la tubería que comunica B con A. A1 y A2 van a escribir mensajes de una tubería que los comunica con el proceso A. Para el proceso B, la estructura es idéntica.

8.6.3 Multiplexación mediante select

La tercera técnica que vamos a ver para conseguir leer de varias fuentes simultáneamente es la multiplexación de la entrada/salida mediante la llamada `select`.

`select` permite interrogar al sistema sobre el estado de varios dispositivos y devuelve cuáles están listos para leer datos de ellos y cuáles lo están para escribir en ellos.

La declaración de `select` es la siguiente:

```
#include <time.h>
int select(int nfds, int &readfds, int &writefds,
      int &exceptfds, struct timeval *timeout);
```

`select` examina los descriptores de archivo especificados en las máscaras de bits `readfds`, `writefds` y `exceptfds`. Se examinan los bits comprendidos entre las posiciones `0` y `nfds-1` (inclusive). El archivo descriptor `fd` se codifica mediante el bit de la posición `1<<fd` de la máscara correspondiente.

El significado de cada máscara es el siguiente:

- `readfds` representa los descriptores de los archivos de lectura por los que preguntamos. Cada bit activo significa que `select` debe interrogar sobre el estado del archivo asociado para ver si tiene datos que leer.
- `writefds` representa los descriptores de los archivos de escritura por los que preguntamos. Cada bit activo significa que `select` debe interrogar sobre el estado del archivo asociado para ver si se puede escribir en él.
- `exceptfds` representa los descriptores de los archivos con alguna condición especial por los que preguntamos. Cada bit activo significa que `select` debe interrogar sobre el estado del archivo asociado para ver si se ha producido alguna condición especial.

Si alguna de las condiciones anteriores no nos interesa, lo indicaremos pasándole a `select` el argumento 0. Los descriptores que se analizan son los comprendidos entre `0` y `nfds - 1`, incluyéndolos.

Como podemos ver, los tres argumentos anteriores se pasan por referencia y no por valor. Esto significa que `select` los va a modificar de acuerdo con el estado de los archivos que interroga. Así, cuando `select` devuelva el control, en `readfds`, `writefds` y `exceptfds` aparecerán activados aquellos bits correspondientes a los descriptores de archivo que reunían algún requisito sobre el que se preguntaba.

`timeout` es un apuntador no nulo que indica el tiempo máximo que se va a esperar desde que `select` entra en ejecución hasta que devuelve el control. `select` puede devolver el control bien porque alguno de los archivos interrogados cumpla los requisitos pedidos o bien porque el tiempo de espera especificado en `timeout` expire. En el caso de que el tiempo de espera se agote, `select` devolverá el valor 0 y en las máscaras aparecerán todos los bits en 0. Si `timeout` es un apuntador `NULL`, la llamada esperará a que se dé algún cambio de estado en alguno de los archivos interrogados.

Las máscaras de bits que codifican los números de los descriptores agrupan 32 posibles descriptores por cada número entero. Si queremos interrogar por archivos con un descriptor más alto, hay que utilizar un arreglo de enteros. Así, por ejemplo, para interrogar por el descriptor número 45, hay que utilizar un arreglo de dos enteros, y el bit número 13 del segundo entero deberá estar a 1. Para encapsular estos detalles, junto con `select` se facilita, ya construido, el tipo `fd_set` y un conjunto de 4 macros para manejar variables de ese tipo. Estas macros se definen con la siguiente interfaz:

Macro	Descripción
`FD_ZERO(fd_set *fdset);`	Pone a cero los bits de `fdset`.
`FD_SET(int fd, fd_set *fdset);`	Activa en `fdset` el bit correspondiente al descriptor `fd`.
`FD_CLEAR(int fd, fd_set *fdset);`	Desactiva en `fdset` el bit correspondiente al descriptor `fd`.
`FD_ISSET(int fd, fd_set *fdset);`	Comprueba en `fdset` el bit correspondiente al descriptor `fd`.

Un ejemplo de uso de estas macros puede ser:

```
#include <sys/types.h>
#include <sys/time.h>
…
fd_set fd_var;
FD_ZERO(&fd_var); /* Pone a cero todos los bits de fd_var. */
FD_SET(5, &fd_var); /* Activa en fd_var el bit correspondiente al descriptor 5.
*/
```

Referencias

Márquez, F. (2004). *Unix Programación Avanzada.* Colombia: Alfa-Omega.

Stevens, W. R. (2008). *Advanced programming in the UNIX environment.* Addison-Wesley.