

Tuberías y FIFOs


Programación en Red / Entornos Distribuidos
 Curso 2024/2025
 Universidad San Pablo-CEU
 Escuela Politécnica Superior
 Campus de Montepríncipe

1

Proceso es un programa en
 exec, se identifica por tener
 memoria privada.

Primitivas para mover
 bits entre procesos

Para Unix basicamente
 todo eran ficheros. Para
 leer de un teclado usaba
 ficheros y esas cosas. Asi
 que se uso un fichero de
 esos raros con una cola
 FIFO mediante un buffer.
 De ahi salio la tuberia.



Tuberías (pipes)

- » La forma más antigua de IPC en Unix.
 - Está presente desde 1973; se encuentra, por tanto, en **todas** las variantes de Unix existentes.
 - Los Unix más antiguos no tienen memoria compartida debido a que el hardware de la PDP-11 hacía difícil su implementación.
- » Una **tubería** permite la transmisión fiable de una cadena o chorro de bytes entre **dos procesos**.
 - Comparable a una **cola** tipo FIFO.
- » Limitaciones de las tuberías:
 - Son *half-duplex* (unidireccionales).
 - Sólo pueden ser usadas **entre procesos que tienen un padre común.**
 - Presentan un tamaño máximo fijo para el buffer.

18-ene.-25 Programación en Red / Entornos Distribuidos página 2

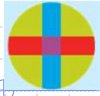
Para comunicarse entre
 procesos, inicialmente se
 usaban ficheros, pero
 muy problematico. Asi
 que se hizo una memoria
 compartida entre proesos.

No tenia sistema de
 memoria virtual
 basicamente

Para poder hacerla
 bidireccional se tendria
 que usar como dos
 tuberia, full-duplex

Sin que el lo lea

2

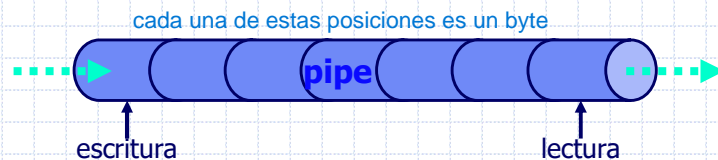


Gestión de las tuberías

- » La **gestión** de las tuberías está integrada en el **sistema de ficheros**.
 - Implementadas clásicamente en el sistema de ficheros.
 - Posteriormente como un caso particular, mediante *sockets* (4.3 BSD) o *STREAMS* (SVR4).

Clasicamente en ficheros, ahora en sockets/sistemas de entrada salida

- » Constituyen un **canal de comunicación**:
 - Los datos escritos en un extremo del canal se leen en el otro extremo.
 - La tubería usa un buffer que define su tamaño:



18-ene.-25

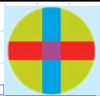
Programación en Red / Entornos Distribuidos

página 3

3

tubo con una zona de escritura y otra de lectura, y un numero de posiciones maximas (byte) donde se puede escribir

Esta en orden, y al ser bytes no hay problema de endianness



Acceso a las tuberías

- » El **acceso** a las tuberías se realiza mediante **descriptores de entrada/salida de Unix**.
 - Los mismos que devuelve la llamada `open()`.
fd
- » La **lectura/escritura** se realiza mediante un **chorro de bytes sin ninguna estructura**. `ByteStream`
 - La lectura de los datos es independiente de la escritura.
 - Permite leer de una vez datos escritos en varias ocasiones.
- » Al realizar una escritura (`write()`) en una tubería...
 - Si hay espacio suficiente, se escriben los bytes y la llamada retorna de inmediato.
 - Si no hay espacio suficiente, la llamada queda bloqueada y la ejecución del proceso es suspendida hasta que otro proceso haga sitio (leyendo datos de la tubería).

Cuando se escribe datos en el fichero, va al disco. Las tuberías igual, van a una zona del disco.

se guarda info de en cuantos golpes se han escrito o leído las cosas

No se hace el mismo numero de lecturas que de escrituras.

Escribes todo lo que quieras, y lees todo lo que quieras.

Si escribo mas rapido de lo que el otro lee, el buffer se llena, sincroniza

18-ene.-25

Programación en Red / Entornos Distribuidos

página 4

4

Las tuberías son bastante pequeñas, unos 512 bytes o así.


Esto funciona porque se usa un buffer

La tubería va a la velocidad del más lento. A la tubería le da igual la velocidad.

Si no cabe en el buffer, se queda bloqueado hasta que se vacíe/se haga la lectura. Igual si escribes más lento, espera el que lee.



Devuelve dos salidas, w y r.
Son dos descriptores de
fichero. EL error que puede
dar es que no haya sitio para
crear descriptores de ficheros.
Pero no es comun.



Creación de una tubería

» La llamada al sistema `pipe()` sirve para crear una tubería:

```
python
import os
r, w = os.pipe()
```

- *w* es un descriptor de fichero de escritura
- *r* es un descriptor de fichero de lectura
- Lo que se escribe en *w* se lee en *r*

- Lanza una excepción si hay error
- En caso de error se fija *errno* con el valor adecuado:
 - *EMFILE* – el proceso tiene demasiados descriptores abiertos
 - *ENFILE* – el sistema tiene demasiados descriptores abiertos

18-ene.-25

Programación en Red / Entornos Distribuidos


página 5

No es una funcion pero lo parece.
Una funcion y una llamada al sistema se diferencian por quien pueden hacerlo. Lo hace el modo supervisor.

Se genera una interrupcion usando un trap

Lo que estas haciendo es cambiando el modo, siendo el sistema operativo quien lo ejecuta

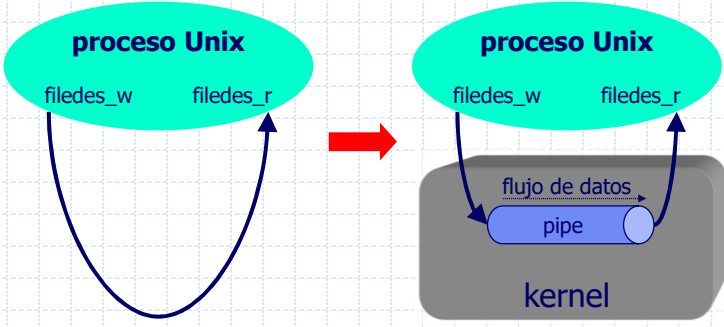
5



Tuberías (pipes)

» La llamada `pipe()` crea *dos* descriptores de fichero. ¿Por qué?

» Respuesta:



18-ene.-25

Programación en Red / Entornos Distribuidos

página 6

La llamada a pipe, hace que se genere un buffer/kernel por el os.


Pero solo de un proceso a si mismo.

Para hacerlo entre procesos, con un FORK, su proceso hijo es quien comparte.

6

Que tiene unix?
Directorios, ficheros, y
especialitos.
Hay dispositivos de modo
caracter y de modo bloque.
Los de modo bloque tienen
una cache (block buffer cache)

Si cogen como tubería el
tamaño del la cache del
disco, y si sale bien la tubería,
no tiene que usar realmente
memoria, se hace todo en la
cache.



Conceptos básicos de las tuberías

- » El **tamaño del buffer** de una tubería es **finito**, es decir, sólo pueden escribirse una cierta cantidad de bytes en la tubería (hasta que no se lean).
 - El tamaño máximo fijo para el buffer es típicamente de **512 bytes**, que es el tamaño mínimo definido por POSIX.
 - Una ventaja de esto es que los datos raramente llegan a escribirse en el disco, sino que quedan en memoria (en la *block buffer cache*).
- » Las tuberías sólo pueden ser usadas entre procesos que tienen un **padre común**.
- » Un proceso creado con `fork()` hereda todas las tuberías abiertas que tenga su padre.

18-ene.-25

Programación en Red / Entornos Distribuidos

página 7


El estandar POSIX no deja
que tengas buffers menor de
512 bytes (igual que el sector
de un disco duro)

7

Despues de hacer un pipe()
hay que hacer un fork()

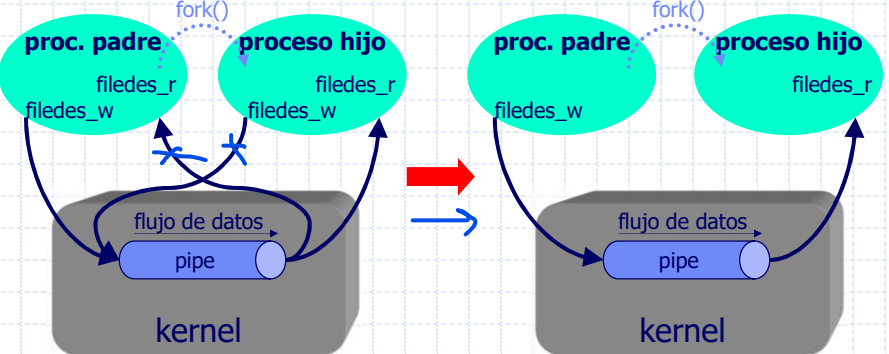
El fork() es una copia exacta
del proceso (menos el PID y
el PPID)

Lo primero que ejecuta el
hijo es la línea siguiente a su
creacion, pero no copia el
Pipe porque no esta en el
proceso, esta en el kernel.



Procesos y tuberías

- » Normalmente, un proceso crea una tubería y luego hace un `fork()`
 - Si se quiere comunicación de **padre a hijo**, se cierra el descriptor de lectura del padre y el de escritura del hijo.
 - Si se quiere comunicación de **hijo a padre**, se cierra el descriptor de escritura del padre y el de lectura del hijo.



18-ene.-25


Programación en Red / Entornos Distribuidos

página 8

No se usan params porque
entonces el otro proceso no
sabe nada al respecto.
Mucho lio.

No tiene realmente riesgos
de seguridad.

8



Entrada/salida en las tuberías

» La tubería usa un **buffer**:

- Cuando el buffer está **lleno**, `write(fd_w...)` se bloquea.
- Cuando el buffer está **vacío**, `read(fd_r...)` se bloquea.
- Si se intenta escribir cuando el extremo **lector** ha **cerrado** se genera `SIGPIPE`.
- Cuando se **cierra** el extremo **escritor**, se recibe un `EOF` (*end-of-file*) tras la recepción de los últimos datos.

» Para comunicación *full-duplex* tendríamos que usar dos tuberías.

» Otra posibilidad útil:

- El hijo hace un `exec()` para ejecutar un programa.
- Pero antes el hijo conecta `fd_r` a `stdin`, con lo que el padre puede enviar datos a la entrada standard del programa.
- Ejemplo...

18-ene.-25

Programación en Red / Entornos Distribuidos

página 9


Al haber dos actores tienen que haber dos `close`, primero cada uno de su lado de escritor después del lector

Hay que tener cuidado con los protocolos de cierre

El que cierra es el escritor, el lector si recibe valor de retorno 0, sabe que ha acabado.

En full duplex si el lector cierra, se llenara el buffer y se bloqueara hasta el final de los tiempos. Entonces para evitar esto se envia una signal de tubería rota.

9

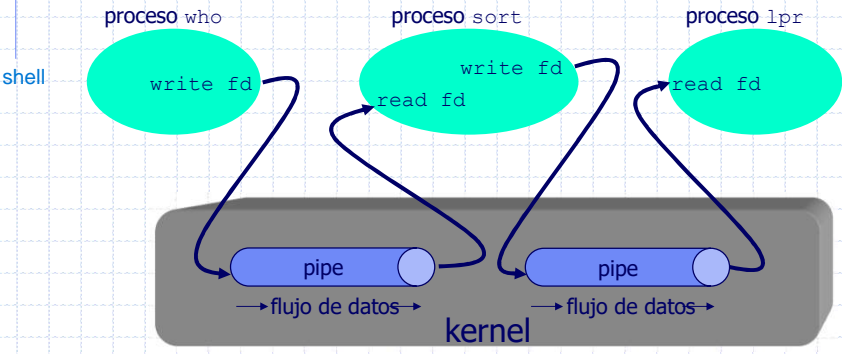


Tuberías en la shell de Unix

» La **shell de Unix** usa **tuberías** y permite que el usuario las manipule:

lineprinter

- Ejemplo: `who | sort | lpr`



18-ene.-25

Programación en Red / Entornos Distribuidos

página 10


who saca una lista de los usuarios conectados al sistema, no ordenados, eso lo hace sort

sort ordena por un monton de opciones para ordenar

ordena lo que le entre , y lo saca por la salida. no hace nada mas

lpr imprime lo que le metas.

10



Ejemplo de código: tuberías


código Python 3

```
import os, sys, time

rd, wd = os.pipe()          # son file descriptors, no file objects
r, w = os.fdopen(rd, 'rb', 0), os.fdopen(wd, 'wb', 0) # file objects
pid = os.fork()
if pid:                      # padre
    w.close()
    while True:
        data = r.readline()
        if not data:
            break
        print("el padre lee: " + data.decode('utf8').strip())
else:                        # hijo
    r.close()
    for i in range(10):
        mensaje = "línea %s\n" % i
        w.write(mensaje.encode('utf8'))
        w.flush()
        time.sleep(1)
```

18-ene.-25 Programación en Red / Entornos Distribuidos página 11

11



FIFOs

- » El método de creación y uso de tuberías clásicas es muy restrictivo.
- » Para paliar este problema aparecen en 1982 las **FIFOs** o **tuberías con nombre** (*named pipes*).
- » Utilizan un nombre en el sistema de ficheros:
 - una FIFO es un fichero de tipo especial `S_IFIFO`.
 - se crean mediante las llamadas `mkfifo()` ó `mknod()`.
- » Características principales:
 - Permiten comunicación entre dos procesos cualesquiera aunque no estén relacionados.
 - › Cada proceso abre la FIFO con el modo que estime oportuno y realiza las operaciones necesarias.
 - Son persistentes (sobreviven al proceso que las crea).
- » **Pregunta abierta:** Las FIFOs son menos seguras que las tuberías clásicas. ¿Por qué?

18-ene.-25 Programación en Red / Entornos Distribuidos página 12

half-duplex
full-duplex

Si se para el escritor se cierran todos los lectores, pero la tubería sigue ahí

12

Se protegen con permisos, pero eso es un mecanismo de protección no de seguridad. Si alguien quiere entrar puede hacerlo haciéndose sudo y cambiando los bits de permisos


Protección es evitar que nos hagamos daño, seguridad es para que no nos entre un chungo, sobre todo exterior

Se manejan como si fueran archivos, con open, read, write, close

rm elimina el nombre con un link, cuando se queda sin nombres es cuando se elimina un archivo

mkfifo
cat > t
|
cat < t
|

El cat hace un open



FIFOs

» Desde la shell, podemos crear una con el comando `mkfifo`. Por ejemplo:

```
unix> mkfifo /tmp/fifo
unix> ls -l /tmp/fifo
prw-rw-rw 1 rgg users 0 Jan 16 14:04 /tmp/fifo
```

» En una ventana leemos de la FIFO:

```
unix1> cat </tmp/fifo
```

» En otra ventana, escribimos a la FIFO:

```
unix2> cat >/tmp/fifo
```

» Tecleamos algunas líneas de texto. Cada vez que se pulsa `ENTER` la línea es enviada por la FIFO, y aparece en la primera ventana

» Cerramos la FIFO pulsando `Ctrl+D` en la segunda ventana. Eliminados la FIFO con:

```
unix> rm /tmp/fifo
```

18-ene.-25

Programación en Red / Entornos Distribuidos

página 13


La unica manera de garantizar que los mensajes se lo lleve entero un proceso es que queta en el buffer

Si no se podrian mezclar

13

hay comando mkfifo desde la consola.

Aparece la entrasda en el sistema de ficheros. El unlink para borrarlo.



Creación de una FIFO

» La llamada `mkfifo()` crea una FIFO:

```
import os
mkfifo(nombre, modo=0666)
mknod(nombre, modo=0666, dispositivo=0)
```

Nombre de la FIFO en el sistema de ficheros

Esta llamada implica automáticamente los flags `O_CREAT/O_EXCL`. El resto de flags, igual que en `open()`

• Lanza una excepción si hay error

• En caso de error se fija *errno* con el valor adecuado:

- *EEXISTS*– el fichero ya existe
- *ENOSPC*– no puede extenderse el fichero o directorio
- *EROFS*– sistema de ficheros de sólo lectura

mkfifo hace lo mismo pero pone el `dispositivo=0` siempre

» Puede usarse también `mknod()`.

- Su explicación queda como ejercicio.

18-ene.-25


Programación en Red / Entornos Distribuidos

página 14

14

Se manejan con
descriptores

Para cambiar el
comportamiento de una
FIFO, con `fcntl()`. Opera
sobre descriptor abierto,
tiene que estar la fifo
abierta.



Otras llamadas al sistema para FIFOs

- » Apertura de una FIFO ya creada: `open()`
- » Cambio de flags en FIFOs y pipes: `fcntl()`

```
import os, fcntl
flags = fcntl.fcntl(fd, fcntl.F_GETFL)
flags |= os.O_NONBLOCK;
try:
    fcntl.fcntl(fd, fcntl.F_SETFL, flags)
except IOError:
    print "error en fcntl"
```

como el += pero con OR

- » Borrado de una FIFO: `unlink()`
- » Pueden usarse llamadas al sistema de E/S ó
funciones de E/S de la biblioteca standard:

```
fd = os.open(fifo_path, os.O_WRONLY);
os.write(fd, datos, longitud_datos);
os.close(fd);
.....
with open(fifo_path, "r") as fileobject:
    buffer = fileobject.read()
```


18-ene.-25
Programación en Red / Entornos Distribuidos
página 15

`creat()` era para hacer
ficheros

la pone en modo para que
no se bloquee cuando se
llena el buffer, si se llena
da error.

Modo de solo escritura y
devuelve un descriptor.

15



Procesos y FIFOs

- » Una FIFO debe tener al menos un **proceso lector** y
uno **escritor**.
 - Puede haber muchos lectores y escritores, pero hay que
tener en cuenta los posibles solapes en las escrituras.
- » La constante `PIPE_BUF` define el **número máximo**
de **caracteres** que se pueden escribir en una FIFO
atómicamente.
- » Su funcionamiento depende del flag `O_NONBLOCK`:
 - Si no se especifica, un `open()` de sólo lectura se bloquea
hasta que otro proceso abra el FIFO para escritura, y
viceversa.
 - Si se especifica, un `open()` de sólo lectura retorna
inmediatamente. Un `open()` de sólo escritura retorna un
error si ningún proceso tiene la FIFO abierta para lectura.

18-ene.-25
Programación en Red / Entornos Distribuidos
página 16

Si queremos que funcione
de un solo golpe, las
peticiones tienen que ser
de size inferior que el
`PIPE_BUF`.


Si se activa y se llena, da
error.

Si no se activa y se llena,
duerme el proceso.

16

Al ser como un archivo se podría abrir en modo lectura y escritura. Pero al final significa que el que recibe los mensajes eres tu mismo. Así no se hace una full-duplex. Cuando el buffer se va llenando, ve que lo tienes abierto y puedes recibir, así que lo recibes tu.

Realmente habría que crear una tubería para cada cliente. El cliente es un proceso y te tendría que enviar el PID. Se puede usar como protección porque el PID lo conoce solo el cliente.




Ejemplo de aplicación de FIFOs

- » Ejemplo de aplicación: Un servidor y varios clientes que se comunican con aquel mediante una FIFO de nombre conocido.
 - Los clientes escriben sus peticiones en la FIFO.
 - › Las peticiones deben tener una longitud inferior a `PIPE_BUF` (que se encuentra en el módulo `select`) para que las escrituras sean atómicas y no se produzcan solapamientos.
 - Pero, ¿cómo responde el servidor a los clientes?
- » Una solución es que cada cliente envíe su PID en la petición.
 - El servidor crea entonces una FIFO específica por cada cliente, con un nombre basado en su PID.
 - El servidor escribe la respuesta en esta FIFO. El lector, que conoce su nombre, lee la respuesta.

18-ene.-25 Programación en Red / Entornos Distribuidos página 17

17



Precauciones a tomar con las FIFOs

- » Crear una FIFO y abrirla para lectura y escritura involucra tres llamadas al sistema.
 - La llamada `pipe()` hace lo mismo de una sola vez.
- » Los siguientes casos son llamadas al sistema bloqueantes, por regla general:
 - Un proceso abre una FIFO en modo de sólo lectura sin que existan otros procesos escritores.
 - Un proceso abre una FIFO en modo de sólo escritura sin que existan otros procesos lectores.
- » **Pregunta abierta:** ¿Por qué una FIFO se abre en modo de sólo lectura o escritura pero no en modo lectura/escritura?

18-ene.-25 Programación en Red / Entornos Distribuidos página 18

FIFO lo tiene que hacer de 3 llamadas, pero sin los `fork`. `pipe()` lo hace de golpe, pero esta la fiesta de los `forks`

El `open` te duerme hasta que el destino haga el `open`. Y una vez se hace los `open`, ya lo que se bloque son los `read` y `write`.

El descriptor queda unido a un solo proceso y lo recibe el mismo.

18



Chorros de bytes y mensajes

- » Modelos de E/S para la transferencia de datos en canales punto a punto (como FIFOs y pipes):
 - **Chorro de bytes** (*byte stream*).
 - › no existen delimitadores.
 - › el receptor no puede saber si los datos que recibe fueron escritos de una vez o con muchas operaciones `write()`.
 - **Mensaje** – dos posibilidades.
 - › Uso de una **estructura común** para el **mensaje**, cuyo formato es conocido tanto por el emisor como por el receptor.
 - Requiere acuerdo previo entre ambos, generalmente en forma de código compartido.
 - › Uso de **delimitadores** de fin de mensaje.
 - Sólo requiere acuerdo en el carácter delimitador.
- » **Pregunta abierta:** ¿qué problemas plantea cada uno de los tres modelos?

Como movemos datos entre cliente y servidor?

Es continuo y no hay manera de saber si se han escrito de un golpe o de varios.

Estructuras que pueden tener un size específico.

Cosas como que el mensaje termina cuando se recibe un punto.

18-ene.-25

Programación en Red / Entornos Distribuidos

página 19

19

De la estructura común como pierdas la sync se lia mucho.

El bytestream significa que tiene que estar abierto todo el rato mientras se recibe y despues cerrarlo y repetir

Problema del limitador es que el limitador no puede estar incluido en el mensaje. Se puede usar un caracter de escape como 'backslash', que tiene problema que para usarlo tienes que escapar el caracter de escape



Ejemplo de código: FIFOs

código Python 3

```
import os, time, sys
nombre_fifo = 'prueba_fifo'

def hijo():
    fifo_escritura = os.open(nombre_fifo, os.O_WRONLY)
    numero = 0
    while True:
        time.sleep(1)
        mensaje = 'Contador %03d\n' % numero
        os.write(fifo_escritura, mensaje.encode('utf8'))
        numero = (numero+1) % 5

def padre():
    fifo_lectura = open(nombre_fifo, 'r')
    while True:
        dato = fifo_lectura.readline()[:-1]
        print('Proc %d ve "%s" en %s' % (os.getpid(), dato, time.asctime()))

if not os.path.exists(nombre_fifo):
    os.mkfifo(nombre_fifo)
if os.fork() == 0:
    hijo()
else:
    padre()
```

ejemplo con os


cont del ej pero sin os

18-ene.-25


Programación en Red / Entornos Distribuidos

página 20

20



¿Preguntas?



18-ene.-25

Programación en Red / Entornos Distribuidos

página 21