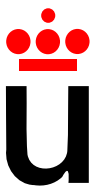
Tema 5: Comunicando procesos

Enrique Soriano

Laboratorio de Sistemas,
Grupo de Sistemas y Comunicaciones,
URJC

17 de marzo de 2010



Universidad
Rey Juan Carlos

Hasta ahora...

- ... los procesos que ejecutábamos tenían su entrada y su salida en la consola (/dev/cons).
- Pero `stdin, stdout, y stderr` se pueden *redirigir* a otros ficheros.

Redirecciones en el Shell

- > y < son caracteres especiales para el shell.
- > para redirigir la salida estándar a otro fichero.
- < para redirigir la entrada estándar a otro fichero.
- P. ej.

```
term% ps > procesos.txt
term% wc -l procesos.txt
term% wc procesos.txt
term% echo hola que tal > hola.txt
term% cat < /NOTICE
```

Redirecciones en Shell

- La mayoría de los comandos que aceptan ficheros como argumentos también leen de su entrada estándar cuando se invocan sin argumentos:

```
term% cat < procesos.txt
term% cat procesos.txt # no es lo mismo!!
term% cat > carta.txt
```

Plan 9: /fd

- Los comandos que sólo admiten argumentos (raro) se puede usar /fd/0 y /fd/1.

```
term% cat /fd/0 # cat lee de su entrada estándar
term% cp /fd/0 /fd/1 # se lee de stdin, se
escribe en stdout
```

/dev/null

- `/dev/null` es "ninguna parte".
  `term% script > /dev/null`
- Los comandos que ejecutan en *background* (&) tiene su entrada estándar redirigida a `/dev/null`. P. ej:
  `term% cat &`

Salida de errores

- Existe para que no se mezclen las salidas. P. ej.
  ```
  term% ls /kokoko > /tmp/afile
  ls: /kokoko does not exist
  term% cat /tmp/afile # no hay nada!
  ```
- Los hijos heredan la tabla de descriptores de fichero: desde el *inicio de los tiempos* tienen 0 (stdin) , 1 (stdout) y 2 (stderr) apuntando a `/dev/cons`!

Más redirecciones

- `>>`
  Redirige la salida sin truncar. Es un open() + seek(), no como un create().
- `>[numero]`
  Redirige la posición especificada en la tabla de FDs. P. ej.
  `term% ls * >[2] /tmp/errores`
- `>[numero1=numero2]`
  Duplica el descriptor numero2 en el numero1. Atención: se aplican de izquierda a derecha. P. ej.
  `term% ping 172.26.0.1 > /dev/null >[2=1]`

Conjugando

- Se pueden conjugar:
  ```
  term% cat < /tmp/afile > /tmp/copia-afile
  term% wc -l /tmp/afile > /tmp/cuentas >[2] /dev/null
  ```
- `term% echo 'error!!!' >[1=2]`
- `term% cat < afile > afile # afile se trunca!`
  Solución: Usa un fichero temporal.

Dup

- `int dup(int oldfd, int newfd);`
- Duplica FDs: dado un FD (oldfd) con un fichero abierto, devuelve otro FD (newfd) con el mismo fichero abierto.
- Si `newfd` es -1, escoge la posición libre más baja de la tabla. Si es distinto, cierra (si es necesario) el fichero abierto en esa posición y duplica el descriptor.

ver dup.c

Pipes

- Estilo "Unix": pequeños programas que leen datos por su entrada estándar, los procesan, y los escriben por su salida estándar.
- |
  Los *pipes* nos permiten concatenar comandos conectando la entrada de uno a la salida de otro.

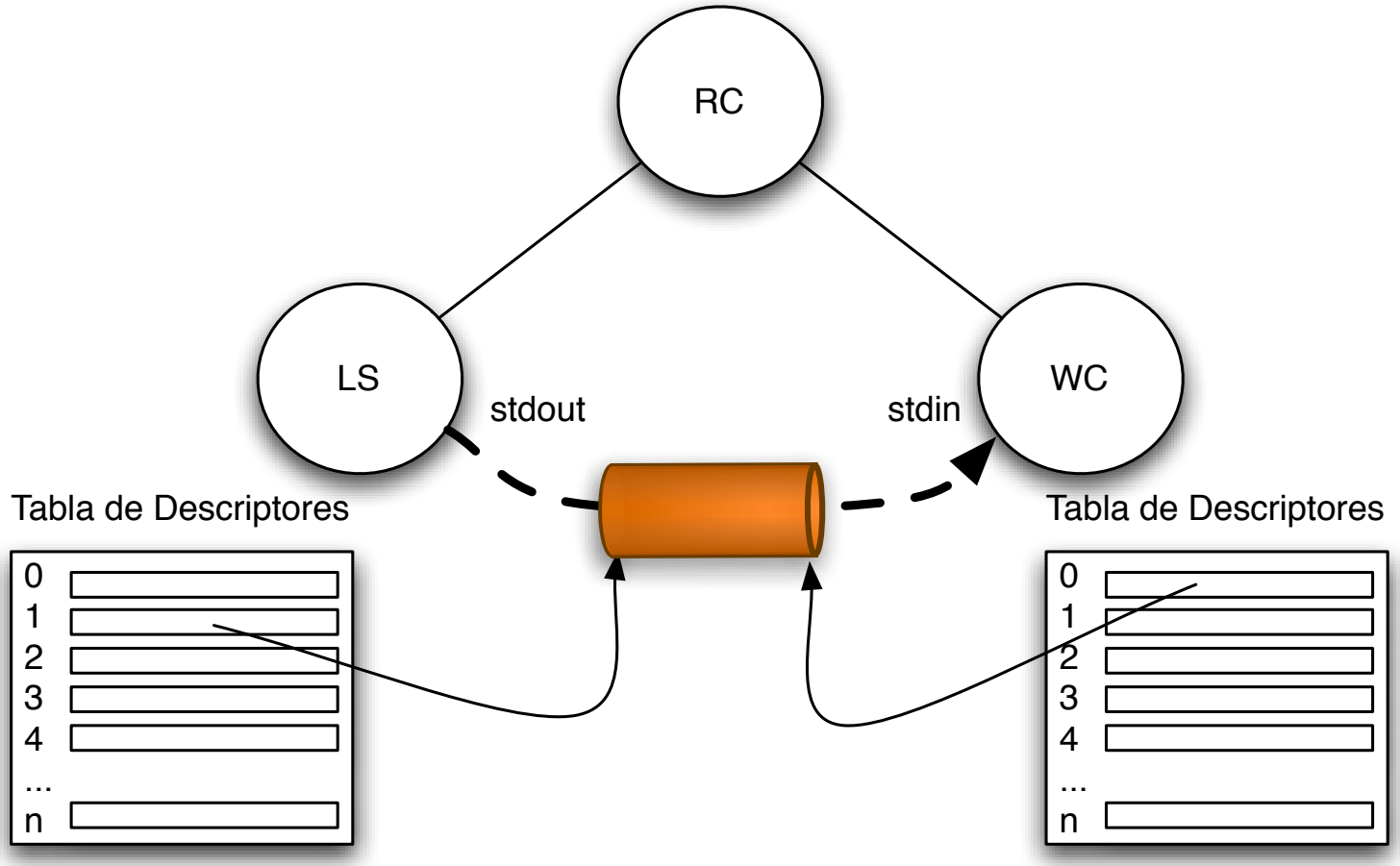
Pipes

- Cada extremo del pipe se comporta como un fichero.
- Todo lo que se escribe en un extremo se lee en el otro.

Pipes

P. ej.

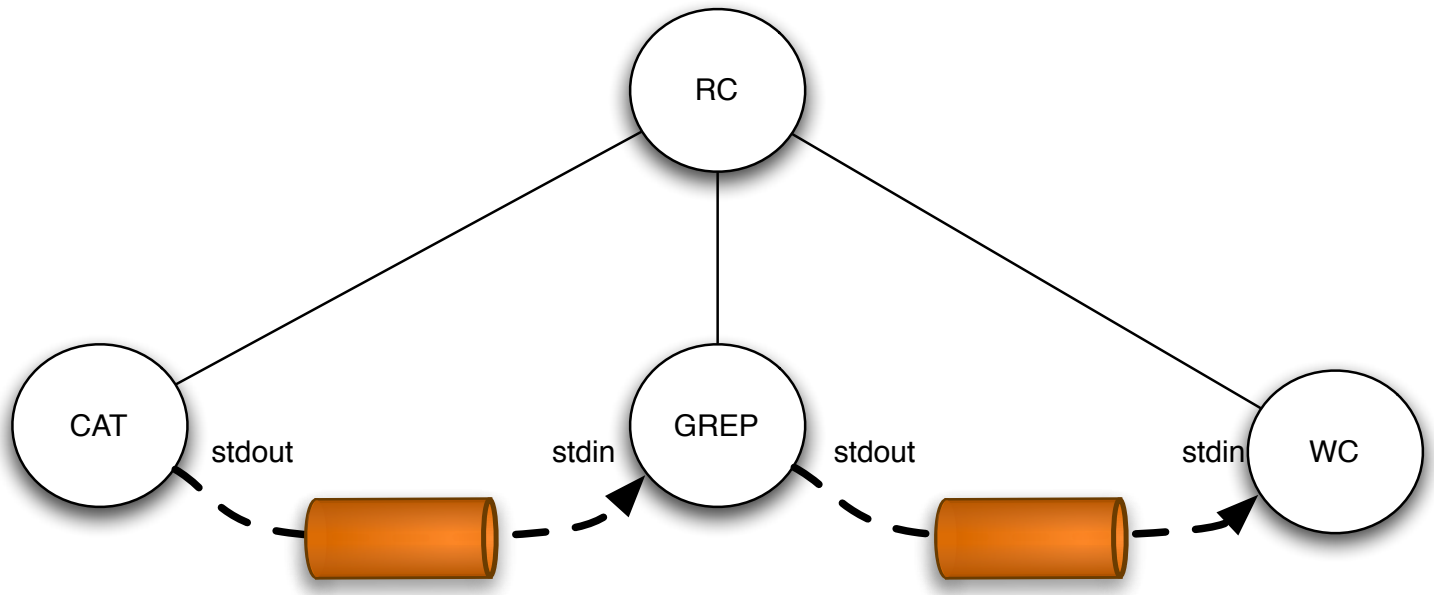
```
term% ls *.txt | wc -l
```



Pipes

P. ej.

```
term% cat *.txt | grep 'Enrique Soriano' | wc -l
```



Pipes

Más ejemplos:

- `term% ps | grep fossil | grep Sleep | wc -l`
- `term% echo ls | rc`
- `term% broke | rc`

Pipe

- `int pipe(int fd[2]);`
- El array contendrá los dos FDs del pipe (uno para cada extremo).
- En Plan 9 los pipes son *full-duplex*: se puede enviar en ambos sentidos simultáneamente. Ojo: en Unix son *simplex*.
- No se debe usar como medio de almacenamiento.
- Se debe crear antes de llamar a fork() para ambos procesos lo compartan.

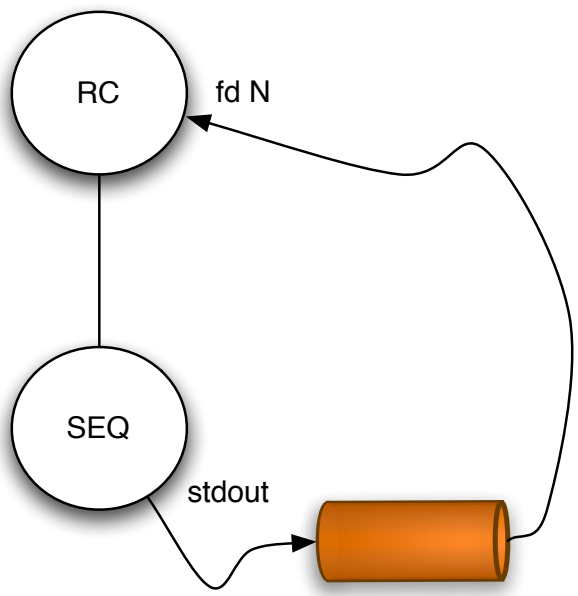
Pipe

- Debemos cerrar el extremo que no vamos a usar.
- Se conservan los *límites de escritura* (en Plan 9).
- Un pipe tiene un buffer limitado.
- Leer de un pipe vacío te deja bloqueado.
- Escribir en un pipe lleno te deja bloqueado.
- Leer si no hay nadie al otro lado: retorna 0 bytes. No se puede diferenciar de un write() de 0 bytes.
- Escribir en un pipe sin nadie al otro lado: error.
- Ambos procesos deben leer/escribir en paralelo, no secuencialmente, ¿Qué pasa si se llena el pipe? $\longrightarrow$ **deadlock**.

ver pipe.c, pipexec.c, piperead.c, fill.c

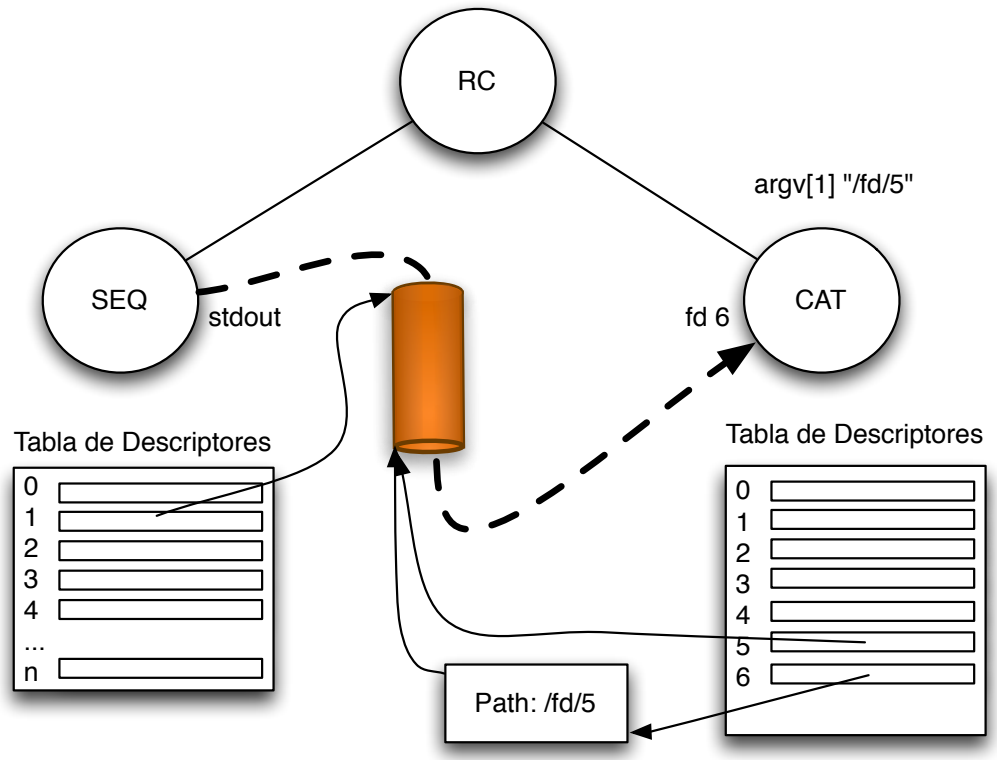
Shell: comillas invertidas

- El shell reemplaza ‘{...} por la salida de la ejecución del comando.
- `term% echo ‘{seq 1 5}`
  `1 2 3 4 5`



Pipes no lineales

- El shell reemplaza <{...} por el path del pipe en el que se escribe la salida de los comandos ...
- `term% cat <{seq 1 3} # se traduce en cat /fd/5`
  1
  2
  3



Síncrono vs. Asíncrono

- Síncrono: la comunicación se realiza cuando se llama a una función.
- Asíncrono: la comunicación se realiza en cualquier momento, no al realizar una llamada. Se interrumpe la ejecución normal del proceso.

Notas

- Las notas (*señales en Unix*) son mensajes que se procesan de forma asíncrona. P. ej. al pulsar *Del* en una ventana, se envía la nota a todos los procesos que comparten dicha ventana.
- En realidad, se envía la nota a un grupo de procesos llamado `notegp`.
- Normalmente hay un `notegp` por cada ventana.
- El proceso que recibe la nota se encarga de procesarla cuando le toca ejecutar.

Atnotify

- `int atnotify(void (*f)(void *, char*), int in);`
- Si `in` es 1 la función f como manejador de notas. Si es 0, la elimina.
- Si el manejador retorna 0, es que no ha procesado la nota. Si devuelve 1, es que sí lo ha hecho.
- Si ningún manejador procesa una nota, el programa muere.

Notas

Las más comunes:

- `interrupt` → el usuario ha interrumpido el proceso (Del).
- `hangup` → se ha cerrado la conexión de E/S.
- `alarm` → se puso una alarma y ha saltado.
- `sys: bad address` → el kernel nos está "matando" por usar una dirección de memoria inválida (p. ej. nil).

ver interrupt.c

Notas

En general, son peligrosas y hay que evitar su uso:

- Interrumpen ciertas llamadas al sistema (las bloqueantes: read(), write(), etc.).
- Rompe tu idea de un único flujo de control secuencial.

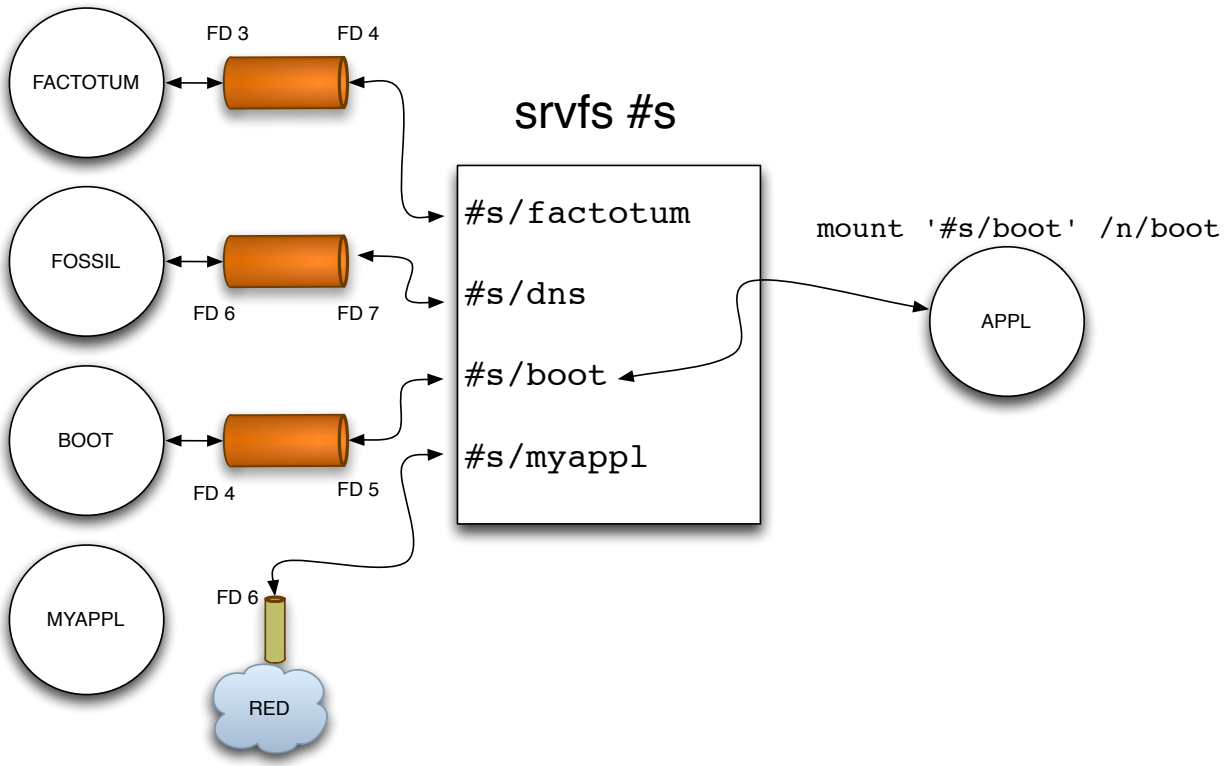
Alarmas

- `long alarm(ulong milsec);`
- Permiten determinar un *timeout* para una operación bloqueante.
- Si `milsec` es 0, desactiva la alarma.
- Retorna el número de milisegundos que faltaban para saltar.

ver alarm.c

Plan 9: /srv

- Plan 9 nos deja "pinchar" descriptores de fichero en un directorio llamado `srv`:



ver srv.c

Polling vs. Eventos

Hay dos modelos para recibir mensajes:

- Polling: cada cierto tiempo se mira si hay algún mensaje que procesar $\longrightarrow$ se malgastan recursos si los mensajes son poco frecuentes. Si se reduce la frecuencia de polling, se procesan con retardo.
- Eventos: se subscribe a un tipo de mensajes, y se recibe un evento cuando hay alguno.

Plumber

- Plumber es un sistema de ficheros que tiene *reglas* y *puertos*. Una regla determina a qué puerto se debe reenviar el mensaje. Un puerto envía el mensaje a todos los procesos subscritos.
- `/mnt/plumb/rules` → reglas actuales.
- `/mnt/plumb/send` → fichero para escribir los mensajes.
- El resto son los puertos disponibles.