

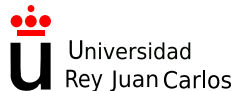
Comunicación entre procesos

Sistemas Operativos

Enrique Soriano

GSYC

26 de noviembre de 2018



(cc) 2018 Grupo de Sistemas y Comunicaciones.

Algunos derechos reservados. Este trabajo se entrega bajo la licencia Creative Commons Reconocimiento -

NoComercial - SinObraDerivada (by-nc-nd). Para obtener la licencia completa, véase

<http://creativecommons.org/licenses/by-sa/2.1/es>. También puede solicitarse a Creative Commons, 559 Nathan

Abbott Way, Stanford, California 94305, USA.

IPC

- Los procesos necesitan comunicarse para sincronizarse, intercambiar información, etc.
- Existen distintos mecanismos de IPC para los procesos que no comparten memoria.

Pipe

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

Pipe

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

Llamada al sistema: pipe

- pipe: crea un nuevo pipe. Se pasa array contendrá los dos FDs del pipe (uno para cada extremo). En Unix son *simplex*, se escribe en fd[1] y se lee de fd[0].
- Se debe crear antes de llamar a fork() para que ambos procesos lo compartan.

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

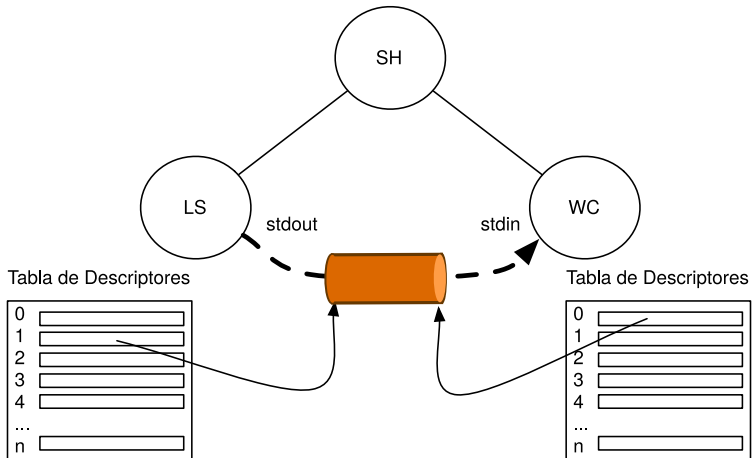
Llamada al sistema: pipe

- Debemos cerrar el extremo que no vamos a usar.
- En general, no se conservan los *límites de escritura*.
- Un pipe tiene un buffer limitado: se puede llenar.
- 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? → **interbloqueo**.

En el shell

P. ej.

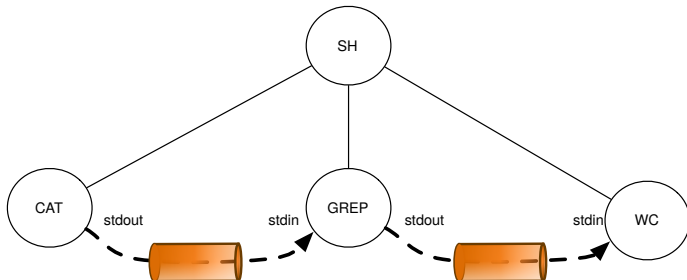
```
$> ls *.txt | wc -l
```



En el shell

P. ej.

```
$> cat *.txt | grep 'pepe' | wc -l
```



Fifo

- Son pipes con nombre: una ruta en el espacio de nombres. Se pueden crear en el shell con el comando `mkfifo`
- En el shell, se eliminan con `rm`. Si se borra, los que lo están usando lo pueden seguir usando (como cualquier fichero en este caso).

Fifo

- Se abren con `open`, como cualquier fichero, para leer o para escribir.
- Un `open` de solo-lectura deja bloqueado al proceso hasta que otro proceso abre el fifo para escribir en él. Si el fifo se borra mientras estás bloqueado, te quedas bloqueado de por vida.
- Se puede forzar una apertura no bloqueante (NON-BLOCKING).
- Un `open` **no bloqueante** de solo-escritura falla si ningún proceso tiene abierto el fifo para leer.

Fifo

Igual que los pipes:

- Cuando no hay ningún proceso que pueda escribir, se leen 0 bytes.
- Cuando no hay ningún proceso que pueda leer, una escritura falla (se recibe la señal SIGPIPE).

Llamada al sistema: mkfifo

- `mkfifo`: crea un fifo con los permisos indicados. El dueño/grupo es el UID/GID del proceso. Si el fifo ya existe, retorna error (-1).

```
int mkfifo(const char *path, mode_t mode);
```

Señales

- Son un mecanismo para notificar cosas a un proceso, o interrumpir su ejecución.
- Son difíciles de entender y usar.
- Pueden ser:
 - Síncronas: se entregan inmediatamente porque son consecuencia de su ejecución, el proceso está ejecutando. Suelen ser enviadas por el sistema.
 - Asíncronas: la señal se entregará y manejará cuando el proceso sea planificado.

Señales

- Cuando recibe una señal, el proceso puede ignorarla o manejarla (con la acción por omisión o definir el manejador).
- Las señales pueden anidarse.
- Hay distintas acciones por omisión dependiendo del tipo de señal:
 - Terminar el proceso.
 - Provocar un core dump y terminar el proceso.
 - Ignorar la señal.
 - Suspender el proceso.
 - Reanudar su ejecución.

Señales

Las señales más populares son:

- 1 SIGHUP, default action: terminate process , description: terminal line hangup (terminal connection lost)
- 2 SIGINT, default action: terminate process , description: interrupt program (ctrl-c in terminal)
- 3 SIGQUIT, default action: create core image , description: quit program (ctrl-\ in terminal)
- 4 SIGILL, default action: create core image , description: illegal instruction
- 8 SIGFPE, default action: create core image, description: floating-point exception
- 9 SIGKILL, default action: terminate process , description: kill program (cannot be ignored)
- 11 SIGSEGV, default action: create core image , description: segmentation violation
- 13 SIGPIPE, default action: terminate process , description: write on a pipe with no reader
- 14 SIGALRM, default action: terminate process , description: real-time timer expired
- 15 SIGTERM, default action: terminate process , description: software termination signal (e.g. shutdown)
- 17 SIGSTOP, default action: stop process , description: stop (cannot be caught or ignored)
- 18 SIGTSTP, default action: stop process , description: stop signal generated from keyboard
- 19 SIGCONT, default action: discard signal , description: continue after stop
- 20 SIGCHLD, default action: discard signal , description: child status has changed
- 30 SIGUSR1, default action: terminate process , description: User defined signal 1
- 31 SIGUSR2, default action: terminate process, description: User defined signal 2

En el shell

- El comando `kill` envía una señal. Por omisión envía un `SIGTERM`.
`$> kill -9 2324`
- El comando `killall` envía una señal a los procesos fijándose en su nombre.
- El comando `pkill` hace lo mismo, pero aplicando un patrón (expresión regular).

En el shell

¿Recuerdas el *job control*?

- Toda sesión tiene un terminal controlador (tty).
- Una sesión tiene distintos grupos de procesos. La sesión está liderada por un proceso. El id de la sesión es el PID del *proceso líder de la sesión*.
- Un grupo de procesos está liderado por un proceso. Un grupo de procesos tiene un id: PGID, que es el PID del *proceso líder del grupo*. Un *job* es un grupo de procesos.
- Como mucho, un grupo de procesos de la sesión está en primer plano en el terminal.
- El terminal manda las señales a todos los procesos del grupo que está en primer plano. P. ej. Ctrl-c
- Si un proceso de un grupo que no está en background intenta leer del terminal, se le manda un SIGTTIN y se quedará parado.

En el shell

Ejemplo:

```
$> sleep 1000 | cat | wc &
```

```
[1] 6308
```

```
$> jobs
```

```
[1]+  Running
```

```
sleep 1000 | cat | wc &
```

```
$> ps j
```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
4381	4391	4391	4391	pts/0	6310	Ss	1000	0:00	/bin/bash
4391	6306	6306	4391	pts/0	6310	S	1000	0:00	sleep 1000
4391	6307	6306	4391	pts/0	6310	S	1000	0:00	cat
4391	6308	6306	4391	pts/0	6310	S	1000	0:00	wc
4391	6310	6310	4391	pts/0	6310	R+	1000	0:00	ps j

Sesión y grupos de procesos

- El grupo de procesos y la sesión se hereda del padre.
- Después de un `exec` se conserva el grupo de procesos.
- La llamada al sistema `setpgid` sirve para cambiar el grupo de procesos.
- La llamada al sistema `setsid` crea una nueva sesión, siendo el proceso llamador el líder.
- Para crear un *demonio*, debemos crear una sesión y dejarlo sin terminal controlador. Eso lo hace la función `daemon` de la `libc`.

Llamada al sistema: kill

- `kill`: manda una señal a un proceso. Para poder hacerlo, el UID del proceso destino debe ser el mismo que el del proceso (o ser root), la única excepción es `SIGCONT` (pero el proceso deber ser descendiente).
- Si `pid` es 0, se pone la señal a todos los procesos del grupo de procesos. Si es -1, manda la señal a todos los procesos que se pueda menos a los del sistema y él mismo.

```
int kill(pid_t pid, int sig);
```

Llamada al sistema: killpg

- killpg: como la anterior, pero manda la señal a un grupo de procesos.

```
int killpg(pid_t pgrp, int sig);
```

Llamada al sistema: signal

- `signal`: Asigna una acción para una señal. Registra un manejador para una señal, poner su acción por omisión (`SIG_DFL`) o ponerla como ignorada (`SIG_IGN`).
- Sobrescribe el estado anterior.
- Es una versión simplificada de `sigaction`. Usaremos `signal` por simplicidad.
- No es portable, difiere en distintos sistemas, y ha variado con el tiempo.

```
typedef void (*sig_t) (int);  
sig_t signal(int sig, sig_t func);
```

Si no quieres esperar por tus hijos...

- Si tu programa no quiere esperar (llamar a `wait`) por los procesos que crea, se debe avisar al sistema para que no se queden **zombies**.
- Se hace ignorando la señal `SIGCHLD`.

```
signal(SIGCHLD, SIG_IGN);
```



Manejadores

- Los manejadores no retornan nada (`void`).
- Se usará la pila del proceso si no se dice lo contrario (p. ej. con `signalstack` de `glibc`).
- Como se pueden recibir señales mientras que se maneja una señal → manejador debe ser **reentrante**: se tiene que poder llamar de nuevo antes de que haya retornado. Para ello: no usar variables globales/estáticas, no debe llamar a funciones no reentrantes (p. ej. `malloc`, `free`, etc.), no modificar `errno`, ...
- Debería ser lo más pequeño que se pueda.

Llamada al sistema: alarm

- `alarm`: programa un temporizador para recibir un `SIGALRM`. Llamando con el valor 0, se anula el temporizador. Si se llama dos veces, se sobrescribe el temporizador.
- Se usa para poner un *timeout*. Hay que usarlo con precaución y medida.

```
unsigned int alarm(unsigned int seconds);
```

Interrumpiendo llamadas

- Algunas llamadas al sistema son *lentas*: el proceso se puede bloquear mientras realiza la llamada: read, write, open, etc.
- Si se pone una señal mientras que se está realizando una llamada al sistema lenta, la llamada fallará (es interrumpida).
- Se puede pedir que las llamadas al sistema lentas que lo soporten, sean reiniciadas (si es posible) cuando se recibe una señal.
- Esto se hace con `siginterrupt`: activa (0) o desactiva (1) el reinicio para una señal.

```
int siginterrupt(int sig, int flag);
```

Bloquear señales

- Las señales se pueden bloquear temporalmente en partes de nuestro programa en las que no queremos ser interrumpidos.
- **No es lo mismo** que ignorar la señal.
- Si una señal está bloqueada y se recibe, se queda pendiente de entregar. No se pierden.
- En ese caso, cuando desbloqueamos la señal, se entregará la pendiente.

Bloquear señales

- La **máscara de señales** del proceso indica qué señales están bloqueadas.
- Podemos modificar o ver las señales bloqueadas con la llamada al sistema `sigprocmask`.
- `sigpending` nos da el conjunto de señales que están pendientes de entregar.

Señales

- El kernel apunta los manejadores de señales en la estructura que mantiene del proceso
- Todo el estado del proceso relacionado con las señales se hereda del padre.
- Después de un exec se restablecen las acciones por omisión, pero se respetan la configuración sobre señales bloqueadas.

Señales

¿Cómo funcionan las señales?

- 1 La señal se entrega al entrar al kernel por cualquier motivo (llamada al sistema, interrupción, etc.) o al salir del kernel.
- 2 Se añade un *signal frame* en la pila que contiene el contexto actual del proceso (registros, etc).
- 3 Se mete en la pila un PC de retorno que corresponde al stub de la libc de la llamada al sistema `sigreturn`.
- 4 Se pone a ejecutar al manejador de la señal.
- 5 Cuando el manejador retorna, se retorna al stub de la llamada al sistema `sigreturn()`: se entra al kernel.
- 6 La llamada al sistema saca el *signal frame* de la pila y restaura el contexto.
- 7 Cuando se vuelva a planificar el proceso, continuará donde fue interrumpido.