Comunicación entre Procesos
(IPC)

Sistemas Operativos
DC - FCEN - UBA

4 de Septiembre de 2025

Temario

Temario

Funciones Útiles (1)

Importante

¡Complementar todo con el manual!

Procesos

- `int fork()`: Crea un nuevo proceso, copiando el actual. Retorna **0** en el proceso hijo, y el **PID** del proceso creado en el proceso padre.
- `void exit(int status)`: Termina el proceso actual utilizando valor de **status** como valor de retorno.

Funciones Útiles (2)

Pipes

- `int pipe(int descriptores[2])`: Crea **un pipe unidireccional**, el cual tiene un **extremo de escritura**[a], y un **extremo de lectura**[b]. Genera dos descriptores, representando a los extremos de lectura y escritura respectivamente, y los guarda en **descriptores**.
- `int dup2(int oldfd, int newfd)`: si `oldfd` y `newfd` son distintos, primero se elimina la referencia al objeto apuntado por `newfd`, y luego se apunta `newfd` al mismo objeto que `oldfd`.

[a] Por donde "entra" la información.
[b] Por donde "sale" la información.

Funciones Útiles (3)

Archivos / Descriptores

- `int open(char* path, int flags, ...)`: Abre el archivo indicado por `path`, retornando un *descriptor* que apunta a dicho archivo.
- `int close(int d)`: Cierra *para el proceso actual* el descriptor `d` pasado por parámetro.
- `int read(int d, void *b, size_t s)`: Lee `s` bytes del archivo apuntado por el descriptor `d`, y los escribe en el buffer `b`.
- `int write(int d, void *b, size_t s)`: Lee `s` bytes del buffer `b`, y los escribe en el archivo apuntado por el descriptor `d`.

Otros

- **printf(char* fmt, ...)**: función **variádica**[a] que toma un string de formato **fmt** y cero o más parámetros adicionales, y escribe el resultado en **stdout**.

[a] Permite un número arbitrario de parámetros.

Funciones Útiles (4)

Ejecutar un archivo

- `int execlp(const char *file, const char *arg, ...)`: Sustituye la imagen de memoria del programa por la del programa ubicado en `file`. El `const char *arg` y la subsiguiente elipsis se pueden pensar como los argumentos `arg0, arg1, ..., argn` del programa a ejecutar. Por convención, `arg0` debería ser el nombre del binario (`file`). La lista de argumentos debe ser terminada con NULL ya que es una función variádica. Ver familia de funciones exec[a].

[a] https://linux.die.net/man/3/execlp

Temario

Ejercicio 1: Los Simonzón

Dados y Doppelgänger...

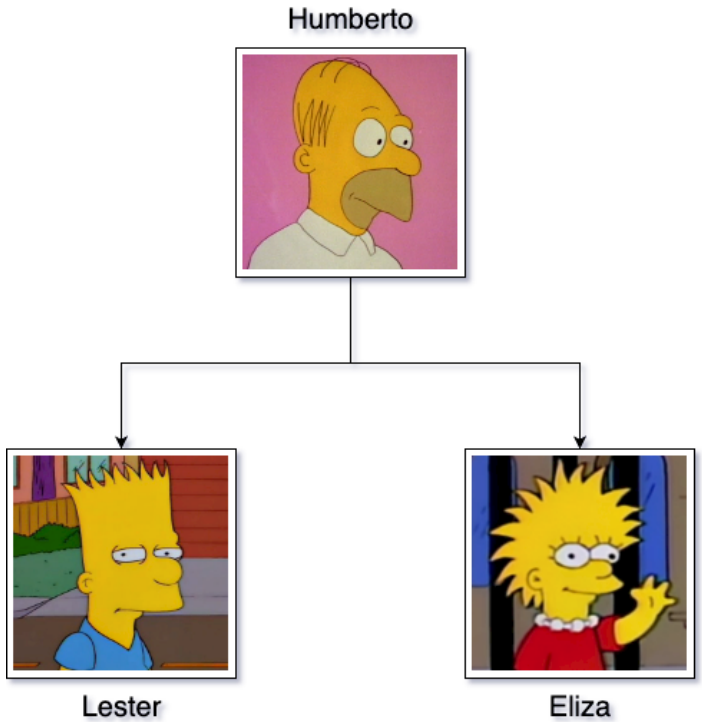
Lester y Eliza están muy aburridos. Para matar el tiempo, su ocurrente padre Humberto les propone un juego:

- Lester y Eliza tiran cada uno un dado, al mismo tiempo.
- Luego le cuentan a Humberto qué valor les dio el dado.
- Finalmente, Humberto se fija cuál es el número más grande y grita bien fuerte el nombre del ganador.

Ejercicio 1: Los Simonzón

Proponer un código en C que represente este juego respetando:

- Cada personaje debe estar representado por un proceso.
- Se debe respetar la genealogía.



- La comunicación entre procesos deberá realizarse con pipes.
- Los anuncios deberán realizarse imprimiendo a `stdout`.

Ejercicio 1: Los Simonzón

Pseudocódigo

Humberto:

Ejercicio 1: Los Simonzón

Pseudocódigo

Humberto:

- Crear a Lester.
- Crear a Eliza.

Lester:

Eliza:

Ejercicio 1: Los Simonzón

Pseudocódigo

Humberto:

- Crear a Lester.
- Crear a Eliza.

Lester:

- Tirar el dado.
- Enviar resultado a Humberto.

Eliza:

Ejercicio 1: Los Simonzón

Pseudocódigo

Humberto:

- Crear a Lester.
- Crear a Eliza.

Lester:

- Tirar el dado.
- Enviar resultado a Humberto.

Eliza:

- Tirar el dado.
- Enviar resultado a Humberto.

Ejercicio 1: Los Simonzón

Pseudocódigo

Humberto:

- Crear a Lester.
- Crear a Eliza.
- Recibir número de Lester.
- Recibir número de Eliza.

Lester:

- Tirar el dado.
- Enviar resultado a Humberto.

Eliza:

- Tirar el dado.
- Enviar resultado a Humberto.

Ejercicio 1: Los Simonzón

Pseudocódigo

Humberto:

- Crear a Lester.
- Crear a Eliza.
- Recibir número de Lester.
- Recibir número de Eliza.
- Anunciar al ganador.

Lester:

- Tirar el dado.
- Enviar resultado a Humberto.

Eliza:

- Tirar el dado.
- Enviar resultado a Humberto.

Ejercicio 1: Los Simonzón

Resolución

Nos tomamos unos minutos para pensarlo...

...y luego lo resolvemos en pizarrón.

Ejercicio 1: Los Simonzón

Solución

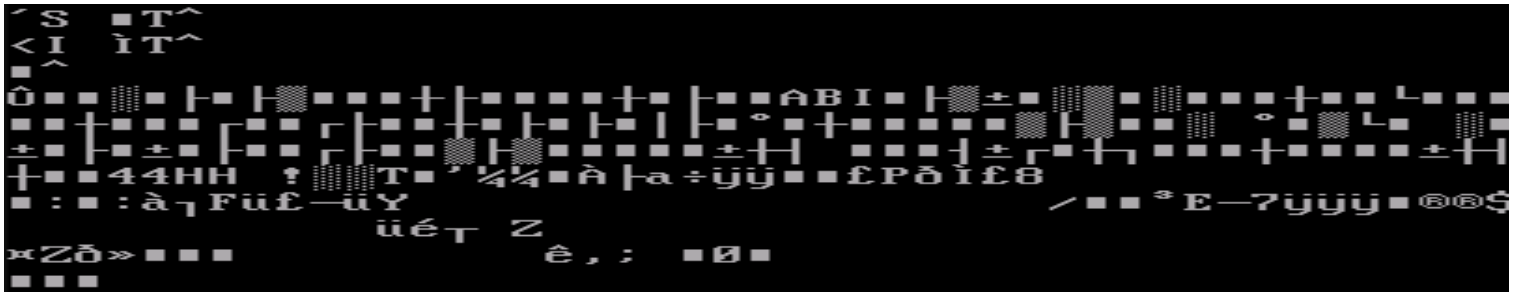
Importante

Todas las soluciones mostradas están simplificadas a efecto de poder incluirlas en la clase. Deberán tener **cuidado** y criterio a la hora de decidir si aplicar estas simplificaciones en sus resoluciones.

Es imprescindible que **lean minuciosamente el manual** para cada función que utilicen.

Temario

Ejercicio 2: Fantasma en el Shell



¡No se que toqué... y ahora no anda más el shell!

Ejercicio 18 de la Práctica 1: "Procesos y APIs del SO"

Escribir el código de un programa que se comporte de la misma manera que la ejecución del comando ``ls -al | wc -l'' en un intérprete de comandos.

No está permitido utilizar la función `system()`, y cada uno de los programas involucrados en la ejecución del comando deberá ejecutarse como un subproceso.

Ejercicio 2: Fantasma en el Shell

Consideraciones preliminares

Primero, veamos qué hacen los comandos:

`ls -al`	
`wc -l`	
`cmd1 | cmd2`	
`ls -al | wc -l`	

Ejercicio 2: Fantasma en el Shell

Consideraciones preliminares

Primero, veamos qué hacen los comandos:

`ls -al`	Lista los archivos del directorio actual.
`wc -l`	
`cmd1 | cmd2`	
`ls -al | wc -l`	

Ejercicio 2: Fantasma en el Shell

Consideraciones preliminares

Primero, veamos qué hacen los comandos:

`ls -al`	Lista los archivos del directorio actual.
`wc -l`	Cuenta la cantidad de líneas del input.
`cmd1 | cmd2`	
`ls -al | wc -l`	

Ejercicio 2: Fantasma en el Shell

Consideraciones preliminares

Primero, veamos qué hacen los comandos:

`ls -al`	Lista los archivos del directorio actual.
`wc -l`	Cuenta la cantidad de líneas del input.
`cmd1 | cmd2`	Ejecuta `cmd1`, y usa su output como input para ejecutar `cmd2`.
`ls -al | wc -l`	

Ejercicio 2: Fantasma en el Shell

Consideraciones preliminares

Primero, veamos qué hacen los comandos:

`ls -al`	Lista los archivos del directorio actual.
`wc -l`	Cuenta la cantidad de líneas del input.
`cmd1 | cmd2`	Ejecuta `cmd1`, y usa su output como input para ejecutar `cmd2`.
`ls -al | wc -l`	El comando completo cuenta la cantidad de archivos en el directorio actual.

Ejercicio 2: Fantasma en el Shell

Pseudocódigo

Pasos a seguir...

- Creamos el pipe.
- Creamos los subprocesos.
- Queremos que todo lo que el subproceso 1 **escriba a stdout**, el subproceso 2 lo **lea desde stdin**. Para ello, usando dup2:
  - Conectamos **el descriptor de stdout** del subproceso 1 al **extremo de escritura** del pipe.
  - Conectamos **el descriptor de stdin** del subproceso 2 al **extremo de lectura** del pipe.
- Ejecutamos el comando `ls -al` en el subproceso 1.
- Ejecutamos el comando `wc -l` en el subproceso 2.

Ejercicio 2: Fantasma en el Shell

Detectando el final del input

Todavía falta algo...

El subproceso 2 tiene que poder **detectar cuando el subproceso 1 terminó de escribir**. ¿Por qué en este caso es necesario esto?

Ejercicio 2: Fantasma en el Shell

Detectando el final del input

Todavía falta algo...

El subproceso 2 tiene que poder **detectar cuando el subproceso 1 terminó de escribir**. ¿Por qué en este caso es necesario esto?

Y, más importante, ¿cómo lo logramos?

Ejercicio 2: Fantasma en el Shell

Detectando el final del input

Todavía falta algo...

El subproceso 2 tiene que poder **detectar cuando el subproceso 1 terminó de escribir**. ¿Por qué en este caso es necesario esto?

Y, más importante, ¿cómo lo logramos?

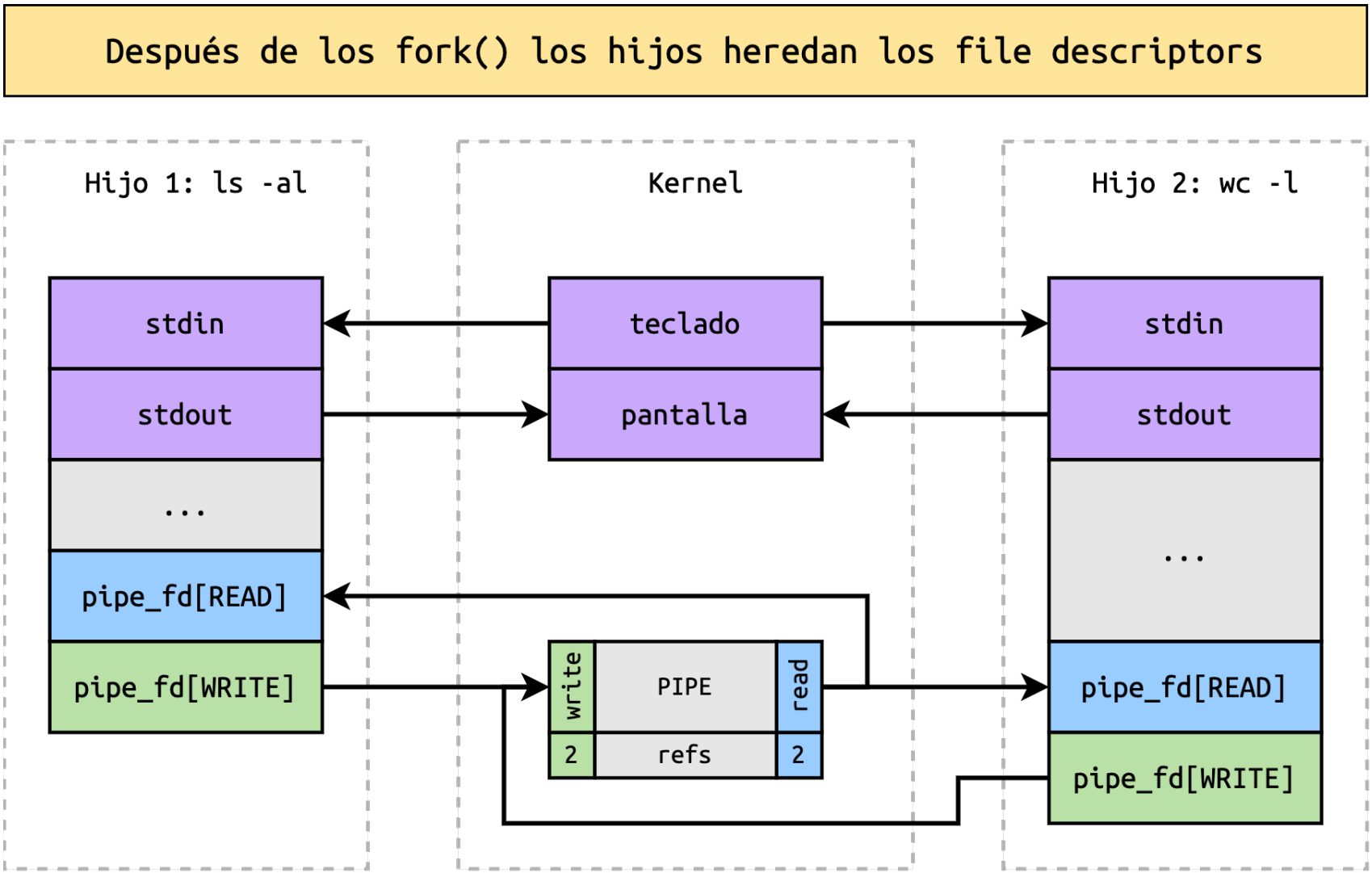
¡Haciendo un buen uso de los pipes!

Debemos cerrar los descriptores no utilizados en cada subproceso.

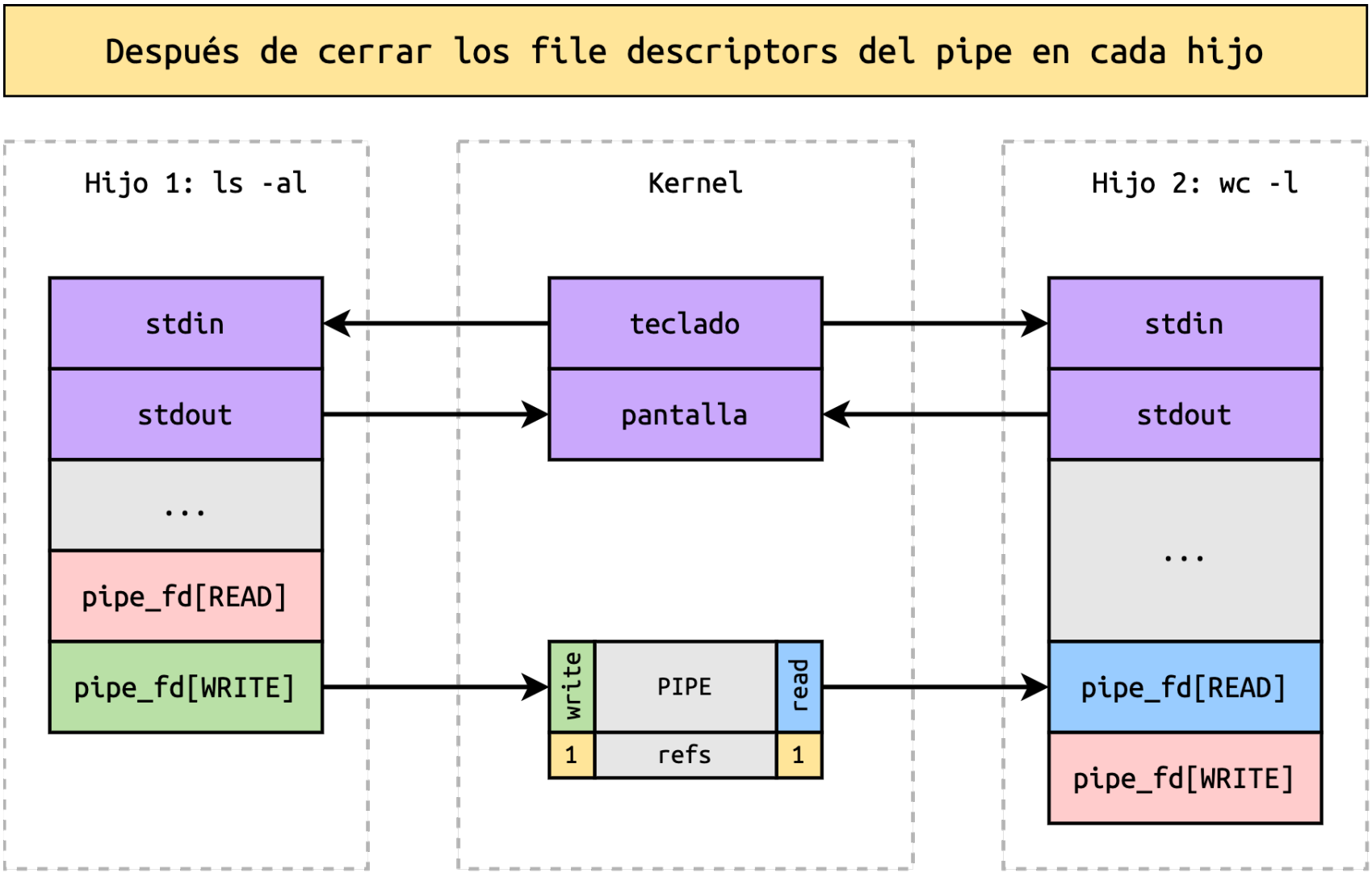
- El subproceso 1 jamás va a usar el extremo de lectura del pipe.
- El subproceso 2 jamás va a usar el extremo de escritura.

Cerrándolos previamente, nos aseguramos de que cuando el subproceso 1 termine de escribir a *"su stdout"* (que en realidad es el extremo de escritura del pipe), el sistema operativo detecte que **ya no existen más referencias hacia ese descriptor**, retornando un valor de **End-Of-File** cuando el subproceso 2 intente leer *"su stdin"* (que en realidad es el extremo de lectura del pipe).

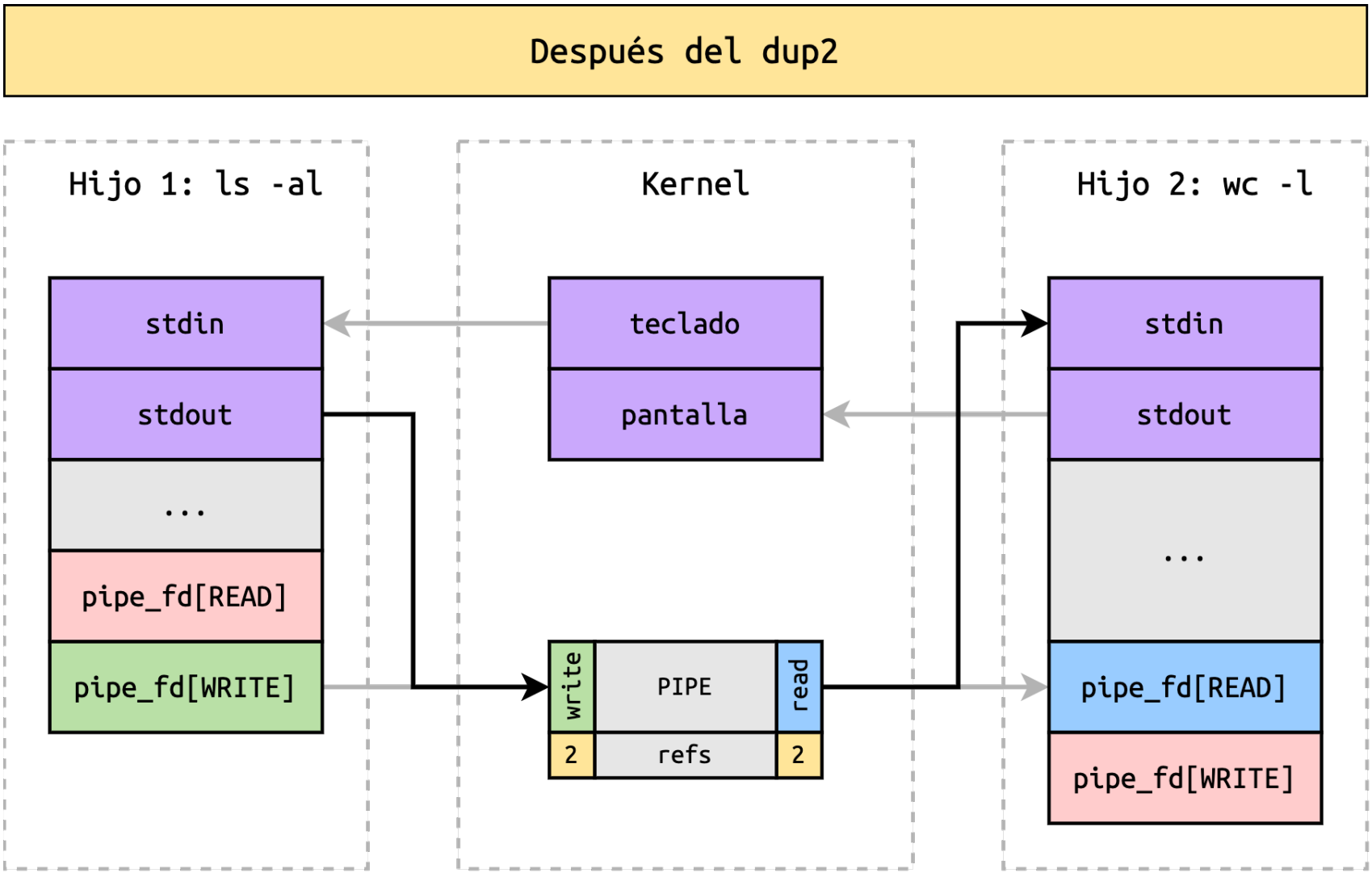
Ejercicio 2: Fantasma en el Shell



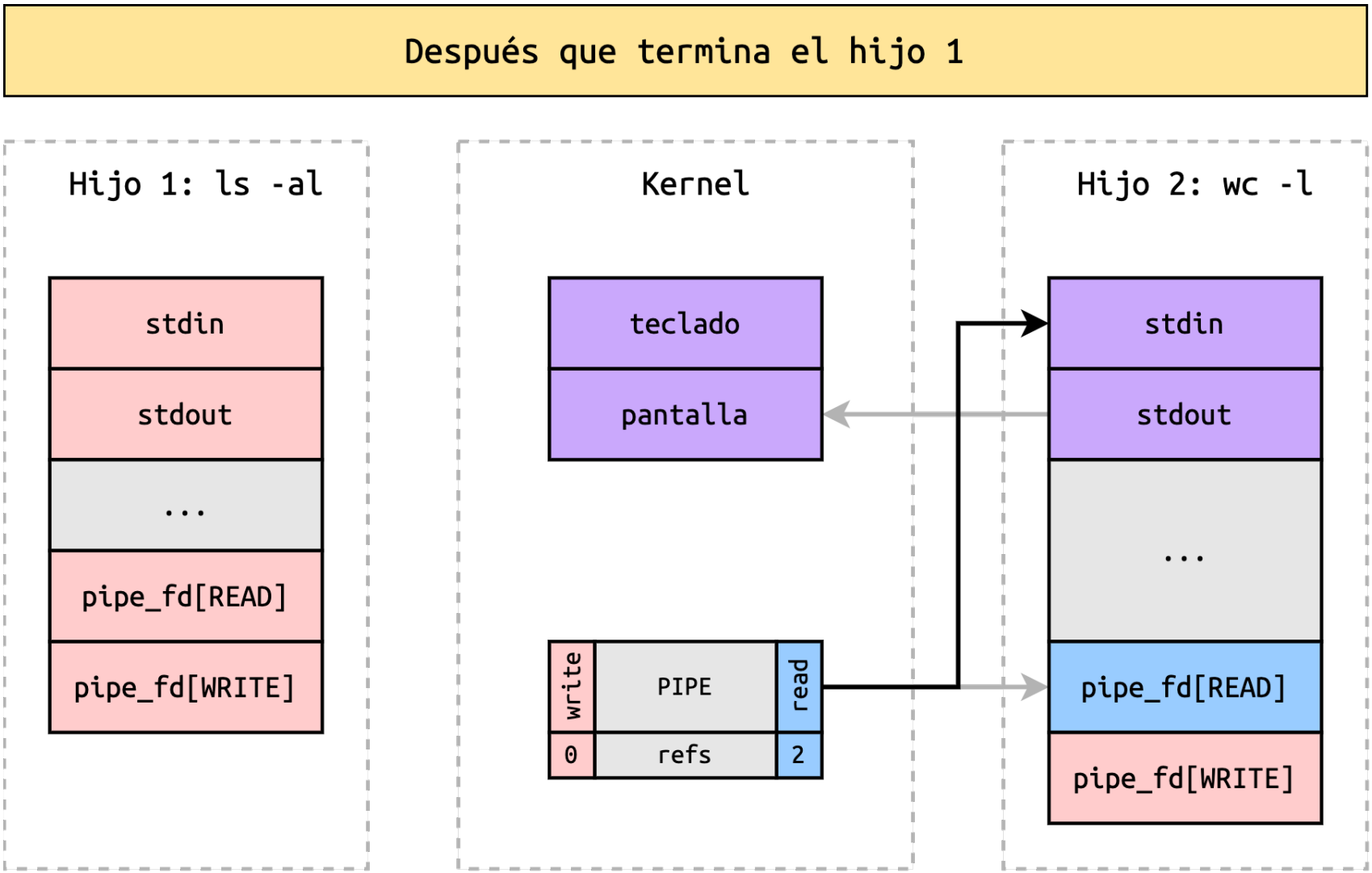
Ejercicio 2: Fantasma en el Shell



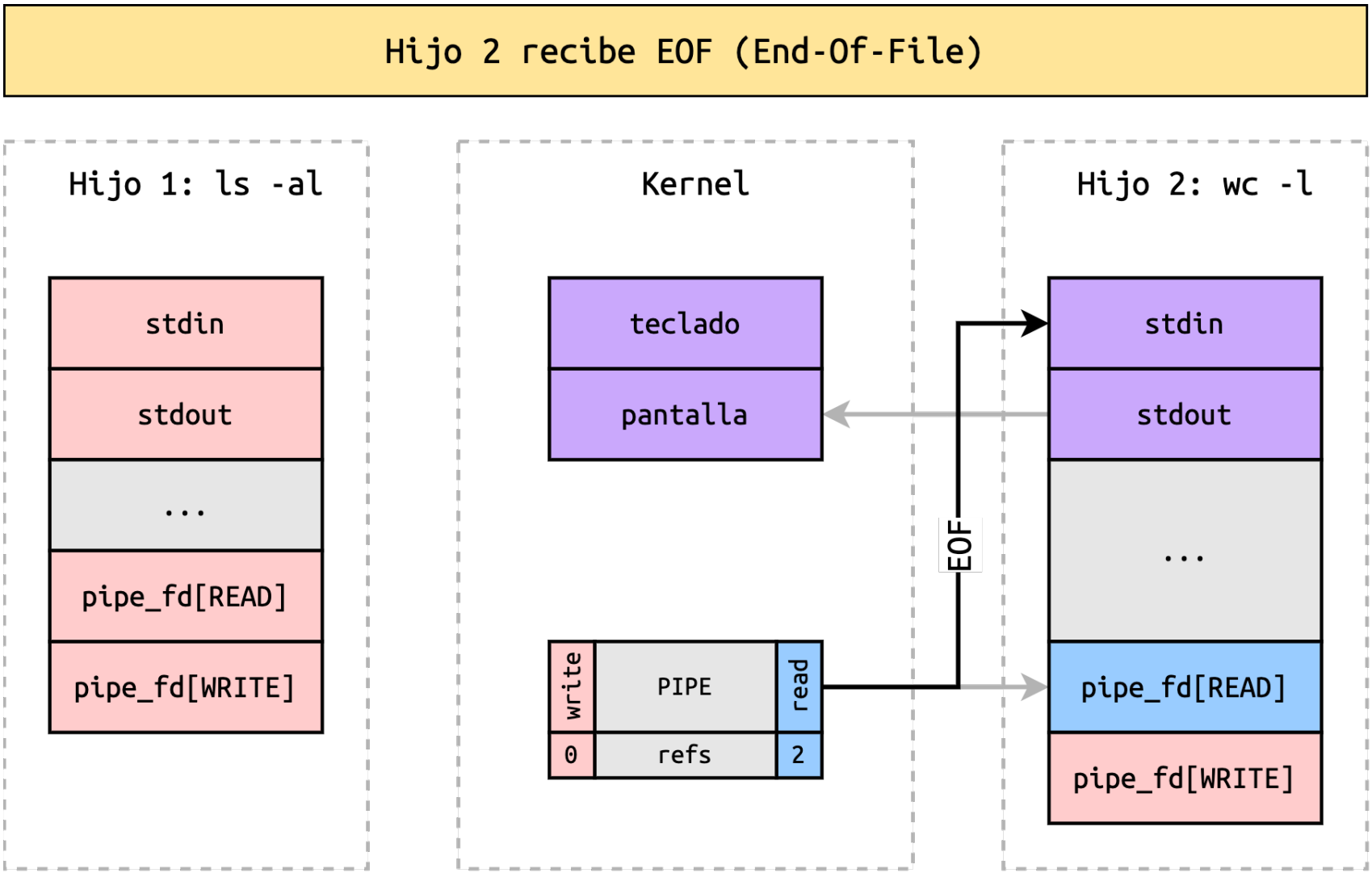
Ejercicio 2: Fantasma en el Shell



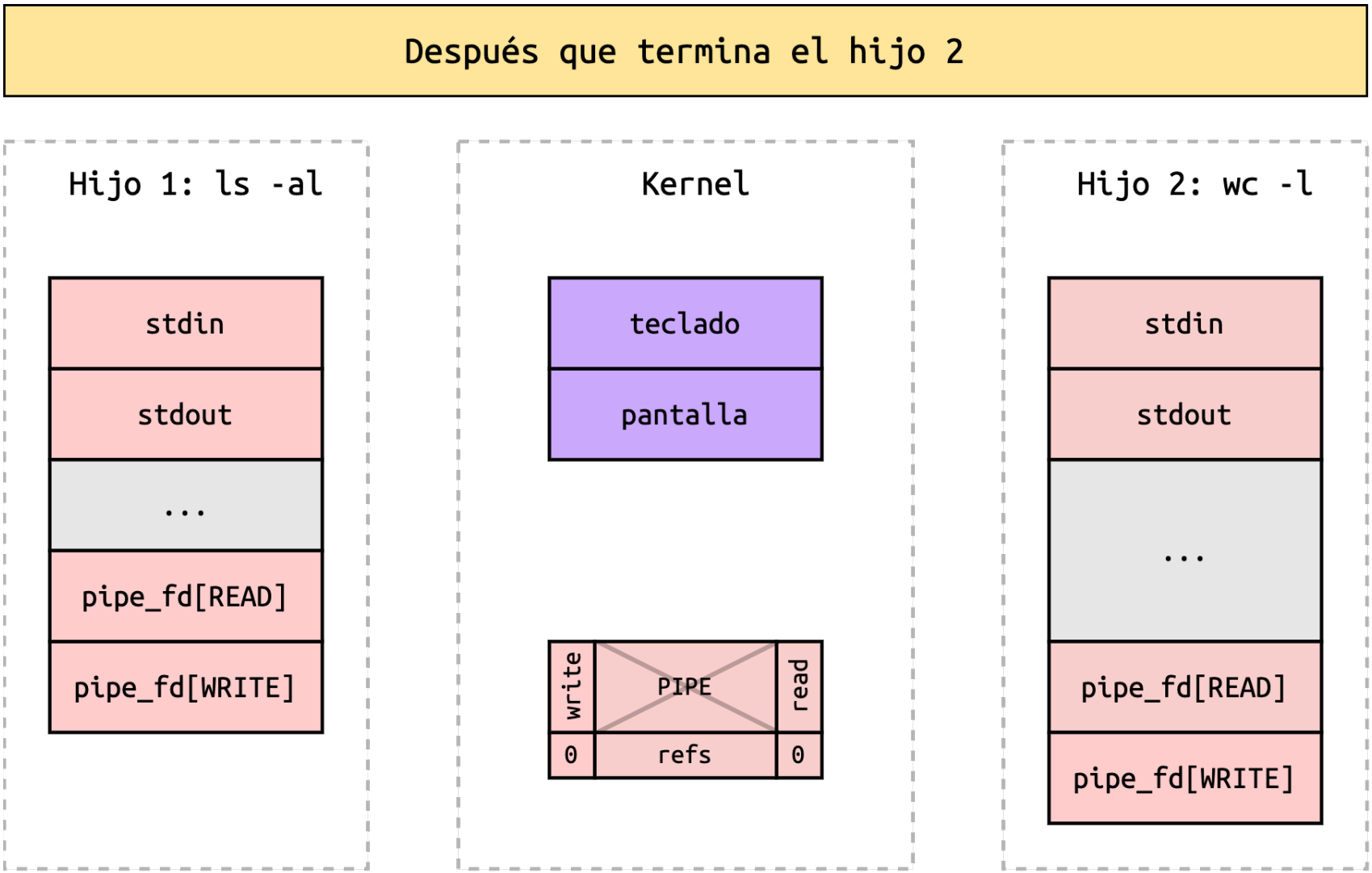
Ejercicio 2: Fantasma en el Shell



Ejercicio 2: Fantasma en el Shell



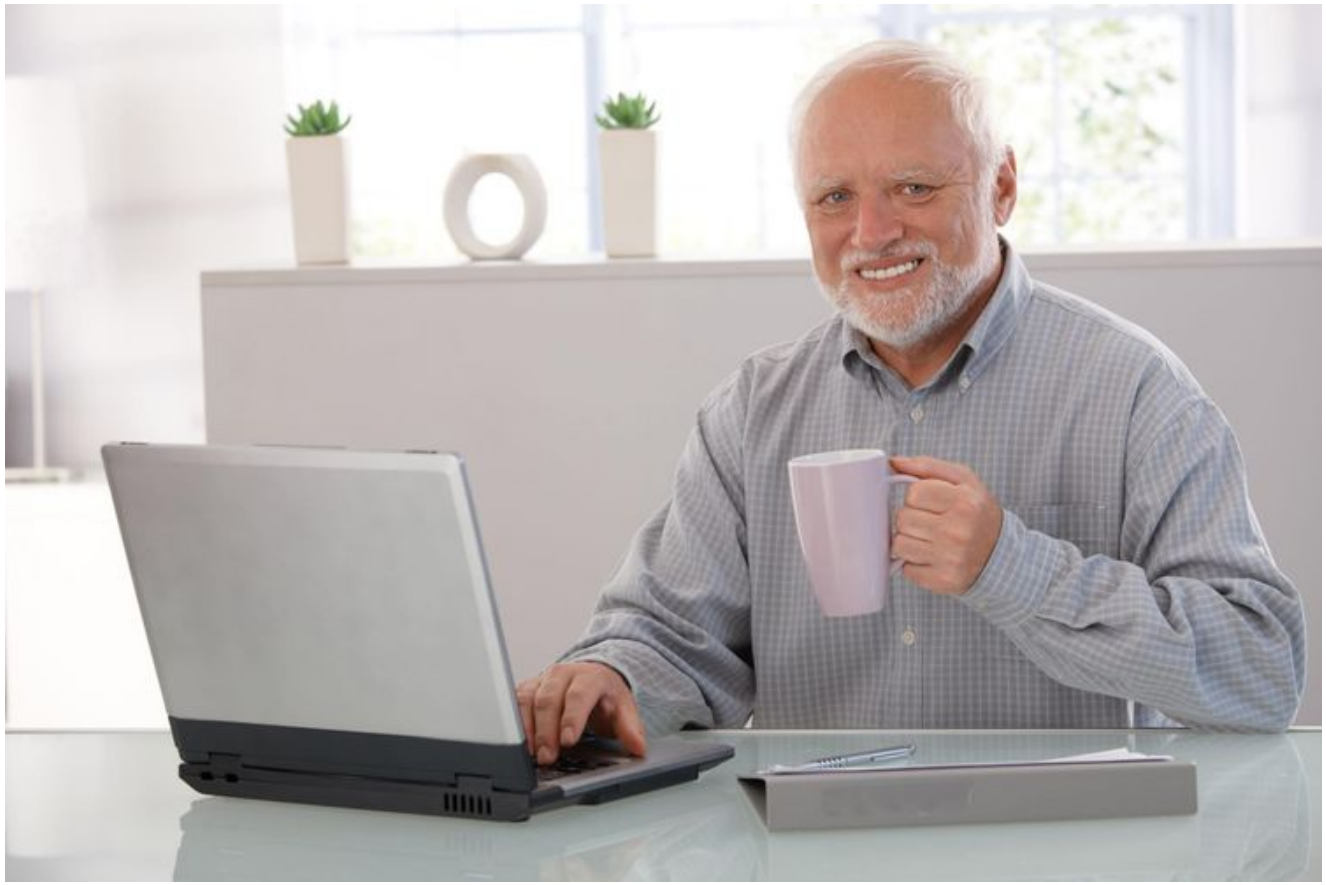
Ejercicio 2: Fantasma en el Shell



Ejercicio 2: Fantasma en el Shell

Resolución

Nos tomamos unos minutos para pensarlo...



...y luego lo resolvemos en pizarrón.

Bonus: Para pensar...

¿Qué tendríamos que **modificar al código** para lograr lo siguiente?

- Encadenar un tercer comando.
  - Por ejemplo: `ls -al | grep e | wc -l`
- Que el output se guarde en un archivo.
  - Por ejemplo: `ls -al | wc -l > file.txt`

Temario

Ejercicio 3: ¿Cien cabezas piensan mejor que una?

Consigna

Escriba un programa que cuente los números pares en el rango de 2 a mil millones, utilizando para ello una función `bool esPar(long numero)` provista por la cátedra.
El programa dividirá el cálculo de forma equitativa entre varios procesos ejecutándose en paralelo. El número de procesos se especificará como parámetro en la línea de comandos.

Ejercicio 3: ¿Cien cabezas piensan mejor que una?

Consigna (2)

- El proceso primario creará el número especificado de procesos, y luego dividirá el rango de números en subrangos iguales y asignará un subrango a cada proceso secundario.
- Por ejemplo, si la cantidad de procesos p_i es 10, cada subrango tendrá aproximadamente cien millones de números.
  - p_1 calculará en el rango [2, 100.000.002).
  - p_2 calculará en el rango [100.000.002, 200.000.001).
  - ...
  - p_{10} calculará en el rango [900.000.002, 1.000.000.001).
- El proceso primario informará a cada proceso secundario el subrango que le corresponde usando pipes. Luego, los procesos secundarios deberán enviar sus recuentos al primario, también utilizando pipes.
- Finalmente, el primario deberá sumar todos los recuentos e imprimir el resultado total.

Ejercicio 3: ¿Cien cabezas piensan mejor que una?

Pseudocódigo

Proceso primario

- Crear los pipes
- Crear los procesos secundarios
- Calcular e informar los rangos
- Recibir los resultados parciales
- Sumarlos e imprimir el total

Procesos secundarios

- Cerrar pipes
- Recibir los rangos
- Contar los pares
- Informar el resultado

Momento para preguntas

Temario

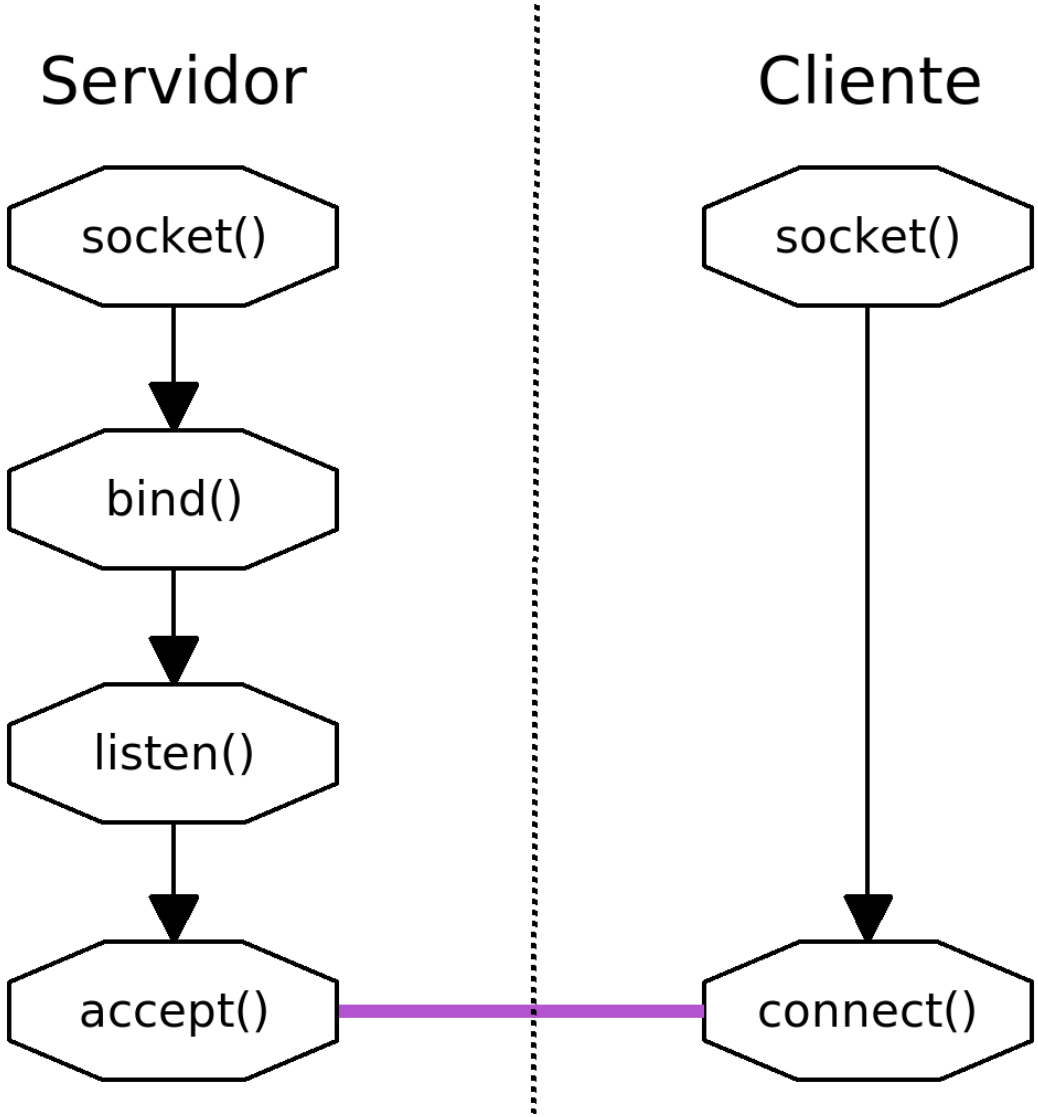
¿Qué pasa si los procesos están en máquinas distintas?

¿Qué pasa si los procesos están en máquinas distintas?

Sockets

- Un socket es el extremo de una conexión, constituida por una dirección ip y un puerto.
- Dos procesos que se quieren comunicar entre sí se deben poner de acuerdo usando los sockets correspondientes.
- Los sockets pueden trabajar en distintas máquinas.
- Sólo se deben conocer de antemano las direccciones ip y los puertos utilizados para la conexión.
- Normalmente utilizan el paradigma cliente-servidor.

Comunicación Cliente-Servidor usando `sockets`



¿Tipos de Sockets?

Sockets

- Cuando un socket es creado, el programa tiene que especificar el **dominio de dirección** y el **tipo de socket**.

¿Tipos de Sockets?

Sockets

- Cuando un socket es creado, el programa tiene que especificar el **dominio de dirección** y el **tipo de socket**.
- Hay muchos **dominios de dirección**, pero hay dos mas usados:
  - Unix Domain: Procesos en una maquina local (con lo que vamos a trabajar).
  - Internet Domain: Procesos a traves de Internet.

¿Tipos de Sockets?

Sockets

- Cuando un socket es creado, el programa tiene que especificar el **dominio de dirección** y el **tipo de socket**.
- Hay muchos **dominios de dirección**, pero hay dos mas usados:
  - Unix Domain: Procesos en una maquina local (con lo que vamos a trabajar).
  - Internet Domain: Procesos a traves de Internet.
- Hay dos tipos de sockets:
  - Stream Sockets: Tratan la comunicacion como un stream continuo de caracteres.
  - Datagram sockets: Tienen que leer el mensaje entero de una vez.

¡Importante! Dos procesos pueden interactuar entre sí sólo si son sockets del mismo tipo.

Estableciendo `sockets`: Servidor

- `int socket(int domain, int type, int protocol);`
  Crea un nuevo socket. Vamos a usar la constante `AF_UNIX` y el tipo `SOCK_STREAM`. El tercer parámetro suele ser 0, indicando al SO que use la configuración predeterminada de protocolos.

Estableciendo sockets: Servidor

- `int bind(int fd, sockaddr* a, socklen_t len);`
  Asigna una dirección (nombre o IP y puerto) al socket.

```
struct sockaddr_in {
    short   sin_family; /* must be AF_INET */
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8]; /* Not used, must be zero */
}
```

Estableciendo `sockets`: Servidor

1. `int listen(int fd, int backlog);`
   Setea al socket del servidor como un socket pasivo que recibirá conexiones entrantes. Se maneja una cola para poder recibir varias conexiones entrantes.

2. `int accept(int fd, sockaddr* a, socklen_t* len);`
   Extrae de la cola una solicitud de conexión y establece la comunicación entre los sockets. Se bloquea en caso de no existir conexiones pendientes. Devuelve un nuevo fd para conexión.

Estableciendo sockets: Cliente

1. `int socket(int domain, int type, int protocol);`
   Crear un nuevo socket.

2. `int connect(int fd, sockaddr* a, socklen_t* len);`
   Conectarse a un socket remoto que debe estar escuchando.

Comunicación vía sockets: Mensajes

Una vez que un cliente solicita conexión y ésta es aceptada por el servidor, se puede comenzar el intercambio de mensajes con:

```
ssize_t send(int s, void *buf, size_t len, int flags);
ssize_t recv(int s, void *buf, size_t len, int flags);
```

Nota

- Por defecto, el envío de mensajes es asincrónico (`send` se comporta similar al `write` en pipes).
- A su vez, la recepción con `recv` se comporta similar al `read` en pipes. Más detalles en `man 2 send/recv`.

Veamos un ejemplo en código

¿Cómo hacemos si un servidor se quiere comunicar con dos clientes a la vez?

Veamos...

Comunicación vía `sockets`: ¿Bloqueante?

Si usamos llamadas bloqueantes podemos esperar indefinidamente por el primer cliente mientras el segundo tiene mucho que enviarnos y nunca se lo pedimos.

Soluciones:

1. Llamadas no bloqueantes.
2. Usar las llamadas al sistema `pselect(2)` o `poll(2)`.
3. Utilizar un proceso (o thread) para atender cada cliente.
   (No la veremos en esta clase.)

Solución 1: Llamadas No Bloqueantes

Demo: Llamadas no bloqueantes.

- Se usa la llamada `fcntl(2)` para indicar que los sockets del servidor sean no bloqueantes.
- Al intentar recibir mensajes, si se devuelve `-1` tenemos que ver si es por un error verdadero o porque no había nada en espera de ser recibido. Usamos la variable `errno` para discernir esto.

Este enfoque tiene un problema: al realizar una "espera activa", ocupamos el procesador innecesariamente.

¡Necesitamos algo mejor!

The Ultimate Client-Server Implementation (sin `Threads`)

Demo: llamadas bloqueantes y `pselect(2)` o `poll(2)`.

- La llamada al sistema `pselect(2)` toma como parámetro un conjunto de file descriptors sobre los cuales se quiere esperar y un timeout.
- Se puede separar los fds según si se espera por lectura, escritura, etc.
- Vencido el plazo retorna el control y avisa que no hubo eventos.
- Si hay algún evento, nos retorna el fd correspondiente.

De esta forma evitamos la espera activa y logramos el mismo efecto que las llamadas no bloqueantes sin morfar el CPU.

Ejercicio

Generalizar el caso del server pselect para que acepte múltiples conexiones.

Los clientes sólo pueden enviar mensajes si recibieron un mensaje de bienvenida del servidor luego de conectarse.

Bibliografía

- Beej's Guide to Unix Interprocess Communication. Disponible aquí.
- Beej's Guide to Network Programming: Using Internet Sockets. Disponible aquí.
- Unix socket programming in C. Disponible aquí.

Momento para preguntas

Temario

Resumen

Hoy vimos...

- Cómo crear una **topología** de procesos.
- Cómo usar **pipes** para comunicar procesos.
- Cómo usar **dup2** para duplicar un pipe.
- Por qué es importante **usar correctamente** los pipes.
- Para qué sirve la señal de **End-Of-File**.
- Cómo establecer una comunicación bidireccional:
  - Usando un solo pipe por cada par de procesos[a]
  - Usando dos pipes por cada par de procesos.

[a] **¡muy peligroso!**

Resumen

Hoy vimos...

- Cómo crear una **topología** de procesos.
- Cómo usar **pipes** para comunicar procesos.
- Cómo usar **dup2** para duplicar un pipe.
- Por qué es importante **usar correctamente** los pipes.
- Para qué sirve la señal de **End-Of-File**.
- Cómo establecer una comunicación bidireccional:
  - Usando un solo pipe por cada par de procesos[a]
  - Usando dos pipes por cada par de procesos.

[a] **¡muy peligroso!**

Cómo seguimos...

- Pueden hacer toda la guía 1.
- Próximamente: taller de IPC.