

Comunicación entre Procesos (IPC)

Rodolfo Baader¹


¹Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, primer cuatrimestre de 2023

(2) La clase anterior...

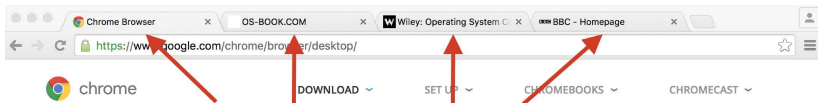
- Vimos
 - El concepto de proceso en detalle.
 - Sus diferentes actividades.
 - Qué es una system call.

(3) La de hoy

- Vamos a ver comunicaciones entre procesos.
- Ya sea entre procesos en un mismo equipo, o entre procesos remotos.
- Sirve para:
 - Compartir información.
 - Mejorar la velocidad de procesamiento.
 - Modularizar.
- La comunicación entre procesos suele llamarse *IPC*, por *InterProcess Communication*. 

(4) Arquitectura multiproceso en Chrome

- Antes, los browsers corrían como un solo proceso (algunos aún lo hacen)
 - Si hay problemas con un sitio, todo el browser se puede colgar.
- Chrome crea 3 tipos de procesos:
 - Browser: administra interface de usuario, acceso a disco y a red.
 - Renderer: muestra las páginas, se ocupa del html y javascript. Uno nuevo por sitio. Corre en sandbox.
 - Plug-in: Proceso para cada tipo de plugin.

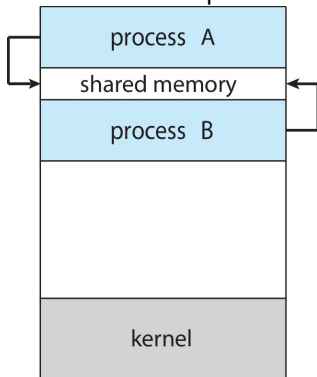


Each tab represents a separate process.

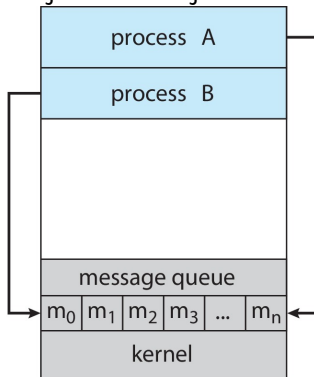
- Hay varias formas de IPC:
 - Memoria compartida.
 - Algún otro recurso compartido (por ejemplo, archivo, base de datos, etc.).
 - Pasaje de mensajes, ya sea entre procesos de la misma máquina, o de equipos conectados en red.

(6) IPC

- a. Memoria compartida b. Pasaje de mensajes



(a)



(b)

(7) Pipes

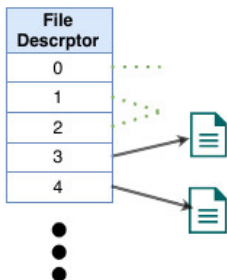
- Un “pseudo archivo” que “esconde” una forma de IPC.
- Ordinary pipes:
 - `ls -l | grep so`
- Named pipes:
 - `mkfifo -m 0640 /tmp/mituberia`

Antes de hablar de pipes es necesario que se tenga una idea de qué es un *File Descriptor*.

- Intuitivamente, representan instancias de archivos abiertos.
- Concretamente, son índices de una tabla que indica los archivos abiertos por el proceso.

FILE DESCRIPTORS

- Cada proceso en UNIX viene con su propia tabla (en su PCB) al momento de ser creado.
- Son usados por el Kernel para referenciar a los archivos abiertos que tiene cada proceso. Cada entrada de la tabla apunta a un archivo.



FILE DESCRIPTORS

- ¿Qué pasa si hago **open** de un archivo? ¿Y si abro el mismo más de una vez?
- ¿Que pase si hago **close** de un archivo?

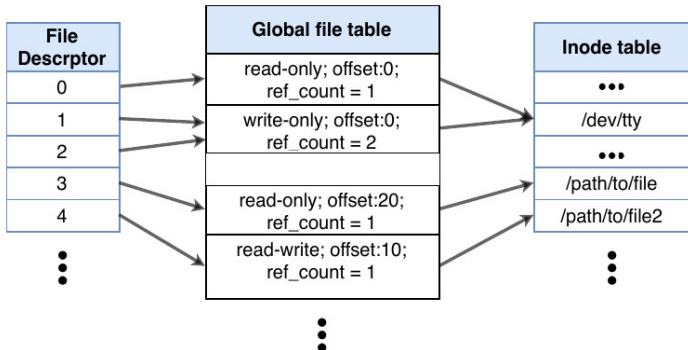
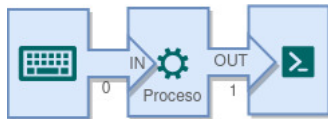


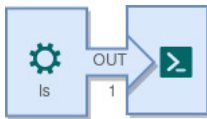
Figura: Un **esquema** más general que puede ayudarnos a entender mejor. Tener en cuenta que la tabla de file descriptor es **por proceso** y que solo se muestra parte de la información que contiene la Global File Table y la de inodos

Modelando el flujo de comunicación

¿Cómo se suele comunicar un proceso en una terminal con una persona?



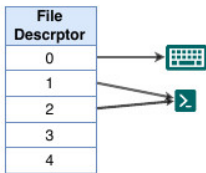
Podemos pensar otros esquemas, como por ejemplo:



Importante: En Unix, el teclado y la pantalla se modelan como un archivo

Modelando el flujo de comunicación

- La mayoría de los procesos esperan tener abiertos 3 *file descriptors* (las entradas 0, 1 y 2 de la tabla) correspondientes a:
0 = standard input, *1 = standard output* y *2 = standard error*.
- Típicamente, al lanzar un proceso desde consola puedo encontrar algo así:



- Una nota importante es que estos *file descriptors* se heredan de un proceso padre a un proceso hijo al usar la llamada a `fork()`, y se mantiene en la llamada a `execve`.

¿Cómo escribo a un archivo?

Si tenemos un *file descriptor*, podemos leer/escribir con:

```
ssize_t read(int fd, void *buf, size_t count);  
ssize_t write(int fd, const void *buf, size_t count);
```

- *fd* *file descriptor*.
- *buf* puntero al *buffer* donde se almacenan los datos a escribir o leer.
- *count* cantidad máxima de bytes a escribir o leer.

Estas funciones devuelven la cantidad de bytes leídos, -1 en caso de error.

Nota

El comportamiento de `read` y `write` dependen del *tipo* de *file descriptor*. Más sobre esto en breve

¿Para qué quiero saber esto? ¿Y los pipes!?

Un ejemplito y después vamos a pipes.

¿Cómo se imaginan que hace la consola para resolver algo así?

```
echo "Faltan 1196 días para el mundial" > archivo.txt
```

- Se llama a `echo`, un programa que escribe su parámetro por *stdout*.
- Con `>` se le indica a la consola que el *stdout* se redirija a un `archivo.txt`.
- ¿Cómo? Abre el `archivo.txt` y hace que la entrada de la salida estándar (*stdout*) en la tabla de File descriptors abiertos, apunte a él.
- ¿Cómo? La función `int dup2(int oldfd, int newfd)` pisa en el file descriptor `newfd` el contenido que está en `oldfd` (más info: Ver `man dup2`)

Esquema de redirección con dup2

```
echo "Faltan 1196 días para el mundial" > archivo.txt
```

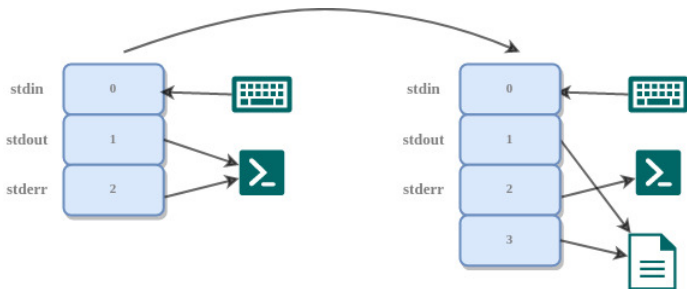


Figura: Tablas de File descriptors del proceso echo antes y después de efectuarse la redirección con `>`

Ahora sí, PIPES

Los *pipes* se escriben en consola con el caracter “|”.

Por ejemplo, qué sucede si escribimos en *bash*:

```
echo "sistemas" | wc -c
```

- Se llama a `echo`, que escribe por *stdout*.
- Se llama a `wc -c`, que cuenta cuántos caracteres entran por *stdin*.
- Se conecta el *stdout* de `echo` con el *stdin* de `wc -c`.

Un esquema de pipes para el ejemplo

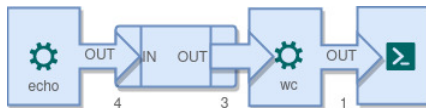


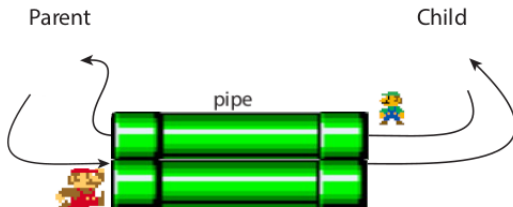
Figura: `echo "sistemas" | wc -c`

Todo esto parece muy mágico, ¿Qué hay detrás de un PIPE?

Un pipe se representa como un **archivo temporal y anónimo**¹ que se aloja en memoria y actúa como un **buffer** para leer y escribir de manera **secuencial**.

Nota

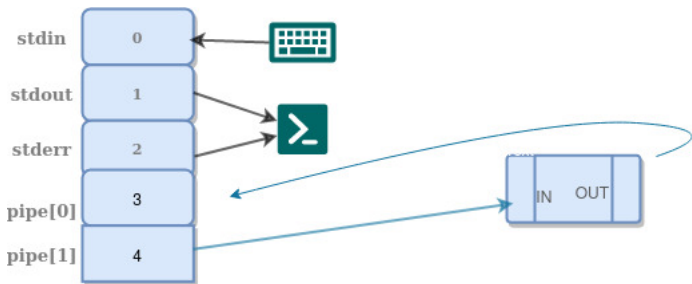
Los pipes son un canal que debe ser interpretado como un byte stream. No hay noción de separación por mensajes.



¹Si bien el uso más frecuente son los pipes anónimos, también se usan los pipes nombrados o con nombre - mkfifo

¿Pera pera, dijiste que era un archivo, no?

Los pipes son archivos, cuando se crean se agregan sus extremos a la tabla de *FILE DESCRIPTORS*.



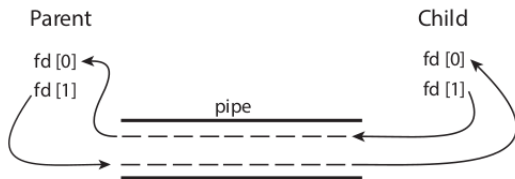
Comunicación vía pipes

Se crea mediante la syscall:

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

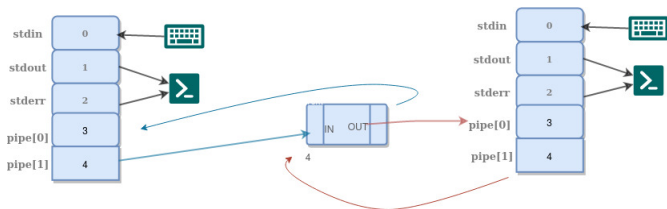
Luego de ejecutar pipe, tenemos:

- En `pipefd[0]`, un *file descriptor* que apunta al extremo del pipe en el cual se **lee**.
- En `pipefd[1]`, otro *file descriptor* que apunta al extremo del pipe en el cual se **escribe**.



Y si hago fork como funciona?

Podemos ver que los *file descriptors* del padre se copian al hijo, y siguen apuntando a los mismo extremos del pipe.



(22) Dónde estamos

- Vimos
 - Distintas formas de IPC.
 - Detalles del funcionamiento de pipes
- En la próxima teórica:
 - Vemos scheduling de procesos