Resumen IPC(Inter-Process Communication)

Tomás Felipe Melli

Noviembre 2024

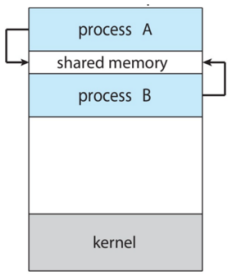
Índice

1 Introducción

La comunicación entre procesos consta de varios *mecanismos* que permiten a los procesos que están en el mismo equipo o remotamente alojados, comunicarse entre sí. Esto facilita compartir información, mejorar la velocidad de procesamiento y modularizar.

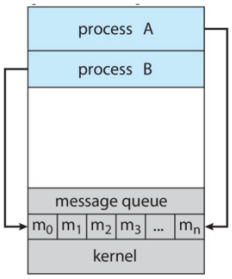
2 Memoria Compartida

Los proceso pueden hacer uso de un espacio compartido de la memoria o base de datos para interactuar entre sí:



3 Pasaje de Mensajes

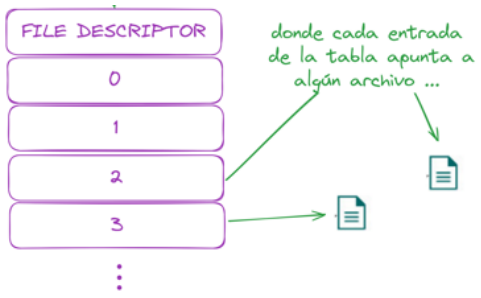
Los procesos se puede enviar mensajes dentro de la misma máquina o en equipos conectados por red:



4 PIPES

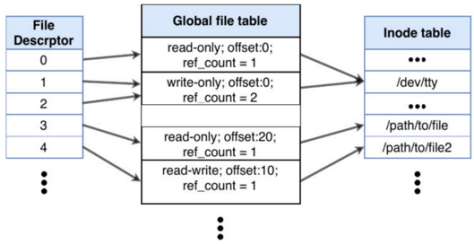
Existe un mecanismo llamado **pipe** que en esencia es un **pseudo-archivo** que esconde un medio de comunicación. Hay una serie de conceptos a enteder antes de hablar de pipes.

- Un **File Descriptor** representa la **instancia de un archivo abierto**. En la implementación son **índices** en una tabla *File Descriptor Table*, que indican **qué archivos están abiertos, por proceso**.



Cada proceso tiene en su PCB la tabla y el KERNEL hace uso de esta para referenciar los archivos abiertos que tiene cada proceso.

- Existe análogamente, una tabla que tiene **todos los archivos abiertos en el sistema** llamada **Global File Table** :



- El flujo de la comunicación se puede modelar como una comunicación entre una persona y un proceso en una terminal. En sistemas UNIX, la mayoría de los procesos *esperan tener 3 File Descriptors*. Las entradas 0,1,2 definidas en la tabla :
  - **0 : STDIN** modela el **teclado** o la entrada standard
  - **1 : STDOUT** modela la **pantalla** o la salida standard
  - **2 : STDERR** modela la **pantalla** o la salida de error standard

  Es importante destacar que **los procesos heredan los file descriptors de su padre**.
  Qué operaciones podemos realizar con ellos ?
  Podemos **escribir** un archivo con

```
ssize_t write(int file_descriptor, const void *buffer, size_t count);
```

  Podemos **leer** un archivo con

```
ssize_t read(int file_descriptor, const void *buffer, size_t count);
```

  Donde el *buffer* es la estructura donde guardamos los datos y *count* la cantidad de **bytes** que vamos a leer
  Podemos definir un **Esquema de Redirección**. La **redirección** es una operación en la que un proceso que tiene como *output* un archivo, por ejemplo **ECHO** que tiene como **STDOUT** la pantalla, en vez de que escriba allí, lo hacemos escribir en cierto archivo llamado out.txt.

```
echo "escribime en un txt" > out.txt
```

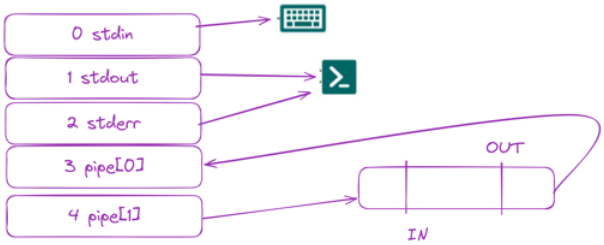
  Y con el símbolo ¿, indicamos la redireccón por consola.
  Con el caracter — podemos usar pipes también, como en el ejemplo :

```
tonius@tonius-XPS-13-9350:~$ echo "Hola" | wc -c
5
```

  En este caso, echo normalmente escribe en pantalla, pero le pasamos a wc -c (word count -character) que recibe hola, y cuenta los caracteres y el de wc es la pantalla el STDOUT.
  Cómo es posible ?
  La función **DUP2(int oldfd, int newfd)** logra pisar el valor anterior del file descriptor (es decir la flechita que apunta al archivo abierto por el proceso definido como STDOUT) y asigna como STDOUT al nuevo.

5 Qué tanto misterio ? Cómo es esto del pipe?

Un pipe se representa como un archivo temporal y *anónimo* que se aloja en memoria y actúa como un buffer para leer y escribir de manera *secuencial*. Son un **canal** (BYTE STREAM) :

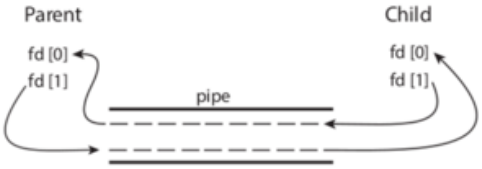


6 Cómo creamos un pipe ?

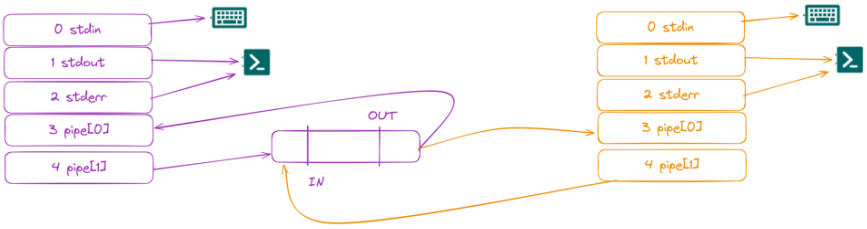
Existe una *syscall*

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

Esto genera **dos extremos**, uno del cual se lee y otro en el cual se escribe :



7 Qué ocurre al realizar un FORK()?



8 Funciones

Algunas ya las mencionamos por arriba, pero profundizaremos.

8.1 PIPE

La función

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

devuelve un entero. Este es el **file descriptor**. Es decir, mediante una syscall, le pide al SO que le genere un número que sea una *abstracción* de la "tupla" [*puntero extremo de lectura, *puntero extremo de escritura] que es el descriptor del pipe. O sea, el sistema se crea un archivo anónimo y temporal que funciona como buffer en el cuál se pueda escribir en un extremo y leer del otro. Por ello, luego se agrega en la lista de descriptores del proceso, este numerito. En otras palabras, este proceso ahora tiene abierto un archivo más en el cuál puede leer y escribir. Después la lógica para que no sea bardo, es otra.

8.2 OPEN

```
int open (char* path, int flags, mode_t modo);
```

Esta función simplemente abre el archivo que le paso en PATH, con los flags adecuados (es decir, si es *read-only, write-only, append* .. etc) y el modo define los permisos cuando se crea (en formato octal).

8.3 DUP2

```
int dup2 (int oldfd, int newfd);
```

Esta función modifica el comportamiento de los file descriptors. Esto es, si normalmente un proceso recibe su input por STDINPUT como el teclado, podríamos decir, en vez de que lo tome de allí, lo tome de otro lugar. Ese lugar, es el archivo anónimo y temporal llamado pipe que será escrito por otro proceso (supongamos) y *tenemos certeza de que así será.* Entonces

```
int dup2 ( file_descriptor, STD_INPUT);
```

Ahora el proceso tomará su info de allí, y si hacemos

```
read (pipe[READ], &lectura, sizeof(int))
```

Leerá lo que el otro proceso le manda.
Y si nadie le escribe ?
Entra en juego el concepto de **END OF FILE**. Primero hay que dar a entender algunas cosas. El sistema operativo, al asignar ese espacio de memoria para que un proceso pueda leer y escribir, se guarda la **cantidad de referencia vigentes** a ese lugar. En otras palabras, el SO sabe cuántos procesos pueden escribir en ese lugar del cual este proceso quiere leer. Como la lectura es **bloqueante**, si nadie puede leer, el proceso quedará **blocked** para siempre. Afortunadamente, si el SO detecta que nadie más tiene una referencia a ese lugar para escribirle, le avisa tipo : "che, nadie te va a escribir! No intentes más."

8.4 CLOSE

```
int close(int d )
```

Cierra el descriptor pasado por parámetro, esto en el SO decrementa el contador de referencias y libera recursos.

8.5 Familia EXEC

Las funciones **EXEC** permiten ejecutar programas

- **execl**

  ```
  int execl(const char* path, const char* arg0, ... , NULL);
  ```

  Abre el archivo en PATH y le pasa varios parámetros, el primero suele ser *el programa.* De antemano conocemos los parámetros, el NULL es para avisar que deje de leer.

- **execle**

  ```
  int execle(const char* path, const char* arg0, ... , NULL, char* const envp[]);
  ```

  Es igual al anterior, pero *le podés pasar variables de entorno.* Naturalmente hereda las del padre, pero se podría querer reemplazar las variables de entorno heredadas y simplemente asignar un arreglo de unas específicas.

- **execv**

  ```
  int execv(const char* path, const char* argv[]);
  ```

  En vez de escribir un argumento atrás de otro, le pasás un array.

- **execve**

```
1         int execve(const char* path, const char* argv[], const char* envp[]);
```

  Lo mismo que antes, pero le podemos modificar las variables de entorno.

- **execvp**

```
1         int execvp(const char* file, const char* arv[]);
```

  Esta función tiene la P de PATH porque ya directamente va a buscar ejecutar el programa desde la ruta definida en la *variable de entorno PATH*

- **execvpe**

```
1         int exevpe(const char* path, const char* arg0, ... , NULL);
```

  Análogo a lo anterior, pero ahora le pasamos el array de variables de entorno.