File Systems

(Parte 2)

Índice general

Filesystems (Clase 2)

*Sistemas de Archivos en *nix*

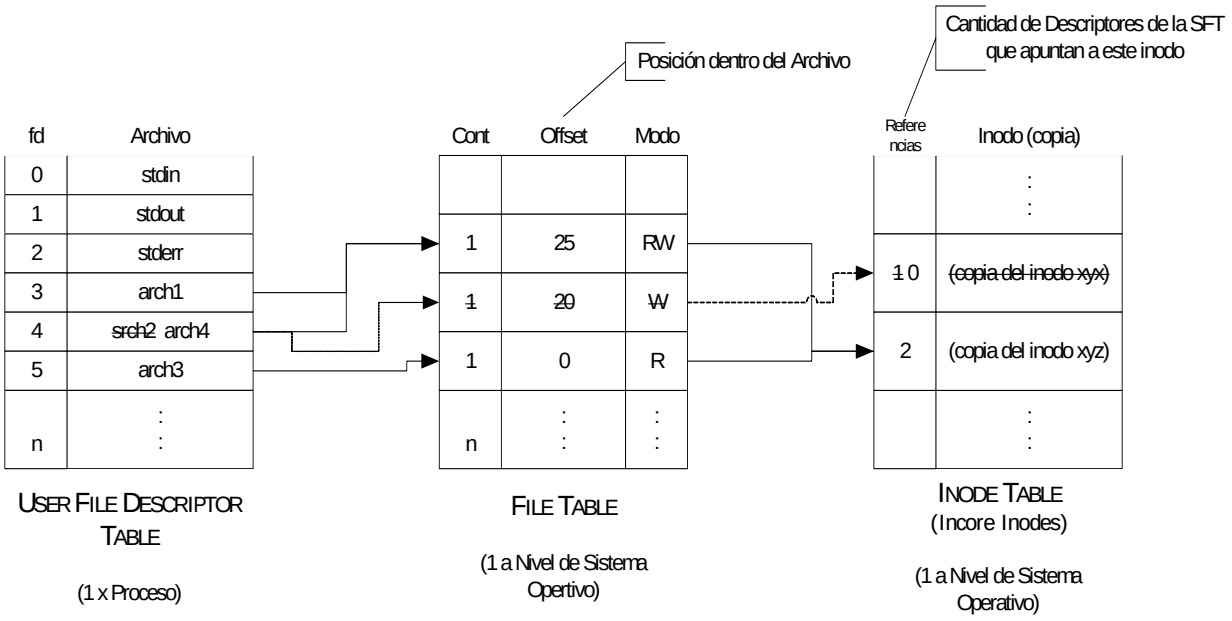
Consideraciones generales

Cuado hablamos de los sistemas de archivos en un entorno *nix, estamos hablando de los sistemas de archivos que se implementaron para sistemas Unix y sus derivados: Linux, Aix, Minix, etc.
Si bien los filesystem de *nix pueden tranquilamente utilizarse en otros Sistemas operativos, es con ellos que se introducen conceptos que difieren mucho de otros filesystem como FAT y es por eso que antes de comenzar a explicar la estructura física se detallarán las estructuras lógicas y de administración de los filesystems en un entorno *nix.

En *nix la estructura de archivos corresponde a un grafo acíclico que comienza en el directorio / (raiz) a partir del cual "cuelgan" el resto de los archivos. Existen varios tipos de archivos distintos como por ejemplo los directorios, dispositivos, y archivos regulares.
Cada componente del sistema es un archivo en la concepción de Unix (puertos, monitor, teclado, discos, etc.), por lo cual cada partición que queramos acceder deberá estar montada en algún punto del grafo.
Para poder utilizar diferentes tipos de sistemas de archivos es que se introduce el concepto de Virtual Filesystem que veremos más adelante.

Explicar File Tables

- ⇨ User File Descriptor Table (UFDT)
- ⇨ System File Table (SFT)
- ⇨ Inode Table (IT)



System Calls

open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char* pathname, int flags[, mode_t modo]);
```
modo sólo vale si se está creando el archivo y equivale a los permisos de usuario que se asignarán al archivo.

Esta función permite convertir un nombre de archivo (*pathname*) es un entero positivo pequeño llamado file descriptor (*fd*).
Si la llamada a la función se completa con éxito, esta devuelve un file descriptor (fd) que será el menor número no asignado en la UFDT.
Para el primer archivo que se abra llamando a esta función, el número de fd asignado será 3, ya que en la tabla se encuentra asignados por defecto los fd 0 a stdin, 1 a stdout y 2 a stderr.

Pseudo código de la System Call open:

```
int open(const char* pathname, int flags[, mode_t modo])
{
convierte pathname a inodo (namei)
if (no existe archivo o acceso denegado)
        return(-1); //devuelve Error
asigna una entrada en la File Table para el inodo, inicializa el contador a 1 y el
  offset a 0 y marca según los flags como R, W o RW;
asigna una entrada en la UFDT y une ese fd con la entrada creada anteriormente
  en la File Table;
if (O_TRUNC)
        libera todos los bloques del archivo (algoritmo free)
unlock(inodo); // libera el inodo bloqueado por namei
return(user file descriptor);
}
```

Por cada llamada a la función open se crea un fd nuevo en la UFDT y una entrada independiente en la File Table.

Los flags O_RDONLY, O_RDWR y O_WRONLY se pueden combinar con los siguientes parámetros utilizando un OR de C (|):

O_CREAT:
: Si el archivo no existe lo crea.

O_TRUNC:
: Trunca el archivo a 0. Sólo si se usa con O_RDWR u O_WRONLY. Si se utiliza con O_RDONLY devuelve error.

O_SYNC:
: Obliga al sistema operativo a utilizar el archivo en forma sincrónica. Cualquier escritura bloquea el proceso hasta que la información se encuentre físicamente grabada en el medio de almacenamiento. *Degrada significativamente el rendimiento del sistema.*

En el caso de estar creando el archivo, los modos de creación posibles son los mismos que en cualquier sistema *nix. Los primero 2 números son 0 y los siguientes se corresponden con permisos del Owner, GroupOwner y OtherUsers respectivamente.

Hay un modo especial de llamar a la función open que equivale a utilizar otra system call disponible en los sistemas *nix, esta system call tiene el nombre creat y su sintaxis es la siguiente:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char* pathname, O_CREAT|O_WRONLY|O_TRUNC, mode_t modo);
int creat(const char* pathname, mode_t modo);
```

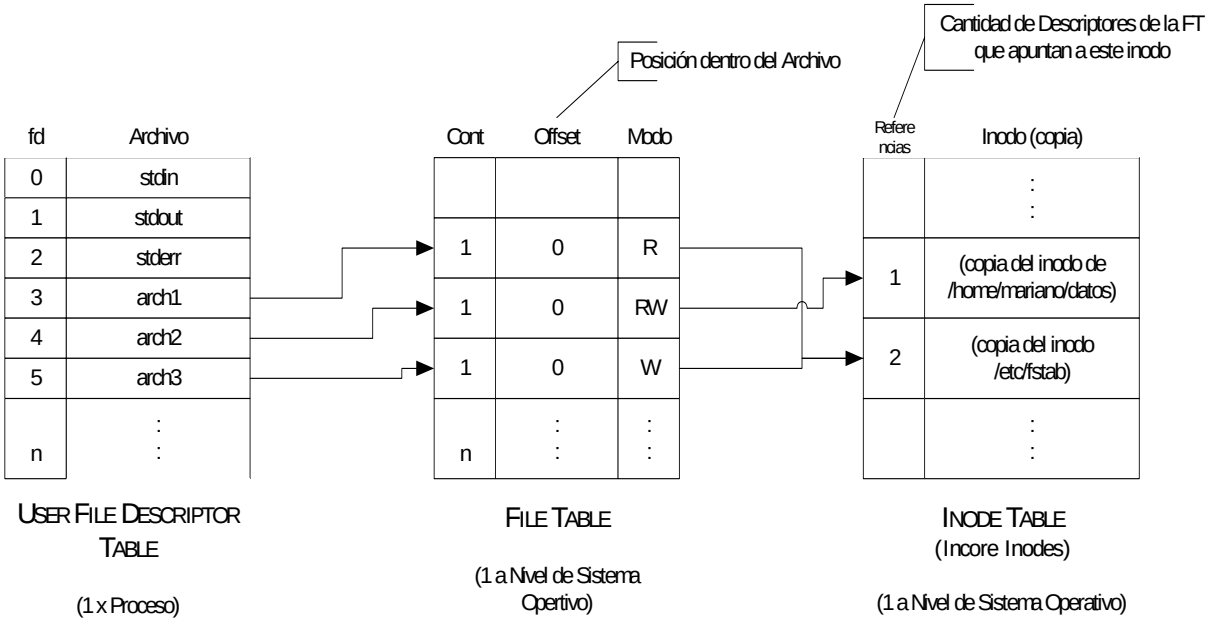
Como puede verse la única diferencia entre la función creat y la función open es que la system call creat es en realidad una subfunción de la system call open con una serie de flags predeterminados.

Ejemplo de utilización de la system call open:

Supongamos que un proceso realiza las siguientes operaciones utilizando la system call open:

```
arch1=open("/etc/fstab",O_RDONLY);
arch2=open("/home/mariano/datos",O_RDWR);
arch3=open("/etc/fstab",O_WRONLY);
```

Las llamadas realizadas por el proceso de ejemplo se ven reflejadas en el siguiente estado de las tablas de archivos del sistema operativo:



close

```
#include <unistd.h>

int close(int fd);
```

Cierra un file descriptor y lo deja disponible para cualquier otro archivo que lo necesite utilizar.

Si es la última copia de un file descriptor asociado a un archivo, los recursos asociados a ese archivo son liberados.
Cerrar un archivo no implica que los datos grabados en el mismo hayan sido escritor a disco. Esto es así ya que el sistema operativo utiliza técnicas de escritura diferida (Buffer Cache).

La system call devuelve 0 si la operación tuvo éxito y -1 en caso de Error.

read

```
#include <unistd.h>

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

Intenta leer *count* bytes del archivo apuntado por el *fd* como primer parámetro y lo graba en el buffer que comienza en la dirección *buf*. La posición dentro del archivo está definida por el valor del offset del archivo indicado en la entrada de la File Table del sistema operativo a la cuál apunta el *fd*.
Si la función finaliza correctamente devuelve la cantidad de bytes leídos que generalmente será igual a *count*. Existe la posibilidad de que la función devuelva un número menor a *count,* esto es así cuando al leer se llega al final del archivo y ya no quedan más datos para copiar al buffer. Es en este caso que el valor devuelto por la system call será únicamente los bytes leídos hasta el fin de archivo.

Esta system call modifica la File Table del sistema operativo, actualizando el offset al último byte leído.

write

```
#include <unistd.h>

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

Intenta escribir *count* bytes en el archivo apuntado por el fd como primer parámetro y tomando los datos a escribir del buffer que comienza en la dirección *buf*. La posición dentro del archivo está definida por el valor del offset del archivo indicado en la entrada de la File Table del sistema operativo a la cuál apunta el *fd*.
Si la función finaliza correctamente devuelve la cantidad de bytes escritos que generalmente será igual a *count*. En caso de error devuelve -1 y se deberá verificar el valor devuelto en errno.

Esta system call modifica la File Table del sistema operativo, actualizando el offset al último byte leído.

lseek

```
#include <unistd.h>
#include <sys/types.h>

off_t lseek(int fd, off_t offset, int whence);
```

Permite reposicionar el offset del *fd* pasado como primer parámetro. Modifica la File Table del sistema operativo.
whence puede tomar los siguientes valor que modifican la forma en la que se calcula el nuevo offset:
SEEK_SET:
EL nuevo offset pasa a ser exactamente el indicado por el parámetro *offset*.
SEEK_CUR:
El nuevo offset pasa a ser el offset actual + el valor indicado por el parámetro *offset*.

Esta system call devuelve el valor del offset nuevo. En caso de error devuelve -1.

dup

```
#include <unistd.h>

int dup(int oldfd);
```

Copia un file descriptor (*oldfd*) en el primer slot libre de la user file descriptor table (UFDT), devolviendo el nuevo file descriptor al usuario o -1 en caso de error. Trabaja con todos los tipos de archivos.

Como el **dup** duplica el fd, incrementa el contador de la correspondiente entrada en la file table. Notar que ambos apuntan a la misma entrada, con un solo offset en el archivo.

dup2

```
#include <unistd.h>

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

El descriptor antiguo (*newfd*) es cerrado con **dup2** antes de ser asignado a el archivo nuevo. Nos permite además asignar el file descriptor (*newfd*) que deseemos a la copia del file descriptor antiguo (*oldfd*).
Esta función devuelve el nuevo file descriptor asignado (*newfd*) o -1 en caso de error.
Con esta particular llamada, tenemos la operación de cerrado, y la duplicación del descriptor actual, relacionado con una llamada al sistema.

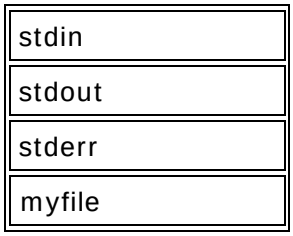
Además, se garantiza el ser atómica, que esencialmente significa que nunca se interrumpirá por la llegada de una señal. Toda la operación transcurrirá antes de devolverle el control al núcleo para despachar la señal. Con la llamada al sistema **dup** original, los programadores tenían que ejecutar un **close** antes de llamarla.

Esto resultaba de dos llamadas del sistema, con un grado pequeño de vulnerabilidad en el breve tiempo que transcurre entre ellas. Si llega una señal durante ese tiempo, la duplicación del descriptor fallaría. Por supuesto, **dup2** resuelve este problema para nosotros.

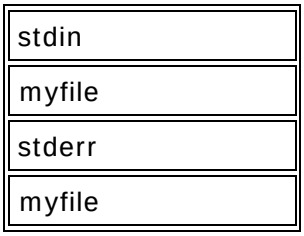
Ejemplo de utilización de dup y dup2 para redireccionar stdin y stdout

```
write(1, buff, 20);
fd = open("myfile", O_WRONLY|O_CREAT, 066);
dup2(fd, 1);  //acá redireccioné la salida estándar
close(fd); // cierro el file descriptor pues ya lo tengo en fd = 1 y no
           necesito la copia
write(1, buff, 20);
```

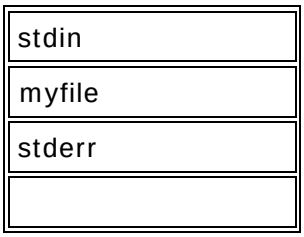
Después del open:



Después del dup2:



Después del close:



link

```
#include <unistd.h>

int link(const char* oldpath , const char* newpath);
```

Este system call linkea un archivo a un nuevo nombre en el file system, creando una nueva entrada de directorio para el nodo existente.
El file system contiene un path para cada link que el archivo tenga, un proceso puede acceder al archivo por medio de cualquiera de estos paths. El Kernel no identifica al nombre original del archivo, por consecuencia, todos los paths son tratados en forma equitativa. (No existe forma de diferenciar el path original del duplicado o link.
Solamente el usuario superuser tiene permiso para linkear directorios, de manera de evitar un loop en los links.

El Kernel utiliza el **namei** para localizar el inodo del archivo fuente, e incrementa el contador de links, actualiza el inodo a disco. Luego utiliza el mismo algoritmo para ubicar el archivo destino. Si lo encuentra, entonces el LINK falla, sino, busca un slot libre en el directorio padre, escribe el nombre del archivo y el # de inodo del archivo fuente, finalmente libera el inodo por medio del IPUT. Este tipo de link es denominado **Hardlink**

unlink

```
#include <unistd.h>

int link(const char* pathname);
```

Remueve una entrada de directorio para un archivo. Si el path al cual se aplica el **unlink** es el último link al archivo, el Kernel libera los bloques de datos. Si no es el último, solo baja la cantidad de links al archivo.
El Kernel utiliza una variante del algoritmo **namei** para ubicar el archivo a modificar. En vez de devolver el inodo del archivo, devuelve el inodo del directorio padre. Utiliza el **iget** para subirlo a memoria y luego escribe un 0 en el dato de # de inodo del archivo. El Kernel graba el directorio a disco, decrementa el contador de links y por medio del **iput** libera los inodos del directorio padre y del archivo.
Si al hacer el **iput** el contador de links es 0, se procede a liberar los inodos y bloques de datos del archivo.

Consistencia del File System

Cuando el File system remueve un archivo de su directorio padre, realiza una escritura sincrónica a disco del directorio, antes de liberar el inodo y los bloques correspondientes al archivo en cuestión.
Si el File System sufre algún problema antes de remover el contenido del archivo, los daños serian mínimos: el inodo del archivo tendría un contador de links mayor al actual, pero el los demás paths serian legales.
Si la escritura del directorio no fuese sincrónica, podría ser que la entrada en el directorio apuntase a un inodo libre o realocado después del fallo del sistema.
El Kernel también libera los inodos y bloques de disco en un orden especial, Cuando libera el contenido de un archivo.
Supongamos que el Kernel libera los bloques del archivo primero, y luego sufre un fallo. Cuando el sistema rebootea, el inodo contiene referencias a los bloques liberados, los cuales pudieron haber sido asignados a otro archivo.
Es por eso que primero modifica el inodo del archivo, lo graba a disco y luego trabaja liberando a los bloques de disco.

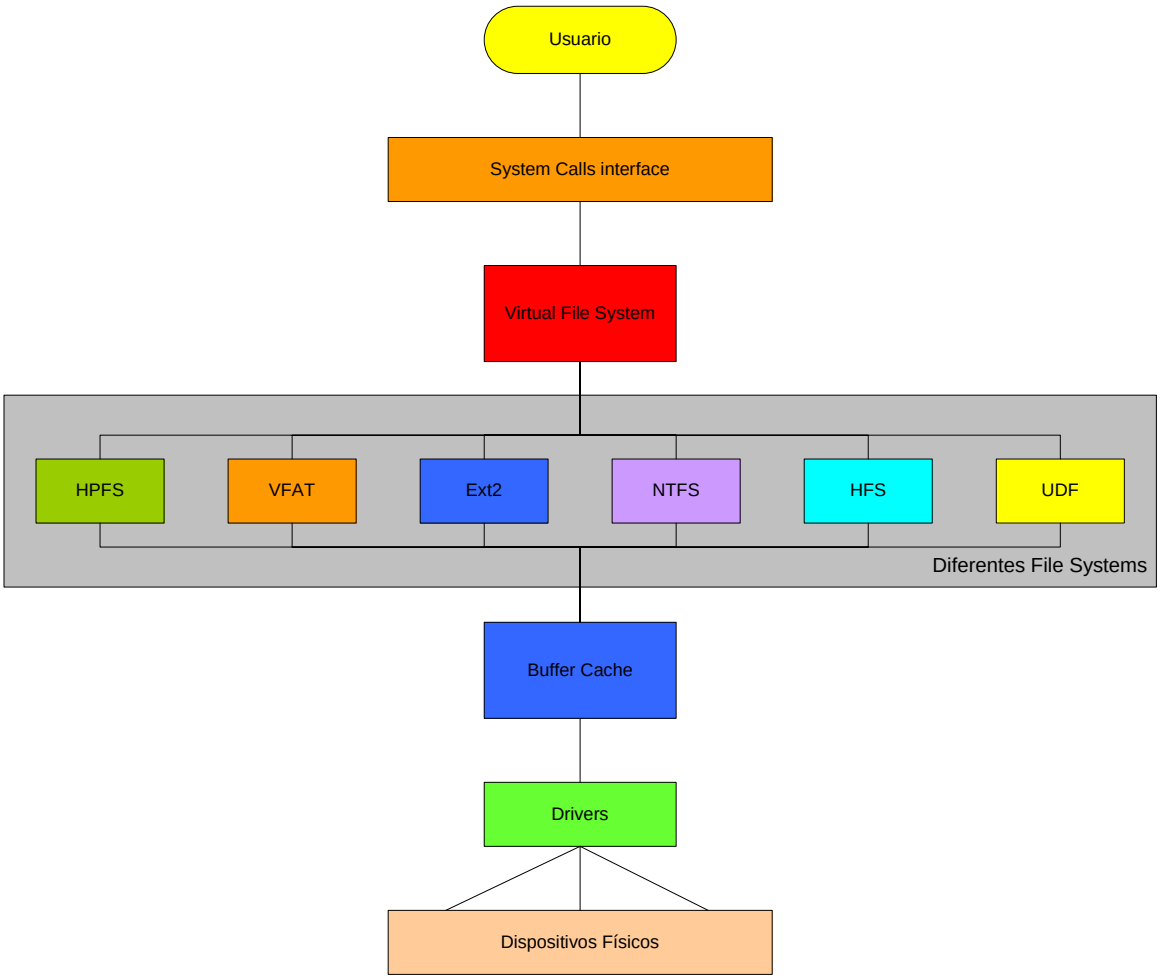
Race Conditions

Los race conditions pueden presentarse en system calls como **unlink**, por ejemplo cuando se ejecuta el comando **rmdir**, para eliminar un directorio, antes de eliminarlo, el Kernel debe asegurarse de que este vacío, es decir, que todos los paths que contenga, tengan como # de inodo un 0. Para esto debe leer el bloque que contiene los datos del directorio.

Puede ser entonces que cuando el comando **rmdir** termino de verificar que todas referencias son al inodo 0, un **creat** cree un archivo en este directorio.

Para evitar esto, se deben utilizar lockeos en los inodos y los bloques de los archivos. Una vez que un proceso comenzó a ejecutar el **unlink**, ningún otro proceso puede acceder al archivo, ya que los inodos del directorio padre y del archivo en si están lockeados.

Virtual Filesystem (VFS)



El virtual file system propone una estructura de llamas a funciones que permite la utilización de distintos tipos de sistemas de archivos sin importar cuál sea su estructura interna. Normalmente esto se realiza a través de módulos del Kernel (también puede compilarse en un Kernel monolítico). Los cuales proveen la interfase necesaria para adaptar el Filesystem en cuestión al VFS. Es así que luego la capa de System Calls es común a todo el sistema operativo. A partir de este momento es que tenemos una abstracción clara y marcada que nos da una clara independencia de la organización interna de cada Filesystem.
Como puede verse también, el buffer cache también es común a todo el sistema operativo, con lo que las ventajas de utilizar esta técnica se propagan a cualquier Filesystem que uno monte en un sistema operativo *nix.

mount

```
#include <sys/mount.h>

int mount(const int* source, const char* target, const char* filesystemtype,
unsigned long mountflags, const void* data);
```

Permite agregar el filesystem determinado por *source* al directorio especificado en *target*. *source* es generalmente un nombre de dispositivo como por ejemplo /dev/fd0 o /dev/hda1 que se corresponden con la primer diskettera y con la primer partición del HDD master conectado en la controladora ide 0 del equipo.
Aquí vemos claramente que todos los dispositivos del sistema son archivos para el sistema operativo y que lo único que indicamos con esta función es cómo se debe interpretar la información leída de ese archivo y dónde queremos dejar disponible para el usuario final.
La system call mount únicamente puede ser invocada por el superusuario (root) o por cada usuario al que root le haya dado permisos de mount. Normalmente en el archivo /etc/fstab se guardan una serie de parámetros por defecto según el dispositivo físico.
El parámetro *fylesystemtype* puede tomar alguno de los siguientes valores: vfat, ext2, iso9660, proc, ntfs, etc. A medida que son incluidos más módulos en el Kernel, esta lista puede ir creciendo, para permitir soportar mayor cantidad de filesystems.
Los parámetros *mountflags* y *data*, vamos a obviarlos en este curso, dado que únicamente proporcionan información relevante para el filesystem indicado en *filesystemtype*.

umount

```
#include <sys/mount.h>

int umount(const char* target);
```

Es la contra función de mount, permite eliminar la relación entre el punto de montaje *target* y el dispositivo físico. Al realizar esta función se escriben todos los datos que estaban pendientes de ser guardados en el filesystem, para luego si proceder a desmontarlo.

Pipes

Pipes permiten la transferencia de datos entre procesos en modo **FIFO** (first in first out), y la sincronización de ejecución de procesos. Permite la comunicación de los procesos aun sin que un proceso sepa quien esta del otro lado del pipe.
Hay dos tipos de pipes, **named pipes** y **unnamed pipes**. La única diferencia entre ambos es la manera que tienen los procesos de acceder inicialmente a los últimos.
Un named pipe es un archivo cuya semánticas son las misma que los unnamed pipes, excepto que tiene una entrada de directorio y es accedido por medio de un path. Los procesos acceden al pipe igual que a otros archivos,
Los named pipes existen permanentemente en el file system, los unnamed pipes son transitorios (cuando los procesos terminan de usarlo, el Kernel libera el espacio asignado).
Los procesos usan la system call **open** para los **named pipes**, y la system call **pipe** para crear los **unnamed pipes**. Solamente los procesos descendientes del proceso que utilizo **pipe**, pueden acceder al unnamed pipe. Sin embargo, cualquier proceso puede acceder a un named pipe.

mknod

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int mknod(const char* pathname, mode_t mode, dev_t dev);
```

Crea archivos especiales en el sistema, incluyendo pipes, devices y directorios. Es similar al system call **creat**, en el sentido de que aloca un inodo para el archivo.
En el caso de que el archivo a crear sea un directorio, el directorio existe luego de la llamada al system call, pero el formato no es el correcto. Es por eso que en Linux particularmente no se puede utilizar esta system call para crear un directorio y debe utilizarse **mkdir**.
El parámetro *pathname* indica dónde debe crease el archivo especial, *mode* indica el tipo de archivo a crear (directorio, named pipe, etc.) y *dev* será ignorada en este curso.

Valores posibles para el parámetro *mode*:
S_IFIFO:
named pipe.
S_IFCHR:
Character device, como por ejemplo el teclado.
S_IFBLK:
Block device, como por ejemplo un disco rígido.
S_IFREG:
Regular File, crea un archivo común.
Según la documentación de Linux, tampoco puede utilizarse esta system call para la creación de pipes del tipo FIFO, ya que existe otra system call que realiza esa tarea, llama **mkfifo**.
Lo mismo ocurre con los sockets, para los que deberá utilizarse la system call **bind.**

pipe

```
#include <unistd.h>

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

Permite la creación de un **unnamed pipe**. *filedes* es un array de enteros que contendrá los dos file descriptors para lectura y escritura del pipe. El Kernel asigna un inodo para el pipe (**ialloc**) de un device especial para pipes y aloca dos entradas en la file table y user file descriptor table.

La función devuelve 0 en caso de finalizar exitosamente o -1 en caso de error.

Los unnamed pipe tienen especial sentido cuando uno utilizar la system call fork.

fork

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);
```

Crea un proceso hijo del proceso actual que difiera únicamente en el PID.
Si la función finaliza exitosamente, devuelve el PID del proceso hijo al padre y 0 al hijo. En caso de Error devuelve -1 y errno tendrá una descripción más precisa del error.