COMUNICACIÓN ENTRE PROCESOS (IPC)

Inter-Process Communication (IPC)

- Conjunto de técnicas para el intercambio de datos entre múltiples threads de diferentes procesos

- IPC también permite la sincronización de procesos aunque no todo mecanismo u *objeto* IPC la asegura

- Los procesos pueden comunicarse entre sí mediante:
  - espacios de memoria
  - elementos de file system
  - otros mecanismos de IPC como ser señales, cola de mensajes, semáforos, etc.

- IPC System V (Unix System V, AT&T)
  Colas de mensaje, semáforos y memoria compartida
- IPC Posix

- Los objetos IPC son mantenidos en la memoria del kernel

Tipos de comunicación

- *Sincrónica*
  El emisor se bloquea a la espera de respuesta del receptor
  Asincrónica
  El emisor continua ejecutándose mientras se comunica con el receptor

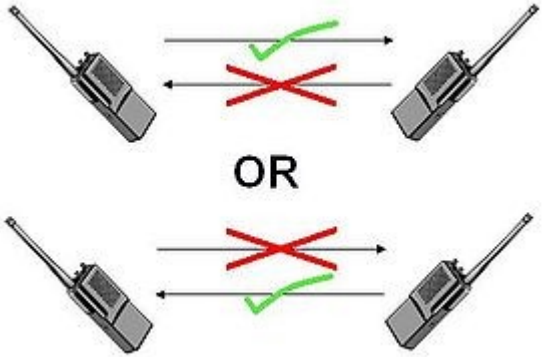
- *Persistente (Persistent)*
  El receptor no tiene que estar operativo durante la comunicación, los mensajes se almacenan el tiempo que sea necesario hasta que los reciba. Ejemplo: e-mail
  Momentánea (Trasient)
  Los mensajes se descartan si el receptor no está operativo. Ejemplo: teléfono

- *Entre dos o más procesos*

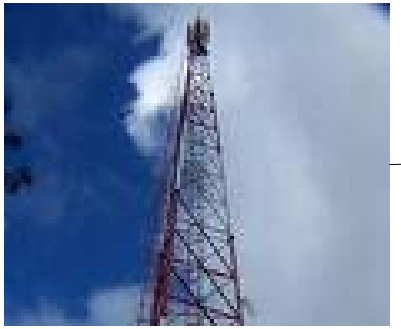
Tipos de comunicación

- *Full duplex:* Envío y recepción simultánea

- *Half duplex:* En ambos sentidos pero no de forma simultánea
  Semi duplex



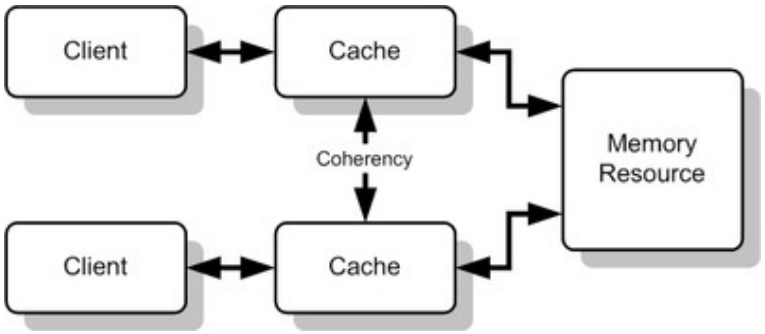
- *Simplex:* Comunicación unidireccional



Shared Memory

- Permite que dos o más procesos accedan simultáneamente a una misma porción o *segmento* de memoria

- Es un mecanismo de IPC eficiente respecto a otros (como ser FIFO y sockets) pero impone la limitaciones:
  - que los procesos convivan en el mismo sistema
  - si los procesos que se comunican se ejecutan sobre distintas CPUs en un mismo sistema éste debe ser *cache coherente*



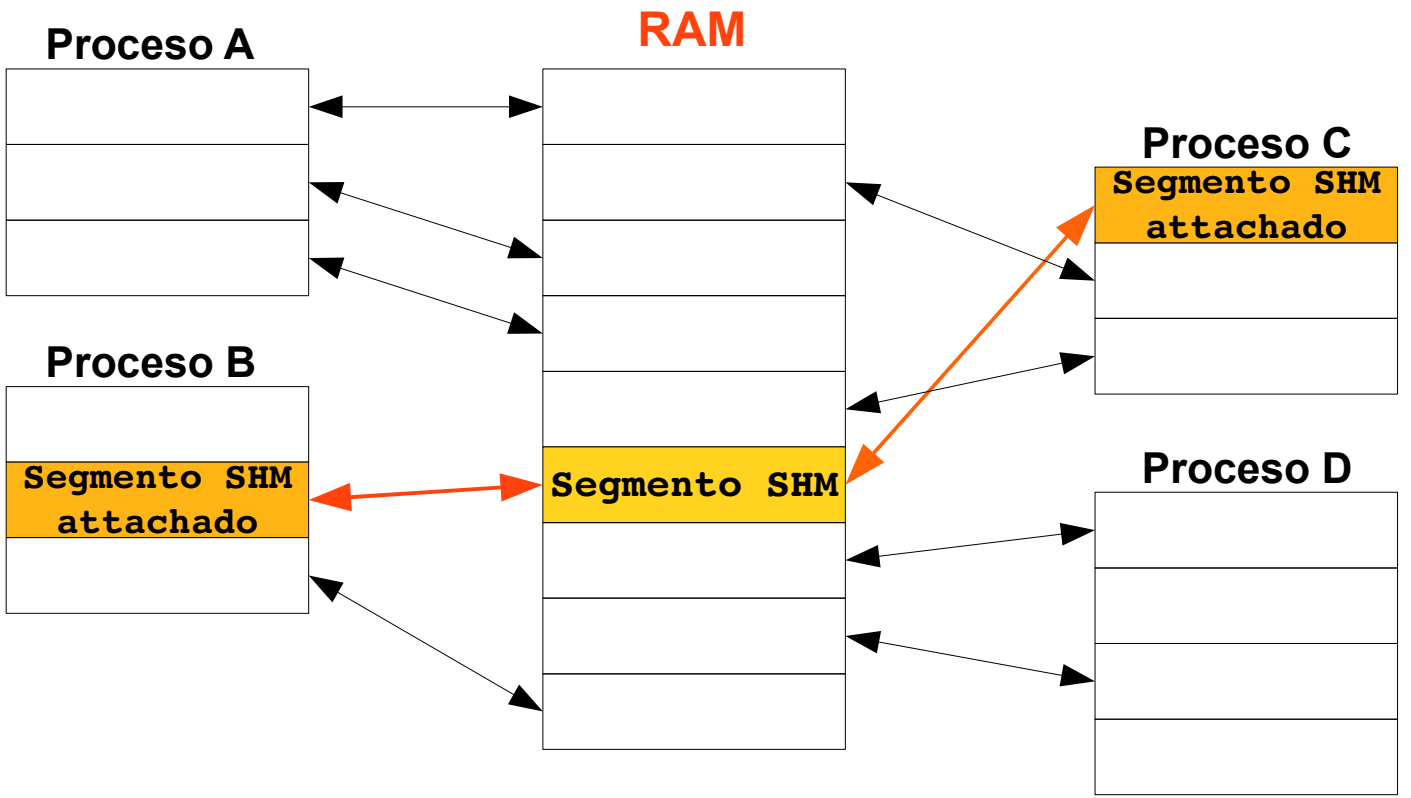
- Si un proceso cuenta con los permisos de acceso a un segmento de memoria puede accederlo mapeándolo a su propio espacio privado de memoria

- Todo cambio realizado sobre un segmento de memoria resulta visible para el resto de los procesos que la comparten
  Por tanto deben evitarse las condiciones de competencia que puedan presentarse

Shared Memory

- Un segmento creado por un proceso puede ser leído y/o escrito por otros procesos

- Cada proceso recibe su propio mapa de la memoria compartida en su espacio privado de memoria



Shared Memory

- Procedimiento:
  - Un proceso debe *allocar* el segmento de memoria
  - Los procesos deben *attachar* (mapear) el segmento a su propio espacio de direcciones
  - Al finalizar la comunicación cada proceso debe *dettachar* el segmento
  - El proceso creador debe *desallocar* el segmento de memoria

- La memoria virtual de los procesos es dividida en páginas.
  Todo proceso hace referencia a direcciones de memoria virtual.

- Cada proceso mantiene un mapeo de sus propias direcciones de memoria a estas páginas de memoria virtual

- Allocar un nuevo segmento (no existente) de *shared memory* provoca la creación de páginas de memoria virtual

Shared Memory: Allocation

• Función

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>        // shmid
int shmget(key_t key, size_t size, int shmflag)
```

- `key`
  entero que indica el segmento a crear
  `IPC_PRIVATE` garantiza la creación de un segmento nuevo

- `size`
  especifica (en bytes) el tamaño del segmento, se redondea a *n*pagesize*

- `shmflag`
  `IPC_CREAT` un nuevo segmento debe ser creado
  `IPC_EXCL` uso junto a IPC_CREAT, `shmget` falla en caso de que el segmento `key` ya exista
  Modos: 9 bits que indican permisos para *owner*, *group* y *world* de control de acceso
  Los bits de ejecución son ignorados, constantes en `sys/stat.h`

• Ejemplo

```
size_t seg_id= shmget(key, getpagesize(), IPC_CREAT|S_IRUSR|S_IWUSR)
```

Shared Memory: Attach & Dettach

• Función

```
void* shmat(size_t shmid, const void *shmaddr, int shmflag)
```

- `shmid`
  identificador del segmento
- `shmaddr`
  puntero que especifica donde mapear el segmento en el espacio de direcciones del proceso. NULL o 0 = El sistema operativo lo determina
- `shmflag`
  - **`SHM_RND`** especifica que la dirección indicada por `shmaddr` debe redondearse
  - **`SHM_RDONLY`** indica que se attache el segmento en sólo lectura, por defecto L/E

Los hijos heredan los segmentos attachados del proceso padre

Se realiza un dettach mediante la función

```
int shmdt(const void *shmaddr)
```

`shmaddr` es el valor retornado por `shmat`

Al invocar a *exit* o *exec* se dettachan los segmentos automáticamente

Shared Memory: Deallocation & Control

• Función

```
void shmctl(size_t shmid, int cmd, struct shmid_ds *buf)
```

- retorna información acerca de un segmento y permite modificarlo
- `cmd=IPC_STAT` para obtener información
- `cmd=IPC_RMID` y `buf=NULL` para borrar un segmento
- cada segmento debe ser explícitamente desallocado usando `shmctl`

Un proceso puede modificar un segmento mediante `shmctl` si se trata del proceso dueño o creador del *shmid*

Otros valores para `cmd`

SHM_LOCK
SHM_UNLOCK
IPC_SET: establece permisos

- **Shared Memory: Ejemplo**

  `shm.c`

- **Shared Memory:**

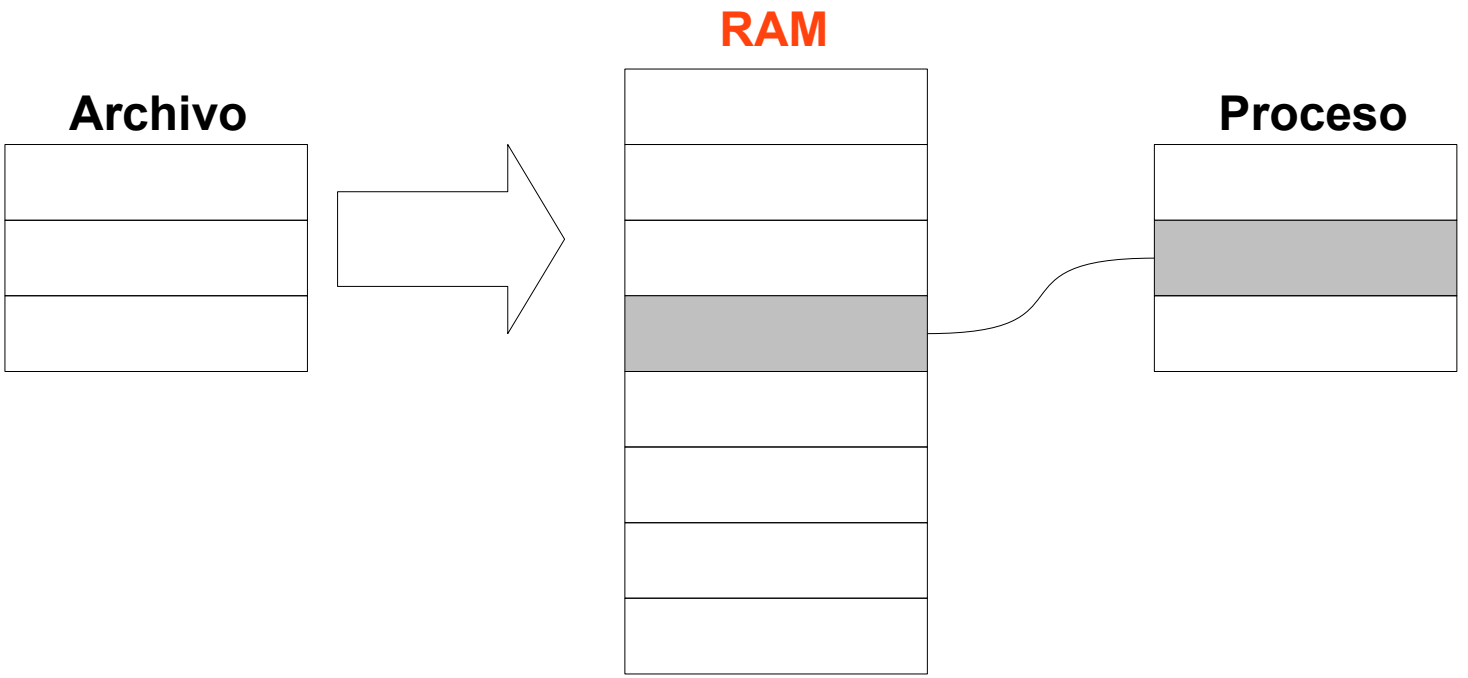
• Comandos

```
# ipcs -l

# ipcs -m
  (nattach)
```

Mapped Memory

- Permite que procesos (relacionados o no) se comuniquen a través de un archivo
- Puede emplearse para IPC o para optimizar el acceso a archivos
- Este mecanismo de IPC establece una relación entre un archivo y la memoria de un proceso
- Un archivo se divide en trozos o bloques de tamaño de una página de memoria los cuales se copian a páginas en memoria virtual para su posterior acceso
- Los procesos pueden leer o modificar el contenido del archivo mediante simples accesos a memoria

- **Mapped Memory**



Mapped Memory

• Funciones

- es posible mapear un archivo total o parcialmente

```
#include <sys/mman.h>
#include <sys/types.h>
void* mmap(void *addr, size_t len, int prot, int flags, int fd, off_t off)
```

`addr`: dirección del espacio de direcciones del proceso en donde se desea mapear el archivo

`len`: tamaño del mapeo en bytes

`prot`: indica nivel de protección (`PROT_READ, PROT_WRITE, PROT_EXEC`)

`flags`:opciones

- `MAP_FIXED`: `addr` se considera tal cual está, debe ser *page_aligned*
- `MAP_PRIVATE`: escrituras a archivo privado, no apto para IPC
- `MAP_SHARED`: escrituras a memoria van a archivo subyacente, útil para IPC

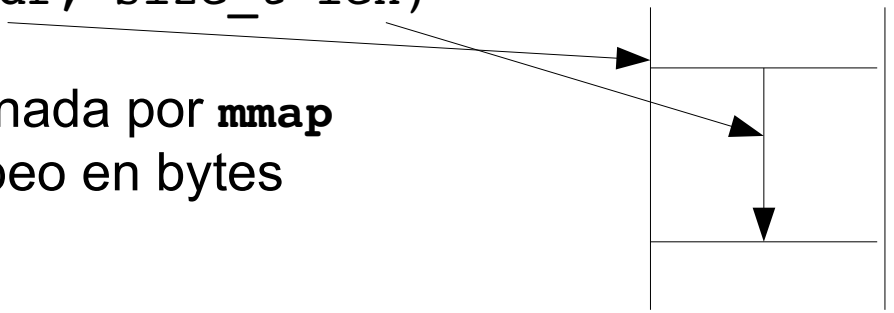
`fd`: file descriptor relacionado con el archivo mapeado

`off`: offset desde el principio del archivo desde donde se comienza a mapear

Mapped Memory

- Funciones

```
int munmap(void *addr, size_t len)
```



  - `addr`: dirección retornada por **`mmap`**
  - `len`: tamaño del mapeo en bytes

Mapped Memory - Escritor

```
#include <stdlib.h>             #include <stdio.h>
#include <fcntl.h>              #include <sys/mman.h>
#include <sys/stat.h>           #include <unistd.h>
#define FILE_LENGTH 0x100
#define INTEGER 123456

int main (int argc, char* const argv[]) {
    int fd;
    void* file_memory;

    fd = open(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    lseek (fd, FILE_LENGTH+1, SEEK_SET);
    write (fd, "", 1);
    lseek (fd, 0, SEEK_SET);

    /* Crea el mapeo de memoria. */
    file_memory = mmap(0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
    close (fd);

    /* Escribe un entero al area de memoria mapeada. */
    sprintf((char*)file_memory, "%d\n", INTEGER);

    /* Libera la memoria. */
    munmap (file_memory, FILE_LENGTH);

    return 0;
}
```

Mapped Memory - Lector

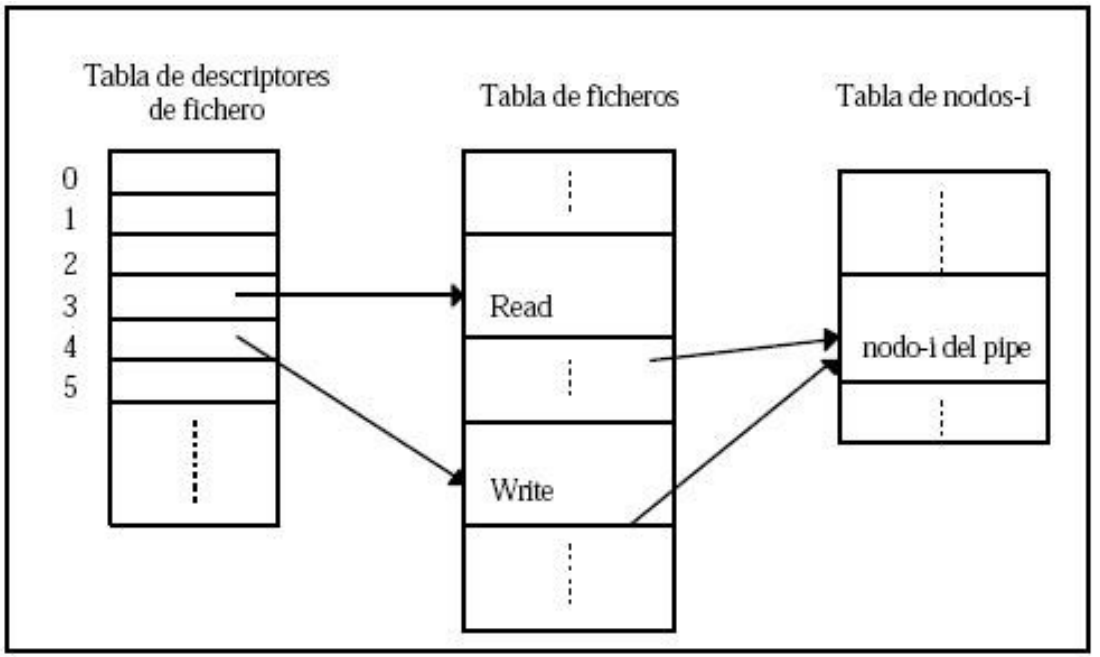
```
#include <stdlib.h>             #include <stdio.h>
#include <fcntl.h>              #include <sys/mman.h>
#include <sys/stat.h>           #include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[]) {
    int fd;
    void* file_memory;
    int integer;

    /* Abre el archivo. */
    fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);

    /* Create the memory mapping. */
    file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    close (fd);

    /* Lee el entero, lo imprime y lo duplica. */
    sscanf ((char*)file_memory, "%d", &integer);
    printf ("El valor del entero es: %d\n", integer*2);

    /* Libera la memoria */
    munmap (file_memory, FILE_LENGTH);

    return 0;
}
```

Pipes

- Permite la comunicación *half-duplex* entre dos procesos *relacionados*
- Son dispositivos seriales, se mantiene el orden de datos en emisión/recepción
- Son dispositivos *non_seekables* (no pueden manipularse con funciones tipo lseek)
- Usos típicos:
  - comunicar dos threads de un mismo proceso
  - comunicar dos procesos relacionados (o emparentados)
- Son conocidos también con el nombre “pipes sin nombre”

Pipes

- Cuando se crean el kernel le asigna.
  - un i-nodo
  - dos entradas en la tabla de archivos
  - dos entradas en la tabla de descriptores de archivo



Pipes

- Cuando se escribe en un pipe el kernel asigna bloques de disco (bloques directos de inodo)

- Cuentan con capacidad limitada
  bloques directos x tamaño bloque
  12 x 8KB = 96KB

  Debido a esto se sincronizan procesos
  - escritor rápido se bloquea al llenarse el pipe empleado para comunicarse con lector lento

- Un proceso hijo hereda los descriptores de archivo presentes en el proceso padre

Pipes

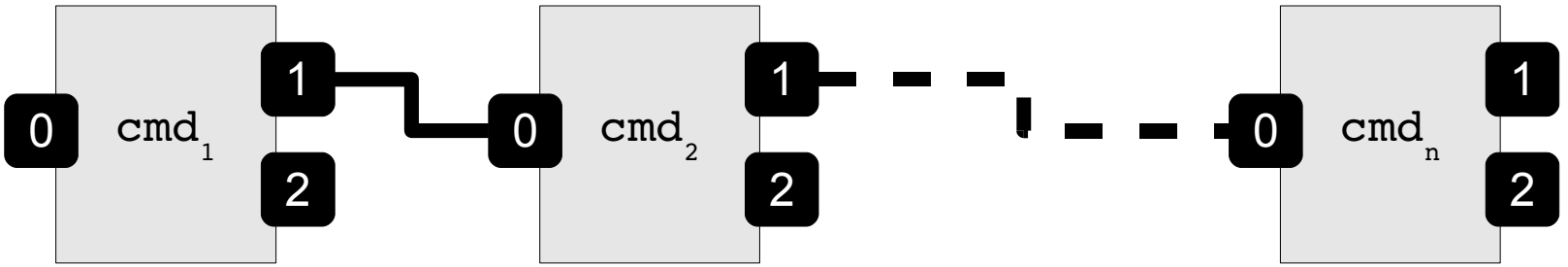
- En el shell, el símbolo " | " crea un pipe

```
$ ls | sort
```

- A los comandos tipo

```
$ cmd1 | cmd2 | ... | cmdn
```

se los denomina *pipelines*



Pipes: Apertura y cierre

- ***Apertura***

```
<unistd.h>
int pipe(int fds[2])
```

- Retorno: *0=éxito, 1=fracaso*
- En caso de éxito
  - abre dos descriptores de archivo y los almacena en el array fds
  - `fds[0]`: se utiliza para lectura, `O_RDONLY`
  - `fds[1]`: se utiliza para escritura, `O_WRONLY`

- Se pueden presentar errores debido a
  - `EMFILE`: el proceso que ejecutó la llamada sobrepasó la cantidad de archivos abiertos
  - `EFAULT`: array argumento inválido
  - `ENFILE`: tabla de archivos del kernel llena

- ***Cierre***

```
int close(int fd)
```

Pipes: Lectura y escritura

- **Opción 1**

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

se lee de un extremo del pipe y se escribe en el otro

- **Opción 2**

convertir los descriptores de archivo en objetos `FILE*` empleando

```
FILE *fopen(char *nombre_archivo, char *modo)
int fprintf (FILE * stream, const char *format, ... )
char * fgets (char *str, int num, FILE *stream )
int fscanf(FILE *stream, const char *format, ... )
int fclose(FILE *pf);
```

Pasos: comunicación padre-hijo con un pipe

- un proceso invoca a *pipe* y posteriormente a *fork*
- el hijo hereda los descriptores de archivo que el padre mantiene abiertos
- cualquiera de los procesos puede actuar como emisor o receptor, pero sólo puede tomar un rol durante la comunicación
- cuando un proceso lee del pipe debe cerrar el extremo de escritura y viceversa

Pipes

- Comunicación entre PADRE e HIJO, el padre es emisor, el hijo es receptor

- **`PADRE (escribe)`**
  - `close(fds[0])`
  - `write(fds[1], buf, n)`
  - `close(fds[1])`

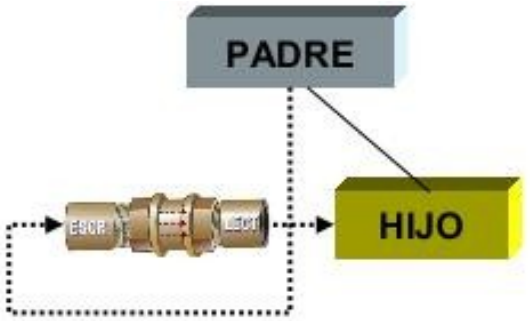
- **`HIJO (lee)`**
  - `close(fds[1])`
  - `read(fds[0], buf, n)`
  - `close(fds[0])`

- Si se desea establecer una comunicación *bidireccional* entre padre e hijo se deben emplear *dos pipes*, no es posible realizarlo con uno solo

Pipes

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#define READ  0
#define WRITE 1

int main () {
  int fds[2];
  pid_t pid;

  pipe(fds);
  pid = fork();

  if (pid == (pid_t) 0) {
     /* Proceso HIJO. Lee desde el pipe. */
     char buffer[1024];
     FILE* stream;
     close(fds[WRITE]); /* Cierra su copia del extremo de escritura */

     stream = fdopen(fds[READ], "r");
     fgets(buffer, sizeof(buffer), stream);
     fputs(buffer, stdout);
     close(fds[READ]);

    } else {
      /* Proceso PADRE. Escribe en el pipe*/
      FILE* stream;
      close(fds[READ]);   /* Cierra su extremo de lectura */

      stream = fdopen(fds[WRITE], "w");
      fprintf(stream, "Mensaje del padre para el hijo...");
      fflush(stream);
      close(fds[WRITE]);
     }

  return 0;
}
```

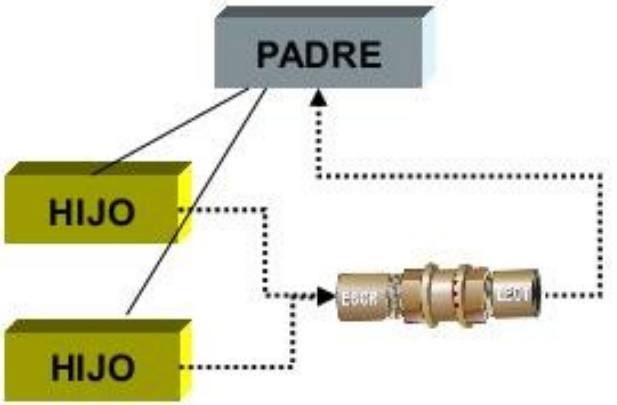
- **Pipes**


PADRE
HIJO


PADRE
HIJO

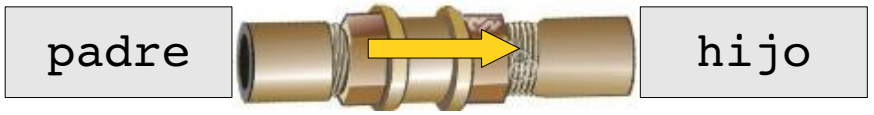

PADRE
HIJO
HIJO

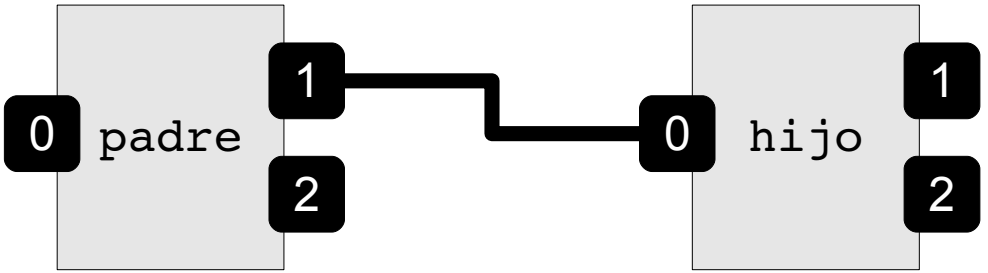

PADRE
HIJO
HIJO
HIJO


PADRE
HIJO
HIJO

Redirecciones de entrada, salida y error estándar

- Generalmente queremos realizar esta operación





- En definitiva lo que se pretende es igualar dos *file descriptors*

Redirecciones de entrada, salida y error estándar

• Funciones

```
#include <unistd.h>
int dup(int fildes)
int dup2(int fildes, int fildes2)
```

- Son interfaces alternativas del servicio ofrecido por *fcntl* (file control)

```
#include <fnctl.h>
int fcntl(int fildes, int cmd, ...);
```

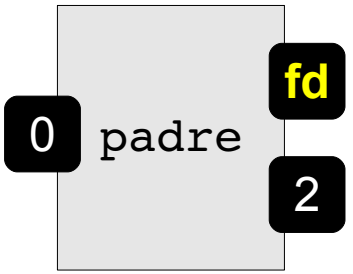
http://www.opengroup.org/onlinepubs/009695399/functions/fcntl.html

`dup(fildes) ≡ fcntl(fildes, F_DUPFD, 0)`

`dup` usa el file descriptor en desuso de número más bajo para el nuevo file descriptor

Ejemplo

```
#include <unistd.h>
...
int fd;
...
close(1);
dup(fd);
close(fd);
```



Redirecciones de entrada, salida y error estándar

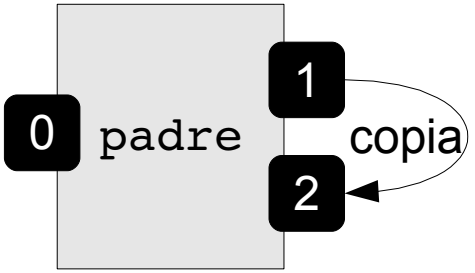
- Funciones

```
dup2(fildes, fildes2) ≡ close(fildes2); &&
                        fcntl(fildes, F_DUPFD, fildes2);
```

`dup2` hace que `fildes2` sea la copia del file descriptor `fildes`

Ejemplo

```
#include <unistd.h>
...
dup2(1, 2);
...
```



”Atajo”

- Es muy común emplear pipes para enviar o recibir datos a o desde un subproceso
- Existen dos funciones que eliminan la necesidad de invocar la serie de funciones

```
pipe – fork – dup2 – exec ...
```

estas son:

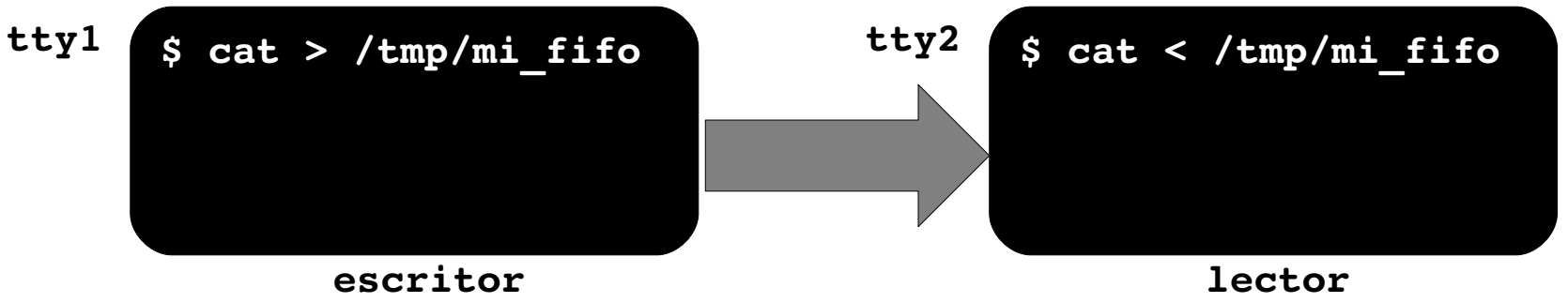
```
FILE* popen (const char* comando, const char *type)
int plcose(FILE* stream)
```

```
#include <stdio.h>
#include <unistd.h>

int main () {
    FILE* stream = popen(“sort”, “w”);
    fprintf(stream, “Hola\n”);
    fprintf(stream, “Hijo\n”);
    fprintf(stream, “Hasta pronto\n”);

    pclose(stream);
    return 0;
}
```

FIFOS

- Denominados como “pipe con nombre”
- Tienen presencia (nombre) a nivel file system
- Dado que poseen un nombre localizable por cualquier proceso pueden comunicar procesos no relacionados
- A través de un FIFO se transfieren datos en una única dirección según cada proceso
- En línea de comandos

```
$ mkfifo /tmp/mi_fifo
$ ls -l /tmp/mi_fifo
prw-r--r-- 1 diego diego 0 2009-10-12 21:17 /tmp/mi_fifo

$ ls -lF /tmp/mi_fifo
pprw-r--r-- 1 diego diego 0 2009-10-12 21:17 /tmp/mi_fifo|

$ rm /tmp/mi_fifo
```



FIFOS

- Ejemplo práctico

```
$ mkfifo mi_fifo
$ gzip -9 -c < mi_fifo > out.gz
```

En otra consola

```
$ cat *.c > mi_fifo
```

FIFOS

- Pueden comunicar múltiples escritores y/o lectores

- Cuando un proceso abre un FIFO para lectura se bloquea hasta que otro proceso escriba en el
  Un lector se bloquea en un FIFO vacío y un escritor en uno lleno

- La escritura en un FIFO es atómica y no excede los 4KB

```
$ grep PIPE_BUF /usr/include/linux/limits.h
#define PIPE_BUF        4096 /* # bytes in atomic write to a pipe */
```

- Las escrituras provenientes de escritores simultáneos pueden intercalarse

FIFOS

• Creación de FIFOS mediante C

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char* nombre_fifo, mode_t modo)
```

- **`nombre_fifo`** se especifica mediante *path* absoluto
- **`modo`** expresado en octal

Valores de retorno
0 = éxito
-1 = falla, debido a

`EEXIST`: nombre de archivo existente
`ENAMETOOLONG`: *path* excesivamente largo
`ENOTDIR`: *path* incorrecto o no existente

• Apertura y cierre
Se emplean las mismas funciones que para archivos aprovechando el concepto de que en *nix *"todo es un archivo"*
Para cierre se puede usar `colse()` y para eliminación `unlink()`

- **Lector**

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <fcntl.h>
#include <errno.h>

int main (int argc, char **argv) {
        int fd;
        char data[PIPE_BUF];
        mode_t modo = 0666;

        if (argc != 2) {
                perror("Error de uso: ./fifo-reader path");
                exit(1);
        }

        if(mkfifo(argv[1], modo) < 0) {
                perror("Error en mkfifo");
                exit(2);
        }
```

```
if ((fd=open(argv[1], O_RDONLY)) < 0) {
                perror("Error de open");
                exit(3);
        }

        /* Leer desde el FIFO y mostar datos */
        while (read(fd, data, PIPE_BUF-1) > 0)
                printf("-> %s\n", data);

        close(fd);
        return 0;
}
```

Escritor

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <limits.h>
#include <fcntl.h>
#include <errno.h>

int main (int argc, char **argv) {
  int fd, nbytes, nmsg=0;
  char data[PIPE_BUF];
  pid_t pid = getpid();

  if (argc != 2) {
    perror("Error de uso: ./fifo-writer path\n");
    exit(1);
  }

  /* Apertura del FIFO */
  if ((fd = open(argv[1], O_WRONLY)) < 0) {
    perror("Error en open");
    exit(2);
  }
  printf("Yo soy el proceso %d\n", pid);

  while (1) {
    /* Mensaje a enviar */
    nbytes = sprintf(data, "Mensaje Nro. %d del escritor [%d]", nmsg++, pid);

    /* Escritura (envio) del mensaje */
   if (write(fd, data, nbytes+1) < 0) {
       perror("Error en write");
       exit(2);
       }
   sleep(2);
   }

  close(fd);
  return 0;
}
```