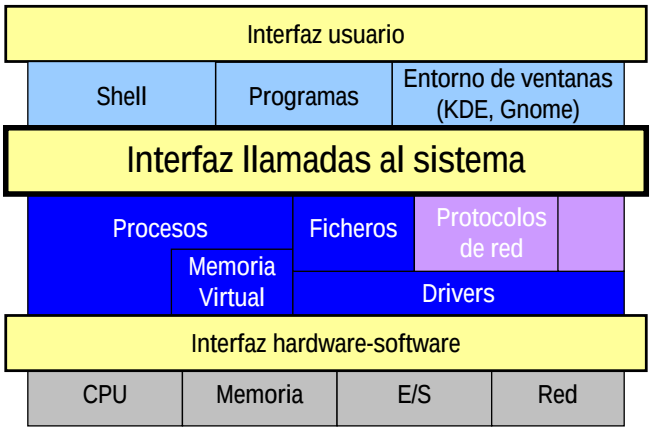
Introducción y Gestión de Procesos

Sistemas Operativos (prácticas)
E.U. Informática en Segovia
Universidad de Valladolid

Llamadas al sistema

- Concepto
  - Constituye el “juego de instrucciones” del sistema operativo
  - Son la interfaz de programación (API, *Application Program Interface*) del sistema operativo
  - A partir del conjunto de llamadas al sistema, pueden construirse órdenes e intérpretes de órdenes para el sistema operativo



Llamadas al sistema

- El estándar POSIX
  - IEEE Std 1003.1-2001 define la ***interfaz estándar de un sistema operativo y su entorno***, incluyendo un intérprete de órdenes (*shell*) y programas de utilidad comunes para soportar la ***portabilidad de aplicaciones a nivel de código fuente***. Se pretende que sea usada tanto por los desarrolladores de aplicaciones como por implementadores de sistemas.
  - IEEE STD 1003.1-2001 se compone de ***cuatro componentes principales***:
    - ***Términos generales, conceptos e interfaces*** comunes a todos los documentos del estándar IEEE Std 1003.1-2001, incluyendo convenciones útiles y definiciones de archivos de cabecera en lenguaje C. Se incluye en el documento "Definiciones Base" del estándar.
    - ***Definiciones de funciones de servicios del sistema y subrutinas, servicios del sistema específicos del lenguaje para C***, y otras cuestiones sobre las funciones: portabilidad, manejo de errores, y recuperación ante errores. Se incluyen en el volumen "Interfaces del Sistema" del estándar.
    - ***Definiciones para una interfaz estándar a nivel de código fuente para servicios de interpretación de órdenes*** (un *shell*) y programas de utilidad comunes para programas de aplicación. Se incluyen en el volumen "*Shell* y utilidades" del estándar.
    - Cuestiones relacionadas que no encajan en la estructura documental del estándar, pero relevante, como información histórica concerniente a los contenidos del estándar y porqué determinadas características fueron incluidas o no por los desarrolladores del estándar. Estas consideraciones se encuentran en el volumen "Razones" del estándar.

Llamadas al sistema

Procesos	
fork	Creación de procesos
exit	Terminación de procesos
wait	Esperar la terminación de un proceso
exec	Cambiar imagen de memoria por la de un ejecutable
getpid	Obtención de atributos de un proceso
setsid	Modificación de atributos de un proceso
Señales	
kill	Enviar una señal
alarm	Generar una alarma
sigemptyset	Iniciar una máscara para que no tenga señales seleccionadas
sigfillset	Iniciar una máscara para que contenga todas las señales
sigaddset	Poner una señal específica en un conjunto de señales
sigdelset	Quitar una señal especificada en un conjunto de señales
sigismember	Consultar si una señal pertenece a un conjunto de señales
sigprocmask	Examinar/modificar máscara de señales
sigaction	Capturar/manejar señales
sigsuspend	Esperar por una señal

Llamadas al sistema

Ficheros	
open	Apertura/creación de ficheros
read	Lectura de ficheros
write	Escritura de ficheros
close	Cierre de un fichero
lseek	Posicionamiento en un fichero
stat	Obtener información asociada a un fichero (tamaño, nombre, tipo, ...)
Redirecciones y tuberías	
dup2	Duplicar un descriptor de fichero
pipe	Creación de una tubería sin nombre (*pipe*)
mkfifo	Creación de una tubería con nombre
Directorios	
mkdir	Crear un directorio
rmdir	Eliminación de un directorio vacío
opendir	Apertura de un directorio
readdir	Leer la siguiente entrada de un directorio
closedir	Cerrar un directorio
link	Crear una nueva entrada de directorio para otra ya existente (*link*)
unlink	Eliminar una entrada de directorio

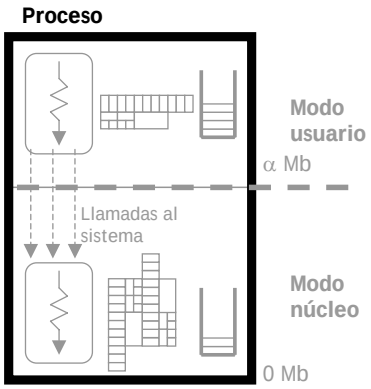
Llamadas al sistema

Protección	
chmod	Modificar los bits de protección (rwx, suid, sgid, ...) de un archivo
chown	Asigna un nuevo propietario y grupo a un archivo
umask	Modificar la máscara de protección de archivos (por defecto) de un proceso
Otras	
mount	Montar un sistema de archivos de un dispositivo sobre un directorio
umount	Desmontar un sistema de archivos
ioctl	Control de dispositivos
time	Valor del reloj de tiempo real
times	Tiempos consumidos por un proceso

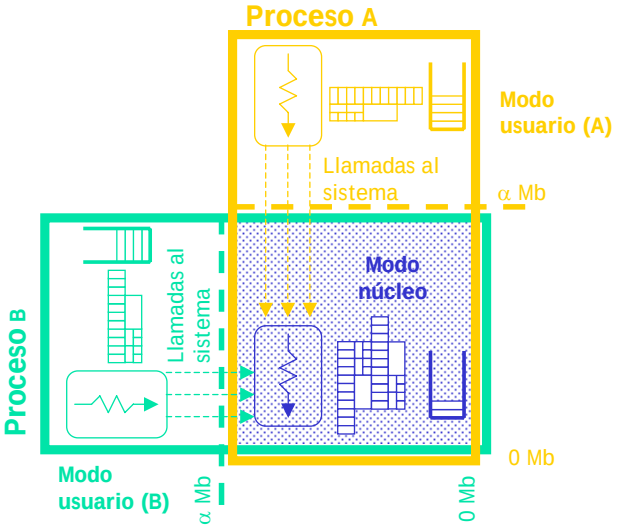
Procesos

- Procesos UNIX
  - Unidad de ejecución y asignación de recursos
  - Programa en ejecución, por tanto, caracterizado por un estado entre cuyos atributos se encuentran:
    - Imagen de memoria (en modo usuario y *kernel*)
    - Identificador del proceso: PID
    - Identificador del proceso padre: PPID
    - Usuario propietario del proceso: rUID y eUID (real y efectivo, respectivamente)
    - Grupo propietario del proceso: rGID y eGID (real y efectivo, respectivamente)
    - Grupo de un proceso: GID
    - Sesión de un proceso: SID
    - Máscara de señales
    - Tiempos de consumo de procesador
    - Descriptores de archivos abiertos
    - Directorio actual (CWD, *Current Work Directory*)
    - Máscara de creación de archivos (umask)
    - ...

Procesos



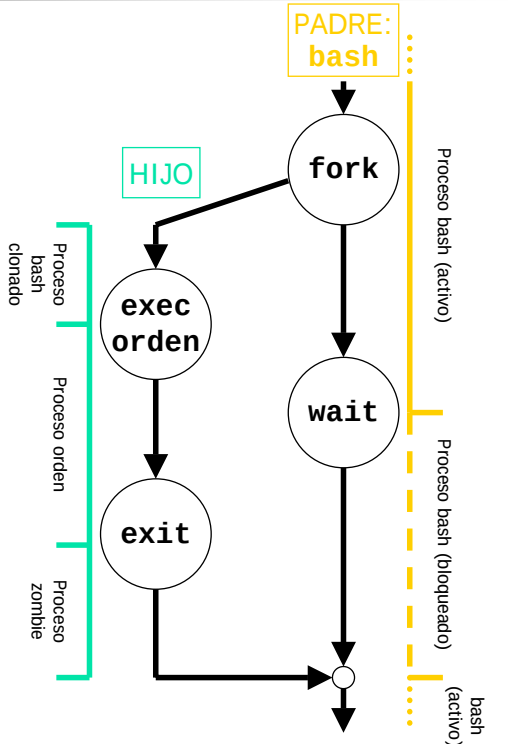
- Un único proceso, dos modos de ejecución



- Dos procesos cualesquiera, comparten un único núcleo

Procesos

- Operaciones sobre procesos
  - Creación: **`fork()`**
    - Creación por copia. Herencia de atributos de atributos a procesos hijos
    - Jerarquía de procesos
  - Terminación: **`exit()`**
    - Terminación normal: `exit()`
    - Terminación anormal: señales
  - Espera de terminación de los hijos: **`wait()`**
  - Sustitución de la imagen de memoria: **`exec()`**



Procesos: llamadas al sistema

Procesos	
`fork`	Creación de procesos
`exit`	Terminación de procesos
`wait`	Esperar la terminación de un proceso
`exec`	Cambiar imagen de memoria por la de un ejecutable
`getpid`	Obtención de atributos de un proceso
`setsid`	Modificación de atributos de un proceso

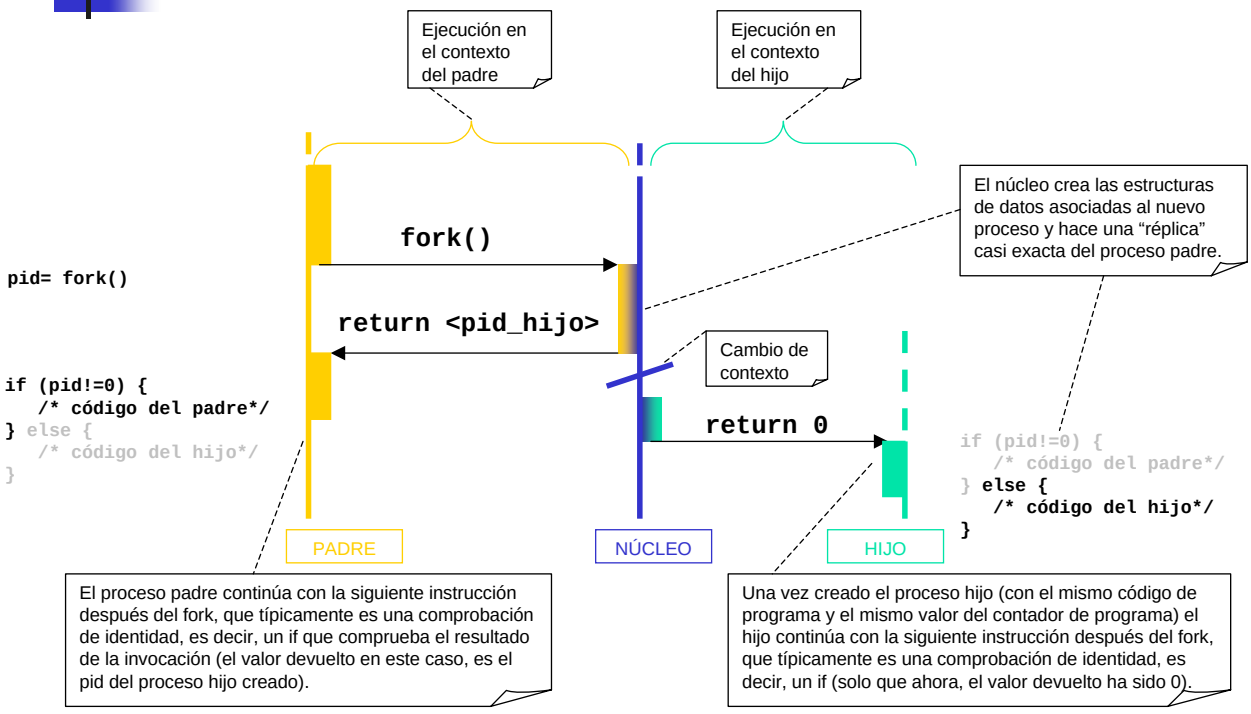
Procesos: `fork`

- **`fork`**: creación de procesos

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

pid_t fork(void)
```

- Descripción
  - Crea un proceso hijo que es un “clon” del padre: hereda gran parte de sus atributos
  - Atributos heredables: todos, excepto PID, PPID, señales pendientes, tiempos/contabilidad
- Valor de retorno
  - 0 al hijo
  - PID del hijo al padre
  - -1 al padre si error
- Errores
  - Insuficiencia de recursos para crear el proceso

Procesos: `fork`



Procesos: **fork**

- **fork**: ejemplo 1

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
   pid_t pid;
   int var= 0;
   printf("PID antes de fork(): %d\n", (int) getpid());
   if ( (pid = fork()) > 0 ) {
      printf ("PID del padre: %d\n", (int) getpid());
      var++;
   } else {
      if (pid == 0)
         printf ("PID del hijo: %d\n", (int) getpid());
      else
         printf ("Error al hacer fork()\n");
   }
   printf("Proceso [%d] -> var = %d\n", (int) getpid(), var);
}
```


Procesos: **fork**

- **fork**: ejemplo 2

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

int main(void) {
   pid_t pid;
   int i, n= 4;

   for (i=0; i<n; i++)
      if ( (pid = fork()) < 0 ) break;

   printf ("Proceso: %d / Padre: %d\n",
           (int) getpid(), (int) getppid());
}
```

Procesos: **exit**

- **exit**: terminación de procesos

```
#include <stdlib.h>

void exit(int status)
```

```
#include <unistd.h>

void _exit(int status)
```

- Descripción
  - Termina “normalmente” la ejecución del proceso que la invoca
  - Si no se invoca explícitamente, se hace de forma implícita al finalizar todo proceso
  - El estado de terminación **status** se transfiere al padre que ejecuta **wait(&status)**
  - Si el padre no está ejecutando **wait**, se transforma en un *zombie*
  - Cuando un proceso ejecuta **exit**, todos los hijos pasan a ser adoptados por un proceso dependiente de la implementación (normalmente **init**) y su PPID pasa a ser el de este proceso (normalmente 1)
- Valor de retorno
  - Ninguno
- Errores
  - Ninguno

Procesos: **wait**

- **wait**: esperar por la terminación de un proceso

```
#include <sys/type.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc)
pid_t waitpid( pid_t pid, int *stat_loc, int options)
```

- Descripción
  - Suspende la ejecución del proceso que la invoca, hasta que alguno de los hijos (**wait**) o un hijo concreto (**waitpid**) finaliza
  - Si existe un hijo *zombie*, **wait** finaliza inmediatamente, sino, se detiene
  - Cuando stat_loc no es NULL, contiene:
    - Hijo termina con exit

MSB: status definido por exit	LSB: 0

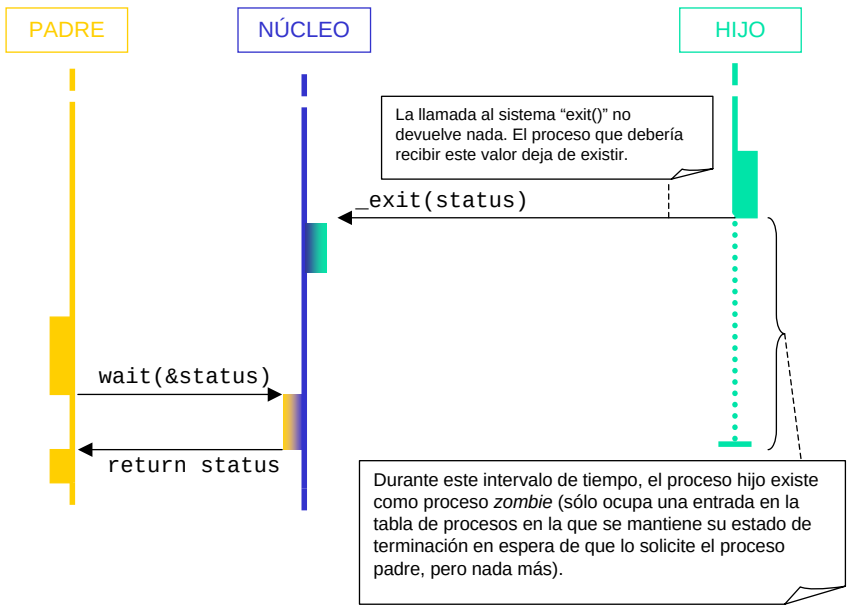
    - Hijo termina por una señal

0	LSB: num. Señal (bit más alto 1 si *core dump*)

Procesos: **wait**

- Descripción (continuación)
  - **waitpid**: argumento **pid**
    - <-1: esperar por cualquier hijo cuyo grupo de procesos sea igual al valor absoluto del **pid** especificado
    - -1: esperar por cualquier hijo (igual que **wait**)
    - 0: esperar cualquier hijo con el mismo grupo de procesos que el proceso que hace la invocación
    - >0: esperar al hijo cuyo **pid** es el indicado
  - **waitpid**: argumento **options**
    - **WNOHANG**: retornar inmediatamente, aunque no haya terminado el hijo
  - Valores de retorno
    - El PID del hijo que ha finalizado (>0)
    - -1 si se recibe una señal o se da una condición de error (no existen hijos)
  - Errores
    - El proceso no tiene hijos

Procesos: **exit** y **wait**



Procesos: **exit** y **wait** (ejemplo)

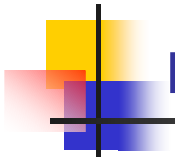
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main(void) {
   pid_t pid_hijo; int estado, x; long i, j;

   if ( (pid_hijo= fork()) == -1){ /* Código PADRE: Error */
      perror("Fallo al hacer fork()");
      exit(-1);
   } else if (pid_hijo == 0) { /* Código HIJO */
      fprintf(stdout, "PID hijo: %ld\n", (long) getpid()); fflush(stdout); sleep(2);
   } else { /* Código PADRE */
      if ( (x=wait(&estado)) != pid_hijo)
         fprintf(stdout, "PADRE: interrumpido por señal\n");
      else
         fprintf(stdout, "PID padre: %ld / PID hijo: %ld / estado hijo: %d\n",
                (long) getpid(), (long) pid_hijo, estado);
      fflush(stdout);
   }
   exit(0); /* Código PADRE e HIJO */
}
```


Procesos: **exec**

- **exec**: cambiar la imagen de memoria por la de un ejecutable

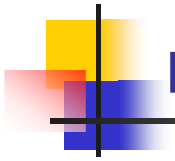
```
#include <sys/unistd.h>

void execl  (const char *path, const char *arg0, ...,
             const char *argn, (char *) 0 )

void execle (const char *path, const char *arg0, ...,
             const char *argn, (char *) 0, char *const envp[])

void execlp (const char *file, const char *arg0, ...,
             const char *argn, (char *) 0 )
```

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

void execv  (const char *path, const char *argv[])

void execve (const char *path, const char *argv[],
             const char *envp[])

void execvp (const char *file, const char *argv[])
```



Procesos: **exec**

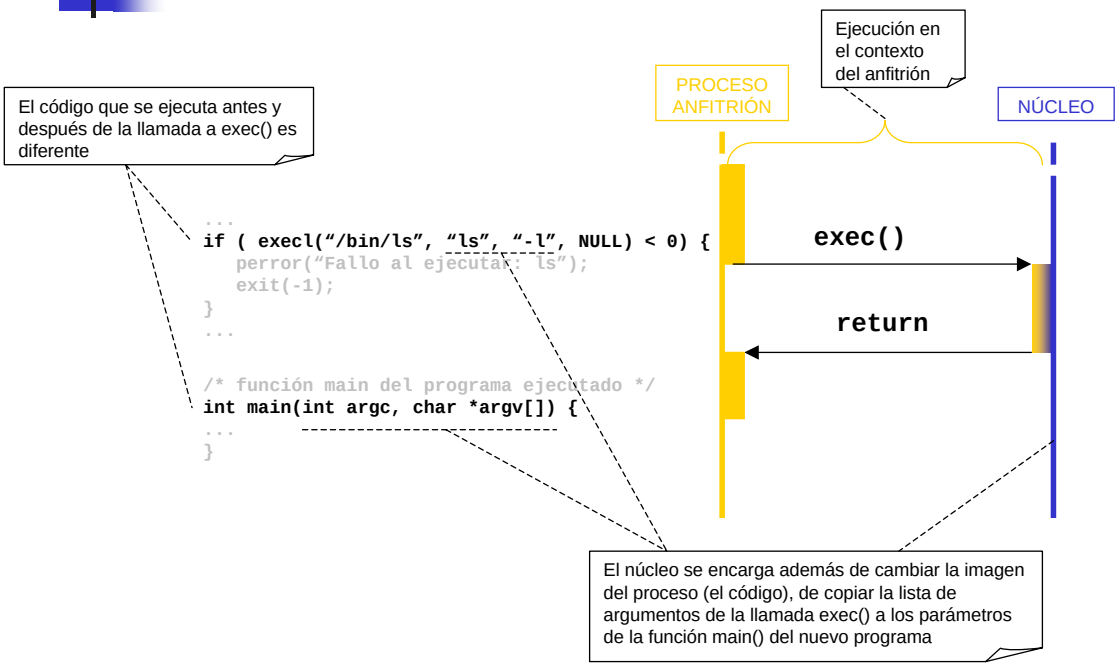
- Descripción
  - Cambia la imagen en memoria de un proceso (proceso anfitrión) por la definida en un fichero ejecutable
  - El fichero ejecutable se puede expresar dando exclusivamente su nombre o su ruta completa (ruta + nombre)
  - Algunos atributos del proceso anfitrión se conservan, en particular:
    - El manejo de señales, excepto las señales capturadas para las que se toma la acción por defecto
    - Los identificadores PID y PPID
    - Contadores de tiempo
    - Descriptores de archivo
    - El directorio de trabajo (CWD, *Current Work Directory*), el directorio raíz y la máscara de permisos por defecto (`umask`) para creación de archivos
  - Si el bit **`SETUID`** del fichero ejecutable está activado, la llamada a **`exec()`** establece como UID efectivo (**`eUID`**) al UID del propietario del fichero ejecutable
  - Ídem para el bit **`SETGID`**



Procesos: **exec**

- Descripción (continuación)
  - Posibles errores:
    - Fichero no existente o no ejecutable
    - No se tienen permisos
    - Argumentos incorrectos
    - Memoria o recursos insuficientes
  - Valor de retorno
    - Si EXEC retorna al programa que lo llamó es que ha ocurrido un error y el valor de retorno es –1

Procesos: **exec**

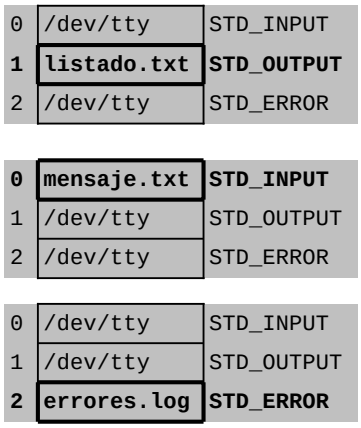


Procesos: **exec** (ejemplo)

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

int main(void) {
   pid_t pid_hijo; int estado, x; long i, j;

   if ( (pid_hijo= fork()) == -1){ /* Código PADRE: Error */
      perror("Fallo al hacer fork()");
      exit(-1);
   } else if (pid_hijo == 0) { /* Código HIJO */
      if ( execl("/bin/ls", "ls", "-l", NULL) < 0) {
         perror("Fallo al ejecutar: ls");
         exit(-1);
      }
   } else  /* Código PADRE */
      if ( (x=wait(&estado)) != pid_hijo) {
         fprintf(stdout, "PADRE: interrumpido por señal\n"); fflush(stdout);
      }
   exit(0); /* Código PADRE e HIJO, aunque el hijo nunca pasará por aquí */
}
```

Redirecciones

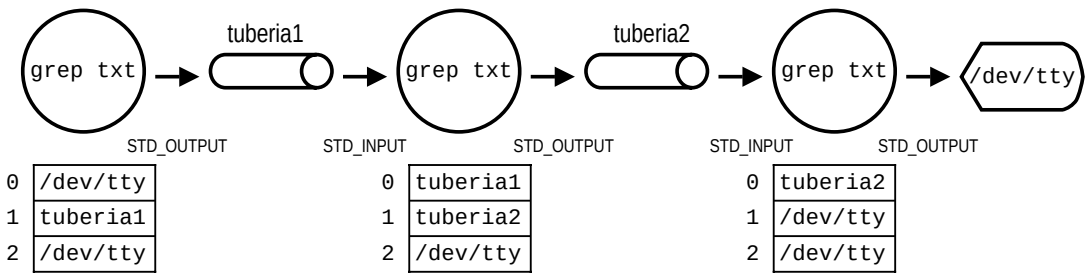
- Redirección de la entrada y la salida estándar
  - Unix/Linux definen tres descriptores con un significado especial, que suelen estar asociados al terminal
    - `0 STD_INPUT:` entrada estándar de datos
    - `1 STD_OUTPUT:` salida estándar de datos
    - `2 STD_ERROR:` salida estándar de errores
  - Redirección de la salida estándar
    - `$ ls –la > listado.txt`
  - Redirección de la entrada estándar
    - `$ write fdiaz < mensaje.txt`
  - Redirección de la salida de error estándar
    - `$ cc hilos.c 2> errores.log`



Tuberías

- Comunicación de procesos Unix/Linux
  - La comunicación entre procesos en Unix/Linux puede realizarse mediante tuberías
  - Las tuberías son un tipo especial de ficheros de capacidad limitada con acceso secuencial
  - Las tuberías pueden compartirse (entre procesos relacionados mediante la relación padre-hijo) gracias al mecanismo de herencia

```
$ ls | grep txt | sort
```



Redirecciones: dup y dup2

- **dup, dup2**: Duplicar un descriptor de fichero

```
#include <unistd.h>

int dup (int fd)
int dup2 (int fd, int copia_fd)
```

- Descripción
  - **dup**: retorna un descriptor de fichero cuya entrada es una copia de la que se corresponde con el parámetro **fd**. El descriptor que retorna es el descriptor disponible más bajo
  - **dup2**: cierra el descriptor que corresponde a **copia_fd** y luego copia la entrada del correspondiente al primer parámetro **fd** sobre la del segundo parámetro **copia_fd**
- Valor de retorno
  - Nuevo descriptor del fichero
  - -1 en caso de error (descriptor no válido o se supera el máximo de ficheros abiertos, **OPEN_MAX**)

Redirecciones: Ejemplo

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

#define NEW (O_WRONLY|O_CREAT|O_EXCL)
#define PERM (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

int redirigir_sal (const char *fich) {
  int fd;

  if ( (fd=open(fich, NEW, PERM)) == -1 )
    return -1;
  if ( dup2(fd, STDOUT_FILENO) == -1)
    return -1;
  close(fd);
  return 0;
}
```

```
int main (int argc, char * argv[]) {
  int fd_origen, fd_destino;

  if (argc < 3) {
    fprintf(stderr, "Uso: %s fich_salida
                     orden args\n", argv[0]);
    exit(1);
  }

  if (redirigir_sal(argv[1]) == -1) {
    fprintf(stderr, "No se puede redirigir salida
                      hacia %s\n", argv[1]);
    exit(1);
  }

  if ( execvp(argv[2], &argv[2]) < 0) {
    fprintf(stderr, "no se puede ejecutar %s\n",
                     argv[2]);
    exit(1);
  }
}
```

Tuberías: pipe

- **pipe**: Creación de una tubería (en memoria)

```
#include <unistd.h>

int pipe (int desc[2])
```

- Descripción
  - **pipe**: crea una estructura de datos en memoria, denominado tubería e inicialmente vacía, que se maneja como una cola FIFO de bytes
  - Las tuberías son un mecanismos de comunicación entre proceso Unix/Linux
  - Tras realizar la llamada, **desc[0]** es un descriptor de fichero válido para leer de la tubería (salida), mientras que **desc[1]** es un descriptor de fichero válido para escribir en la tubería (entrada)
  - La capacidad máxima de una tubería está limitada por la implementación,
  - Las tuberías junto con los descriptores de acceso son heredados por los procesos hijos y preservados tras la ejecución de **exec**
- Valor de retorno
  - 0 si no hay error
  - -1 en caso de error (array de descriptores **desc** no válido o se supera el máximo de ficheros abiertos, **OPEN_MAX**)

Tuberías

- Lectura de una tubería
  - Una vez abierta una tubería, por ejemplo, mediante **pipe**, puede leerse de la tubería, utilizando la llamada al sistema **read**, con una serie de consideraciones especiales:
    - Si existen bytes disponibles, se leen como mucho los solicitados en la llamada a **read**
    - Si en la tubería no hay datos, **read** suspende al proceso que lo invoca hasta que existan datos en la tubería (se puede evitar esta situación utilizando el modo de lectura no bloqueante **O_NONBLOCK** mediante la llamada al sistema **fnctl**)
    - Cuando no existe ningún descriptor de escritura asociada a la tubería (tanto del proceso lector como de cualquier otro proceso) **read** no suspende al proceso y retorna el valor 0, indicando así la condición de **final de datos en la tubería** (final de fichero)

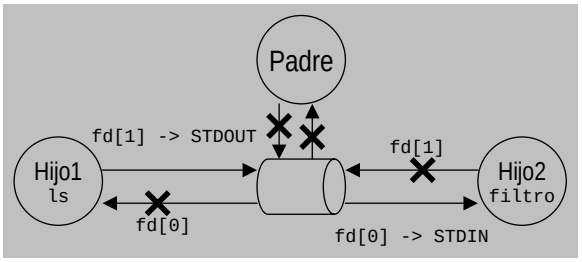
Tuberías

- Escritura en una tubería
  - Una vez abierta una tubería, por ejemplo, mediante **`pipe`**, puede escribirse en la tubería, utilizando la llamada al sistema **`write`**, con una serie de consideraciones especiales:
    - Si existe capacidad suficiente en la tubería para almacenar la cantidad de bytes solicitados en la llamada a **`write`**, éstos se almacenan en la tubería al final y, además, la operación puede considerarse atómica
    - Si no se pueden almacenar los datos de la escritura porque no hay suficiente sitio, el proceso escritor se bloquea hasta que exista disponibilidad de espacio
    - Si se intenta escribir en una tubería que no posee ningún descriptor de lectura asociado (tanto del proceso escritor como de cualquier otro proceso), el proceso que intenta escribir recibe la señal **`SIGPIPE`**. Este mecanismo facilita la eliminación automática de una cadena de procesos comunicados por tuberías cuando se aborta inesperadamente un componente de la cadena

Tuberías: Ejemplo 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (int argc, char *argv[]) {
  int i, fd[2], pid[2];
  /* Comprobación argumentos */
  if (argc < 2) {
    fprintf(stderr, "Uso: %s filtro\n", argv[0]);
    exit(1);
  }
  /* Creación recurso tubería, por el padre */
  pipe(fd);
  /* 1er hijo, ejecuta ls */
  if ( (pid[0]= fork()) == 0) {
    dup2(fd[1], STDOUT_FILENO);
    close(fd[0]);
    close(fd[1]);
    execlp("/bin/ls", "ls", NULL);
    perror("Hijo1: Fallo al hacer exec");
    exit(1);
  }
```

```
  /* 2do hijo, ejecuta filtro */
  if ( (pid[1]=fork()) == 0) {
    dup2(fd[0], STDIN_FILENO);
    close(fd[0]);
    close(fd[1]);
    execvp(argv[1], &argv[1]);
    perror("Hijo2: Fallo al hacer exec");
    exit(1);
  }
  /* El padre no interviene */
  close(fd[0]);
  close(fd[1]);
  for (i=0; i<2; i++)
    wait(pid[i], NULL);
  exit(0);
}
```



Tuberías: Ejemplo 2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#define  TAM 256

int main (int argc, char *argv[]) {
  int fd[2], status;
  char buffer[TAM];
  unsigned lon;
  if (argc != 1) {
    fprintf(stderr,“Uso: %s\n”,argv[0]);
    exit(1);
  }
  /* Creación recurso tubería */
  pipe(fd);
  switch (fork()) {
    case –1: fprintf(stderr, “Error al
                    hacer fork”);
             exit(1);
             break;
    case 0: /* Código del hijo */
      dup2(fd[1], STDOUT_FILENO);
      close(fd[0]); close(fd[1]);
      fprintf(stderr, “Hijo[%ld] va a escribir en la
                       tuberia\n”,(long)getpid());
      sprintf(buffer, “MENSAJE ESCRITO POR EL HIJO
                       [%ld]”, (long)getpid());
      lon=strlen(buffer)+1;
      if ( write(1, buffer, lon) != lon ) {
        fprintf(stderr, “Fallo al escribir el hijo\n”);
        exit(1);
      }
      break;
    default:/* Código del padre */
      dup2(fd[0], STDIN_FILENO);
      close(fd[0]); close(fd[1]);
      fprintf(stderr, “Padre[%ld] va a leer de la
                       tuberia\n”,(long)getpid());
      while ( (wait(&status)==-1) && (errno==EINTR) );
      if ( read(0, buffer, TAM) <= 0) {
        fprintf(stderr,”Fallo al leer el padre\n”);
        exit(1);
      }
    printf("Proceso PADRE (mensaje leido): %s\n", buffer);
    }
    exit(0);
}
```