



# Boletín 4- Procesos

Departamento de Lenguajes y  
Sistemas Informáticos

# Indice

---

1. Introducción (Procesos)
2. Identificadores de usuarios y procesos
3. La llamada `fork()`
4. Las llamadas `wait()` y `exit()`
5. La llamada `exec()`
6. La terminación de procesos
7. Comunicación entre procesos con tuberías

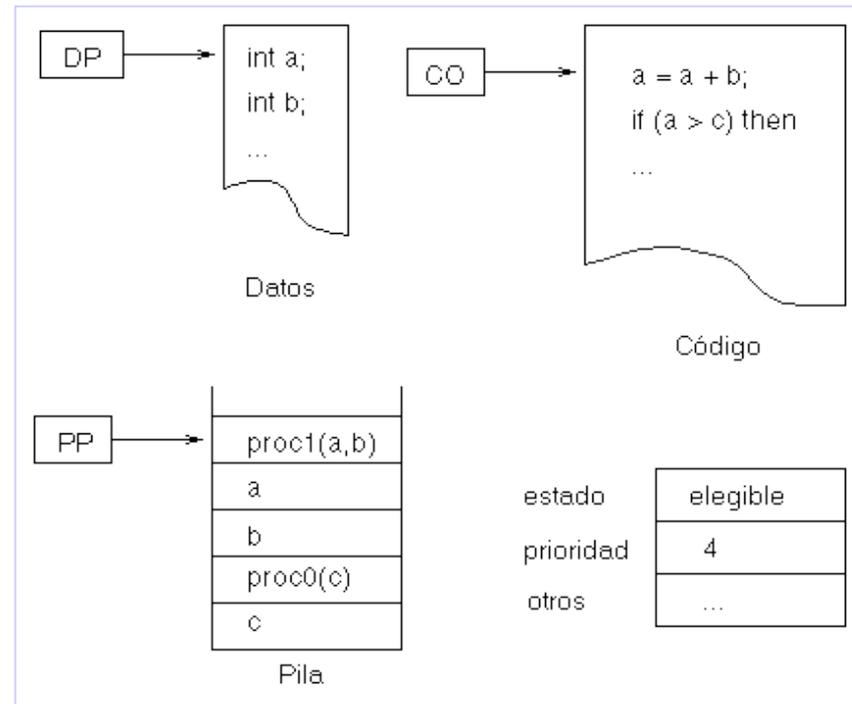
# Introducción (Procesos)

---

- -Definición: Proceso en UNIX.
- **Proceso**: es un programa en ejecución.
- Cada proceso dentro de su información tiene el código del programa, el área de datos, el program counter, registros e información adicional.
- El sistema operativo debe poder crear un proceso, destruirlo, suspenderlo, retomarlo, debe tener un mecanismo para retomar un proceso y para sincronizarlos, y mecanismos de concurrencia. Los procesos tienen una estructura jerárquica tipo árbol.

# Introducción (Procesos)

## El contexto de un proceso



Es el conjunto de las informaciones dinámicas que representan el estado de ejecución de un proceso (en qué punto de su ejecución se encuentra el proceso).

# Introducción (Procesos)

---

## Información de los procesos desde la línea de comando

```
murillo:/export/home/prof/lensis#>ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	0	0	0	Aug 19	?	0:01	sched
root	1	0	0	Aug 19	?	0:50	/etc/init -
root	2	0	0	Aug 19	?	0:00	pageout
root	3	0	1	Aug 19	?	2110:11	fsflush
root	442	1	0	Aug 19	?	0:00	/usr/lib/saf/sac -t 3
root	523	507	0	Aug 19	pts/3	0:19	/usr/dt/bin/dtsession
root	189	1	0	Aug 19	?	0:02	/usr/sbin/rpcbind
root	79	1	0	Aug 19	?	0:00	/usr/lib/sysevent
root	98	1	0	Aug 19	?	0:01	/usr/lib/picl/picld
root	237	1	0	Aug 19	?	0:01	/usr/sbin/cron
root	224	1	0	Aug 19	?	0:02	/usr/lib/automountd
root	236	1	0	Aug 19	?	2:53	/usr/sbin/syslogd -m 6

# Introducción (Procesos)

```
murillo:/export/home/prof/lensis#>ps -ef
  UID      PID  PPID  C    STIME TTY          TIME CMD
  root         0     0  0    Aug 19 ?           0:01 sched
  root         1     0  0    Aug 19 ?           0:50 /etc/init -
  root         2     0  0    Aug 19 ?           0:00 pageout
```

<b>Campo</b>	<b>Significado</b>
UID	ID del usuario propietario del proceso (entero)
PID	ID del proceso (entero)
PPID	ID del padre del proceso
C	Utilización del preprocesador de C para la administración de procesos (desuso)
STIME	Hora de comienzo
TTY	terminal de control
TIME	Tiempo acumulado de CPU
CMD	Nombre del comando

# Identificar usuarios y procesos

---

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
```

Devuelve

**pid\_t** es un entero largo con el ID del proceso llamante (**getpid**) o del padre del proceso llamante (**getppid**)

**uid\_t** es un entero con el ID del usuario propietario del proceso llamante.

En caso de error se devuelve -1.

# Creación de procesos: fork()

---

Todo proceso en UNIX ha sido creado por una instrucción `fork()`.

El nuevo proceso que se crea recibe **una copia del contexto de proceso del padre**. Los dos procesos **continúan su ejecución en la instrucción siguiente** al `fork()`.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Devuelve

Devuelve 0 al proceso hijo y el PID del hijo al padre.  
En caso se error se devuelve -1.

# Creación de procesos: fork()

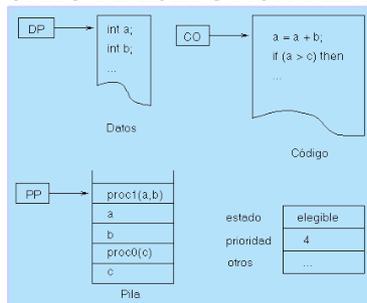
## PROCESO PADRE

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (void)
{
  if (fork() == 0)
  {
    printf("Este es el proceso hijo con padre
    %ld\n", (long)getppid());
  }
  else
  {
    printf("Este es el proceso padre con ID
    %ld\n", (long)getpid());
  }
  return 0;
}
```

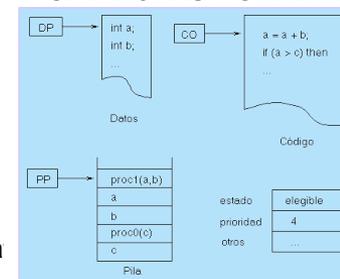
## PROCESO HIJO

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (void)
{
  if (fork() == 0)
  {
    printf("Este es el proceso hijo con padre
    %ld\n", (long)getppid());
  }
  else
  {
    printf("Este es el proceso padre con ID
    %ld\n", (long)getpid());
  }
  return 0;
}
```

## ENTORNO DE EJECUCIÓN DEL PADRE



## ENTORNO DE EJECUCIÓN DEL HIJO



# Las llamadas wait() y exit()

---

**exit()** termina la ejecución de un proceso

```
#include <stdlib.h>
void exit(int status);
```

## Parámetros

Status: entero que se devuelve al padre

## COMPORTAMIENTO:

- Si el proceso padre espera al hijo con un wait(), se le notifica la terminación y se le manda el valor de status.
- Si el proceso padre no espera, el hijo se convertirá en un proceso **huérfano**.

# Las llamadas wait() y exit()

---

## COMPORTAMIENTO:

- **Proceso Zombi:** se produce cuando un proceso se queda parado a la espera de que acabe su hijo y el hijo ya ha acabado. Se produce cuando el proceso padre no recoge el código de salida del hijo.
- **Proceso Huérfano:** proceso en ejecución cuando su padre ha finalizado. El nuevo identificador del padre (ppid) coincide con el identificador del proceso init (1).

# Las llamadas `wait()` y `waitpid()`

---

**`wait()`** y **`waitpid()`** suspende la ejecución del proceso actual hasta que un proceso hijo ha terminado, o hasta que se produce una señal.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

## Parámetros

**`stat_loc`**: puntero a entero con el valor del status devuelto por el hijo

**`pid`**: proceso a esperar.

**`options`**: opciones de terminación

# Las llamadas `wait()` y `exit()`

---

## `wait()` y `waitpid()`

**stat\_loc**: puntero a entero con el valor del status devuelto por el hijo

El estado puede ser evaluado con las siguientes macros

**WIFEXITED**(*status*)

es distinto de cero si el hijo terminó normalmente.

**WEXITSTATUS**(*status*)

evalúa los ocho bits menos significativos del código de retorno del hijo que terminó, que podrían estar activados como el argumento de una llamada a **exit()** o como el argumento de un **return** en el programa principal. Esta macro solamente puede ser tenida en cuenta si **WIFEXITED** devuelve un valor distinto de cero.

**WIFSIGNALED**(*status*)

devuelve true si el proceso hijo terminó a causa de una señal no capturada.

**WTERMSIG**(*status*)

devuelve el número de la señal que provocó la muerte del proceso hijo. Esta macro sólo puede ser evaluada si **WIFSIGNALED** devolvió un valor distinto de cero.

**WIFSTOPPED**(*status*)

devuelve true si el proceso hijo que provocó el retorno está actualmente parado; esto solamente es posible si la llamada se hizo usando **WUNTRACED**.

**WSTOPSIG**(*status*)

devuelve el número de la señal que provocó la parada del hijo. Esta macro solamente puede ser evaluada si **WIFSTOPPED** devolvió un valor distinto de cero.

# Las llamadas `exec()`

```
#include <unistd.h>
int  execl(const char *path,
char *argn, char * /*NULL*/);
int  execv(const char *path, char *const argv[]);
int  execlp(const char *file, co
const char *argn, char * /*NULL*
int  execvp(const char *file,
char *const argv[]);
int  execl(const char *path, const char *arg0, ...,
const char *argn, char * /*NULL*/, char *const envp[]);
int  execve(const char *path, char *const argv[], char *const
envp[]);
```

**execl** usa una lista indeterminada de parámetros.

Se usa cuando se sabe los parámetros de los procesos

**execv** usa una un array de punteros a char.

Se usa cuando **NO** se sabe los parámetros de los procesos

## Devuelve

Si cualquiera de las funciones **exec** regresa, es que ha ocurrido un error. El valor de retorno es -1, y en la variable global *errno* se pondrá el código de error adecuado.

# Las llamadas `exec()`

---

- **l** : Los argumentos para el programa (cadenas de texto) se incluyen uno a uno en los argumentos de la función, indicando el último mediante el puntero 0.
- **v** : Los argumentos para el programa se pasan mediante un array de punteros a las cadenas de texto que son los argumentos. El último elemento del array debe ser el puntero 0.
- **e** : Permite pasar las variables globales de entorno que deben definirse para la ejecución del nuevo programa.
- **p** : La búsqueda del programa se hará en todos los directorios contenidos en la variable de entorno PATH.

# Las llamadas `exec()`

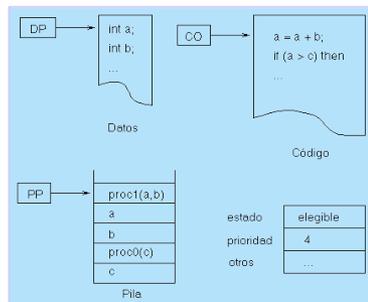
La familia de funciones **exec** reemplaza la imagen del proceso en curso con una nueva.

PROCESO PADRE **PID=228**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (void)
{
  if ( execl ("/bin/ps", "ps", "-fu", getenv
("USER"), 0) < 0)
  {
    fprintf(stderr, "Error en exec %d\n", errno);
    exit(1);
  }else
  {
    printf("Este es el proceso padre con ID
%d\n", (long) getpid());
  }
  return 0;
}
```

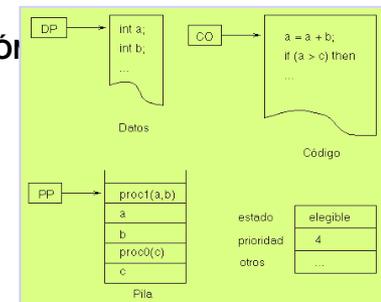
PROCESO HIJO **PID=228**

`/bin/ps -fu`



ENTORNO DE EJECUCIÓN DEL PADRE

ENTORNO DE EJECUCIÓN DEL HIJO



# El esquema fork()-exec()

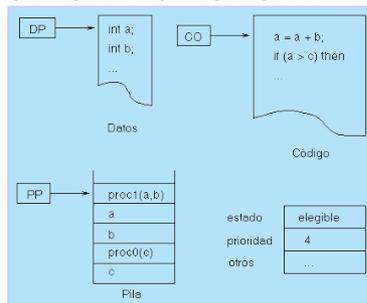
PROCESO PADRE PID=34

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (void)
{
if (fork() == 0)
{
execl("/bin/lS", "lS", "-la");
}
else
{
printf("Este es el proceso padre con ID
%d\n", (long) getpid());
}
return 0;
}
```

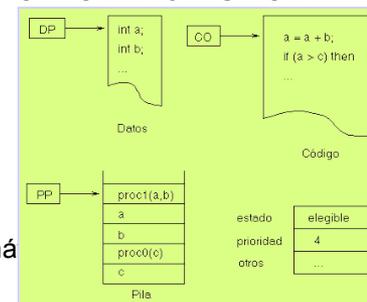
PROCESO HIJO PID=35

/bin/lS -la

ENTORNO DE EJECUCIÓN DEL PADRE



ENTORNO DE EJECUCIÓN DEL HIJO



# Terminación de procesos: atexit()

---

**Atexit()** La función `atexit` sirve para instalar un manejador de terminación de usuario. Se puede registrar más de una función. Las funciones así registradas se llaman en orden inverso al de su registro; no se puede pasar ningún argumento.

```
#include <stdlib.h>
#include <sys/wait.h>
int atexit(void (*MiFuncion) (void));
```

## Parámetros

**\*MiFuncion:** puntero a una función que se ejecute cuando se llame a `exit`.

## Devuelve

0 si todo ha ido correcto y -1 en caso de error, y en la variable global `errno` se pondrá el código de error adecuado.

# Tuberías

---

- ❑ Son el mecanismo más antiguo de IPC y lo proveen todos los sistemas Unix
- ❑ Proveen canal unidireccional para flujo de datos entre dos procesos
- ❑ Sólo puede ser usado entre procesos con un antecesor común que crea la pipe (heredando descriptores comunes).



# Tuberías

---

```
#include <unistd.h>
int pipe (int fildes[2]);
```

## Parámetros

**fildes** es una tabla con dos descriptores de ficheros: **fildes[0]** (descriptor de lectura) y **fildes[1]** (descriptor de escritura).

## Devuelve

Devuelve 0 en caso de éxito y -1 en caso de error, almacenando en errno el código del mismo.

- *Buffering se realiza en el kernel*
- *Normalmente el proceso que llama pipe luego llama a fork*
- *Dependiendo de dirección de flujo cada proceso cierra uno de los descriptores usando close*
- *Podrían existir múltiples lectores y escritores*

# Tuberías

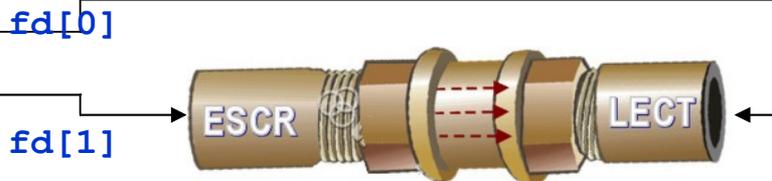
TABLA DE  
DESCRIPTORES  
DE FICHEROS

0
1
2
22
23



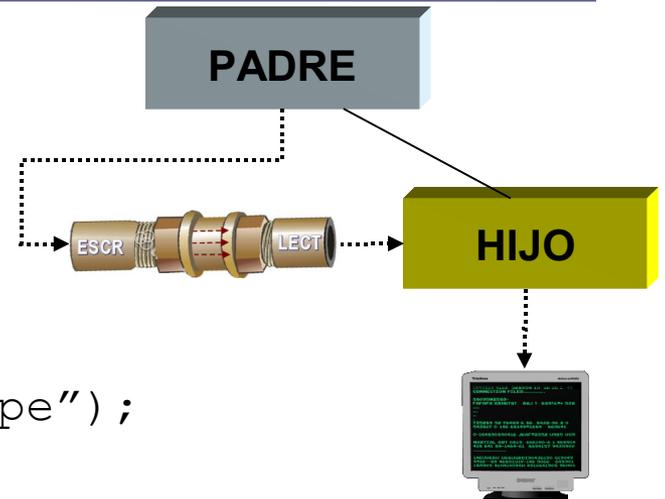
PROGRAMA/PROCESO

```
int main (void)
{
  int n, fd[2];
  pid_t pid;
  char linea[MAXLINEA];
  pipe (fd)
```

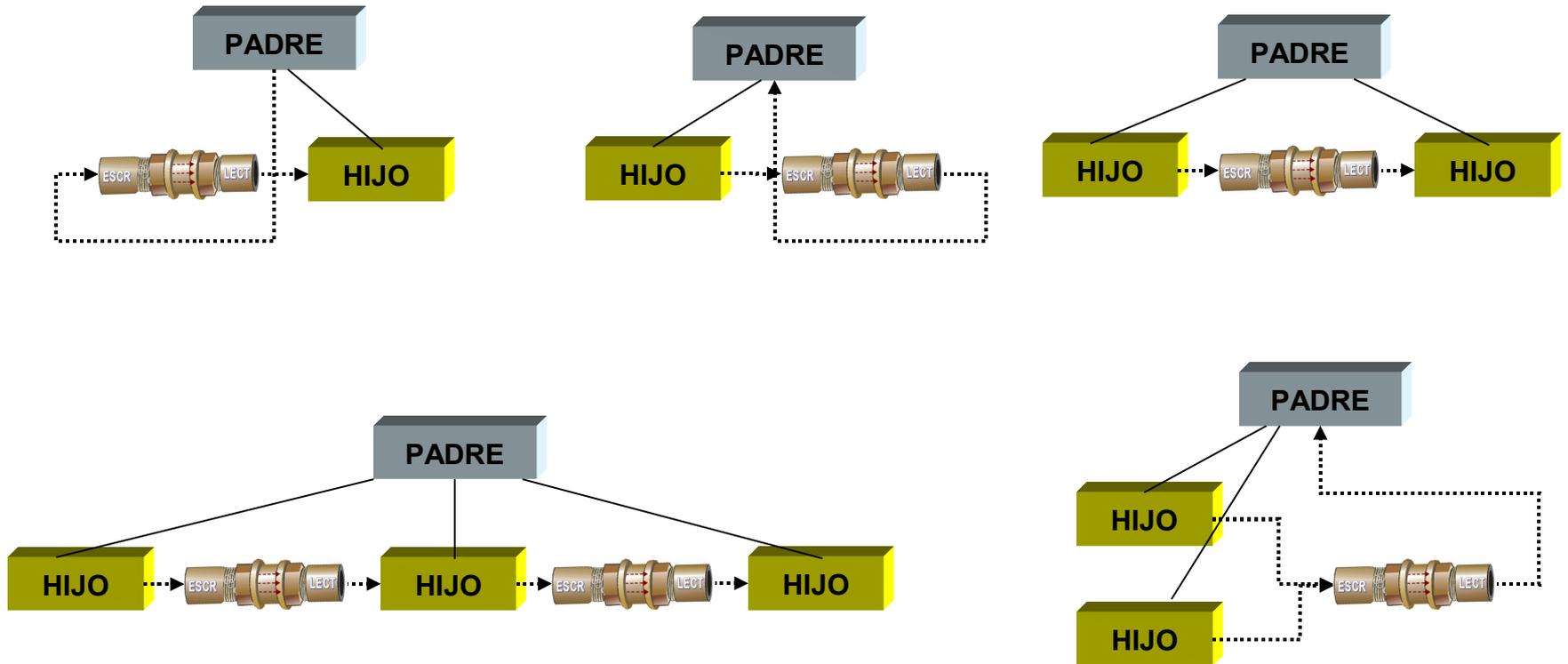


# Tuberías

```
int main (void)
{
    int n, fd[2];
    pid_t pid;
    char linea[MAXLINEA];
    if (pipe(fd) < 0)
        error_sys("error en creación de pipe");
    if ( (pid =fork()) < 0)
        error_sys("error en el fork");
    else if (pid > 0) { // soy el papa
        close (fd[0]); // papa no lee, sólo escribe
        write(fd[1], "Hola mundo\n");
    } else { // soy el hijo
        close (fd[1]); // hijo sólo leerá
        n = read(fd[0], linea, MAXLINEA);
        write(STDOUT_FILENO, linea, n);
    }
    exit(0);
}
```



# Tuberías



# Tuberías: dup() y dup2()

---

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

## Descripción

Duplica un descriptor de fichero.

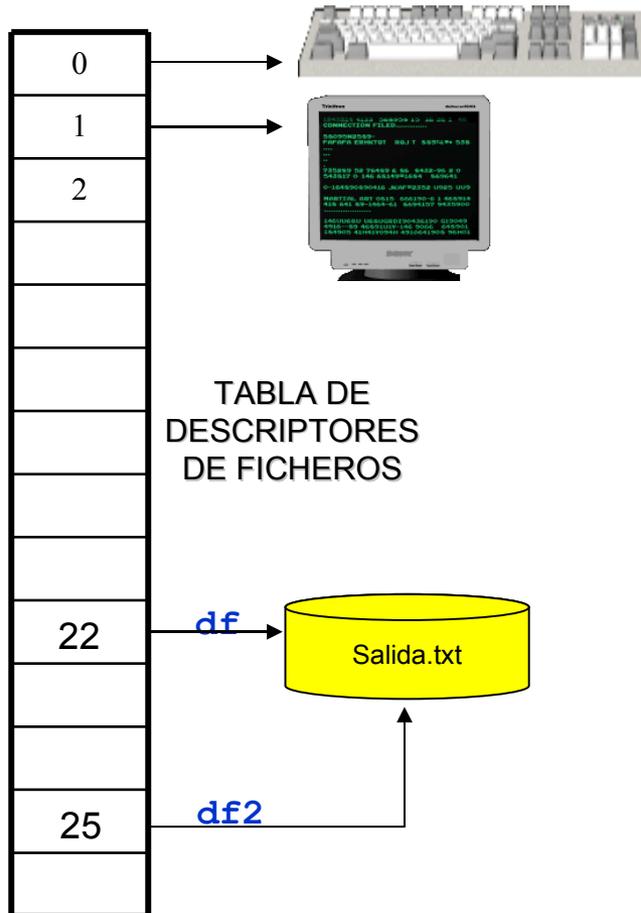
**dup** duplica el descriptor oldfd sobre la primera entrada de la tabla de descriptors del proceso que esté vacía.

**dup2** duplica del descriptor oldfd sobre el descriptor newfd. En el caso en que éste ya hiciera referencia a un fichero, lo cierra antes de duplicar.

## Devuelve

Ambas rutinas devuelven el valor del nuevo descriptor de fichero y  $-1$  en caso de error.

# Tuberías: dup() y dup2()



## PROGRAMA/PROCESO

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
int main (int argc, char *argv[])
{
    int df, df2;
    if (argc < 3)
    {
        printf ("Formato: %s  [opciones].\n",
                argv[0]);
        exit (1);
    }
    printf ("Ejemplo de redirección.\n");
    df = open ("salida.txt", O_CREAT);
    if (desc_fich===-1)
        exit(-1);
    df2=dup(df);
    close (df);
    ...
    exit (1);
}
```

# Tuberías: dup() y dup2()

---

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

## Descripción

**Duplica un descriptor de fichero.**

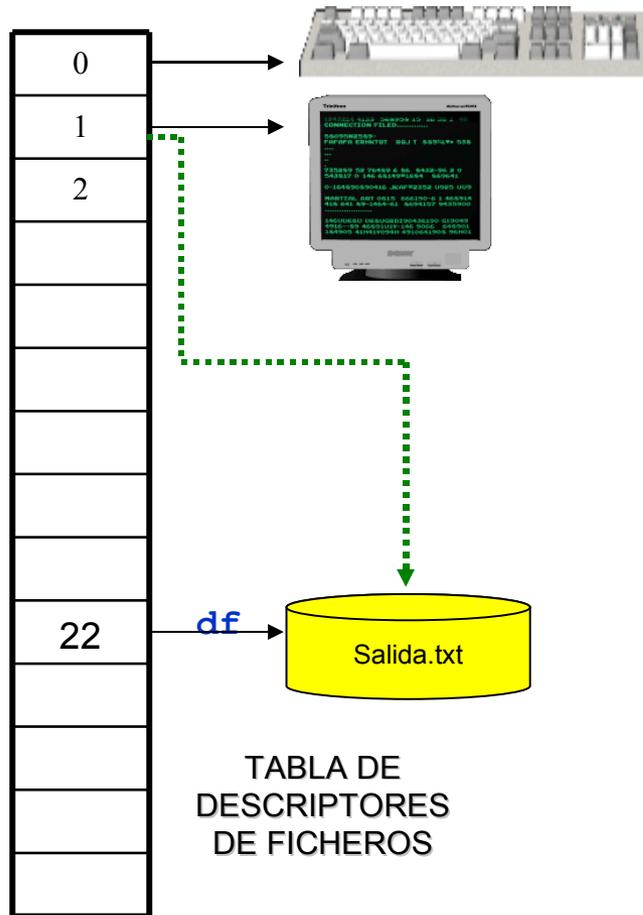
**dup** duplica el descriptor oldfd sobre la primera entrada de la tabla de descriptores del proceso que esté vacía.

**dup2** duplica del descriptor oldfd sobre el descriptor newfd. En el caso en que éste ya hiciera referencia a un fichero, lo cierra antes de duplicar.

## Devuelve

Ambas rutinas devuelven el valor del nuevo descriptor de fichero y  $-1$  en caso de error.

# Tuberías: pipe(), dup2() y descriptores de ficheros



## PROGRAMA/PROCESO

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
int main (int argc, char *argv[])
{
    int df;
    if (argc < 3)
    {
        printf ("Formato: %s [opciones].\n",
                argv[0]);
        exit (1);
    }
    printf ("Ejemplo de redirección.\n");
    df = open ("salida.txt", O_CREAT);
    if (desc_fich===-1)
        exit(-1);
    dup2 (df, STDOUT_FILENO);
    close (df);
    execvp (argv[2], &argv[2]);
    exit (1);
}
```