Prácticas de Sistemas operativos

David Arroyo Guardeño

Escuela Politécnica Superior de la Universidad Autónoma de Madrid

Tercera Semana: Comunicación entre procesos con Tuberías

1 *Entregas*

2 *Introducción*

3 *Tuberías*
- *Ejemplo 1*
- *Ejemplo 2*
- *Ejemplo 3*
- *Ejemplo 4*

Entregas

- Ejercicio 9
- Entrega antes de la sesión del próximo jueves 19 de febrero
- Examen de la primera práctica : jueves 19 de febrero

Comunicación entre procesos

- ✓ Formas elementales
  - ✗ Envío de señales
  - ✗ Uso de ficheros ordinarios
  - ✗ padre $\xrightarrow{ptrace}$ hijo
- ✓ Tuberías
- ✓ Facilidades IPC del Unix System V
  1. Semáforos
  2. Memoria compartida
  3. Colas de mensajes

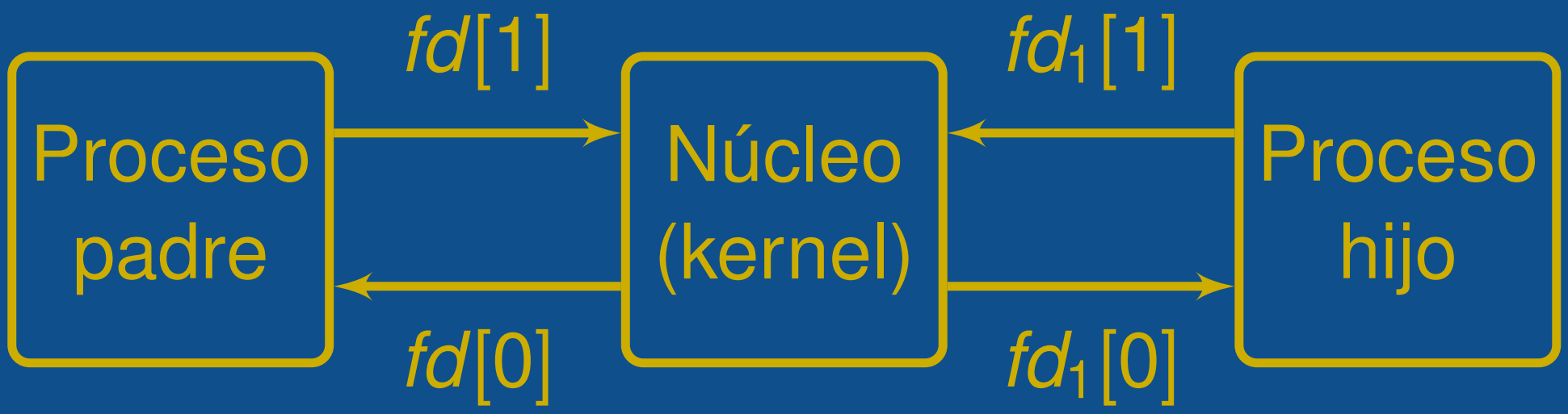
Tuberías

Canal de comunicación entre dos procesos: semi-dúplex

1. Tuberías con nombre: FIFOS
2. Tuberías sin nombre

Tuberías sin nombre I

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

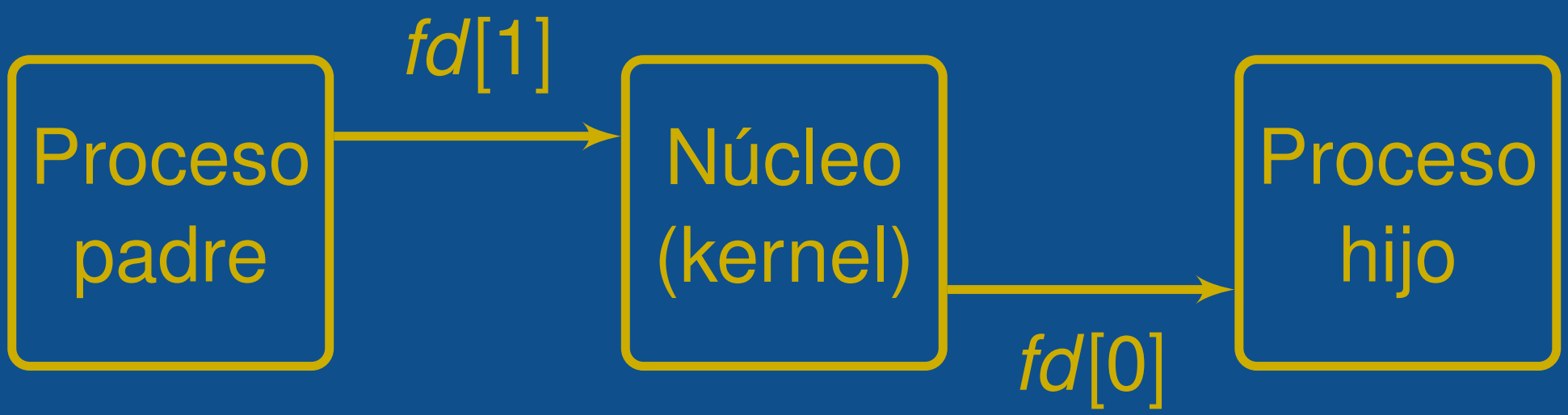


- Sólo el proceso que hace la llamada y sus descendientes pueden utilizarla
- *fildes*: descriptores de fichero
  - Leemos (*read*) de *fildes*[0] (fichero de sólo lectura)
  - Escribimos (*write*) en *fildes*[1] (fichero de sólo escritura)
- Tras *fork*/*exec* los hijos heredan los descriptores de ficheros
  - Abrimos la tubería en el padre
  - Padre e hijo comparten la tubería
- La tubería es gestionada por el núcleo
  - La dota de una disciplina de *acceso en hilera*

- ✓ Llamadas a *read* sobre la tubería no devolverán el control hasta que no haya datos escritos por otro proceso mediante *write*

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#define MAX 256
main()
{
  int tube[2];
  char message[MAX];
  if (pipe (tube) == -1){
    perror ("pipe");
    exit(1);
  }

  printf("Writing TO the file with descriptor #%d\n",tube[1]);
  write (tube[1], "TEST",5);
  printf("Reading FROM the file with descriptor #%d\n",tube[0]);
  read (tube[0], message,5);
  printf("READ DATA: \"%s \"\n",message);
  close (tube[0]);
  close (tube[1]);
  exit(0);
}
```

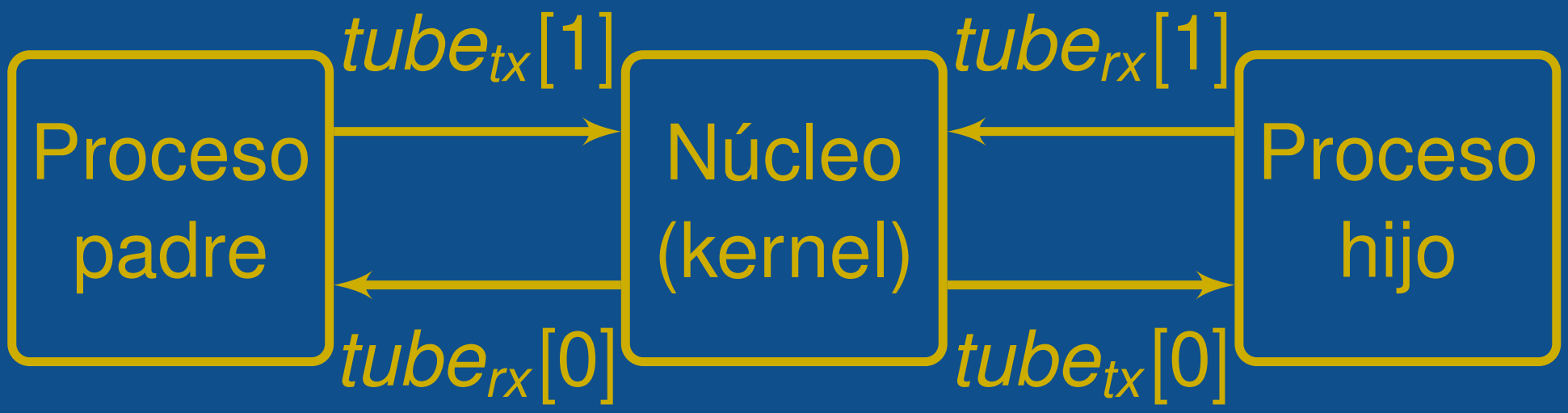
Ejemplo 2



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 256
main()
{
  int tube[2];
  int pid;
  char message [MAX];

  if (pipe (tube) == -1){
    perror ("pipe");
    exit(-1);
  }

  if ((pid = fork()) == -1) {
    perror ("fork");
    exit (-1);
  } else if (pid == 0){

    /*Cierra la tuberia de escritura porque no la va a usar*/
    close (tube[1]);
    while (read(tube [0],message,MAX)> 0 && strcmp (message, "END\n") !=
        0)
      printf("\nreceiver process. Message: %s\n",message);
```

```
    exit (0);

  } else {
    /* Cierra la tuberia de lectura porque no la va a usar */
    close (tube[0]);
    while (printf ("sender process. message: ") !=0 && fgets (message,
        sizeof( message), stdin) != NULL && write (tube [1], message,
        strlen (message) + 1) > 0 && strcmp (message, "END\n") !=0);

    exit(0);
  }
}
```

Ejemplo 3


Proceso padre
Núcleo (kernel)
Proceso hijo
$tube_{tx}[1]$
$tube_{rx}[1]$
$tube_{rx}[0]$
$tube_{tx}[0]$

Comunicación bidireccional I

```
main()
{
  int tube_tx[2], tube_rx[2];
  int pid;
  char message [MAX];

  if (pipe (tube_tx) == -1
  || pipe (tube_rx) == -1){
    perror ("pipe");
    exit(-1);
  }

  if ((pid = fork()) == -1) {
    perror ("fork");
    exit (-1);
  } else if (pid == 0){

    close (tube_tx[1]);
    close (tube_rx[0]);
    while (read(tube_tx [0],message,MAX)> 0
       && strcmp (message, "END\n") != 0){
      printf("\nreceiver process. Message: %s\n",message);
      strcpy(message,"READY");
      write (tube_rx [1], message, strlen(message) + 1);
```

Comunicación bidireccional II

```
    }

    exit (0);

  } else {
    close (tube_rx[1]);
    close (tube_tx[0]);
    while (printf ("sender process. message: ") !=0
       && fgets (message, sizeof( message), stdin) != NULL
       && write (tube_tx [1], message, strlen (message) + 1) > 0 &&
            strcmp (message, "END\n") !=0){
      do{
    read (tube_rx[0], message, MAX);
      } while (strcmp (message, "READY") != 0);
    }


    exit(0);
  }
}
```

Duplicación de descriptores de ficheros

- ✓ Si en la shell hacemos **2>&1**
  - ✗ Por ejemplo, ¿qué ocurre si hacemos lo siguiente?

```
$ls -la . fichero_inventado >
  results.log 2>&1
```

Duplicación de descriptores de ficheros

- ✓ Si en la shell hacemos **2>&1**
  - ✗ Por ejemplo, ¿qué ocurre si hacemos lo siguiente?

```
$ls -la . fichero_inventado >
  results.log 2>&1
```

  - ✗ La salida estándar de error (descriptor de fichero= 2) es redirigida al mismo sitio al que se envía la salida estándar (descriptor de fichero=1

Duplicación de descriptores de ficheros

- ✓ Si en la shell hacemos **2>&1**
  - ✗ Por ejemplo, ¿qué ocurre si hacemos lo siguiente?

```
$ls -la . fichero_inventado >
  results.log 2>&1
```

  - ✗ La salida estándar de error (descriptor de fichero= 2) es redirigida al mismo sitio al que se envía la salida estándar (descriptor de fichero=1
  - ✗ La shell efectúa el redireccionamiento de la salida estándar de error
    1. Duplica el descriptor de fichero 2
    2. Dicho descriptor ahora se refiere al mismo fichero con descriptor 1

- ✓ *fcntl* $\rightarrow$ *man fcntl*
- ✓ dup $\rightarrow$ man dup
- ✓ dup2 $\rightarrow$ man dup2

Si la shell sólo ha abierto los ficheros con descriptores 0,1, y el descriptor 2 se refiere al programa en ejecución y no existen otros descriptores

- ¿Qué hace la siguiente instrucción?

```
newfd = dup(1);
```

- ¿Cómo puedo asociar el descriptor 2 a nuestro duplicado?

Si la shell sólo ha abierto los ficheros con descriptores 0,1, y el descriptor 2 se refiere al programa en ejecución y no existen otros descriptores

- ✗ ¿Qué hace la siguiente instrucción?

```
newfd = dup(1);
```

- ✓ Crea el duplicado del descriptor 1 usando el fichero con descriptor 3
- ✗ ¿Cómo puedo asociar el descriptor 2 a nuestro duplicado?

Si la shell sólo ha abierto los ficheros con descriptores 0,1, y el descriptor 2 se refiere al programa en ejecución y no existen otros descriptores

- ✗ ¿Cómo puedo asociar el descriptor 2 a nuestro duplicado?
  - ✓ Primero cierro el fichero con descriptor 2 y luego llamo a dup

```
close(2);
newfd = dup(1);
```

$\Rightarrow$ En el ejemplo de antes bastaría hacer

*dup*2(1,2);

```c
#include <stdlib.h>
#include <stdio.h>
char *cmd1[] = { "/bin/ls", "-al", "/", 0 }; char *cmd2[] = { "/usr/bin/
    tr", "a-z", "A-Z", 0 };
void run1(int tube[]); void run2(int tube[]);

int main(int argc, char **argv)
{
   int pid, status;
   int tube[2];

   if(pipe(tube)==-1){
     fprintf(stderr,"Error en la linea %d del fichero %s\n",__LINE__,
         __FILE__);
     exit(EXIT_FAILURE);
   }
   run1(tube);
   run2(tube);
   close(tube[0]);
   close(tube[1]); /* Importante: hay que cerrar los dos descriptores de
       la tuberia */
   while ((pid = wait(&status)) != -1) /* Esperar a que hayan terminado
       todos los hijos */
     fprintf(stderr, "El proceso %d ha terminado y su estado de
         finalizacion es %d\n", pid, WEXITSTATUS(status));
   exit(EXIT_SUCCESS);
}
```

```c
void run1(int tube[]) /* Ejecutar la primera parte de la tuberia */
{
  int pid;
  switch (pid = fork()){
  case 0: /* hijo */
    dup2(tube[1], 1); /* Cerramos el descriptor 1, la salida estandar,
        que pasa a ser la salida de la tuberia */
    close(tube[0]); /* Este proceso no necesita el otro extremo de la
        tuberia */
    execvp(cmd1[0], cmd1); /* Ejecutar el primer comando, cmd1 */
    perror(cmd1[0]); /* Estamos aqui solo si ha habido algun fallo*/
  default: /* El padre no hace nada */
    break;
  case -1:
    perror("fork");
    exit(EXIT_FAILURE);
  }
}
```

```c
void run2(int tuberia[]) /* Se ejecuta la segunda parte de la tuberia */
{
  int pid;
  switch (pid = fork())
    { case 0: /* hijo */
    dup2(tuberia[0], 0); /* Este extremo de la tuberia pasa a ser la
        entrada estandar */
    close(tuberia[1]); /* Extremo de tuberia que no necesita este proceso
        */
    execvp(cmd2[0], cmd2); /* Se ejecuta este comando */
    perror(cmd2[0]); /* Estoy aqui solo si ha producido algun error*/
    default: /* El padre no hace nada */
      break;
    case -1:
      perror("fork");
      exit(1);
    }
}
```

Referencias

- Francisco M. Márquez. Unix, Programación Avanzada. Editorial: Ra-Ma. 3ª Edición. ISBN: 84-7897-603-5