Comunicación entre Procesos
(IPC)

Leandro Ezequiel Barrios

Departamento de Computación, FCEyN
Universidad de Buenos Aires

30 de Marzo de 2023

Temario

Temario

Funciones Útiles (1)

Importante

¡Complementar todo con el manual!

Procesos

- **int fork()**: Crea un nuevo proceso, copiando el actual. Retorna **0** en el proceso hijo, y el **PID** del proceso creado en el proceso padre.
- **void exit(int status)**: Termina el proceso actual utilizando valor de **status** como valor de retorno.

Funciones Útiles (2)

Pipes

- **int pipe(int descriptores[2])**: Crea **un pipe unidireccional**, el cual tiene un **extremo de escritura**[a], y un **extremo de lectura**[b]. Genera dos descriptores, representando a los extremos de lectura y escritura respectivamente, y los guarda en **descriptores**.
- **int dup2(int desc1, int desc2)**: si desc1 y desc2 son distintos, primero se elimina la referencia al objeto apuntado por desc2, y luego se apunta desc2 al mismo objeto que desc1.

[a] Por donde "entra" la información.
[b] Por donde "sale" la información.

Funciones Útiles (3)

Archivos / Descriptores

- **int open(char* path, int flags, ...)**: Abre el archivo indicado por `path`, retornando un *descriptor* que apunta a dicho archivo.
- **int close(int d)**: Cierra *para el proceso actual* el descriptor d pasado por parámetro.
- **int read(int d, void *b, size_t s)**: Lee **s** bytes del archivo apuntado por el descriptor **d**, y los escribe en el buffer **b**.
- **int write(int d, void *b, size_t s)**: Lee **s** bytes del buffer **b**, y los escribe en el archivo apuntado por el descriptor **d**.

Otros

- **printf(char* fmt, ...)**: función **variádica**[a] que toma un string de formato **fmt** y cero o más parámetros adicionales, y escribe el resultado en **stdout**.

[a] Permite un número arbitrario de parámetros.

Temario

Ejercicio 1: Los Simonzón

Dados y Doppelgänger...

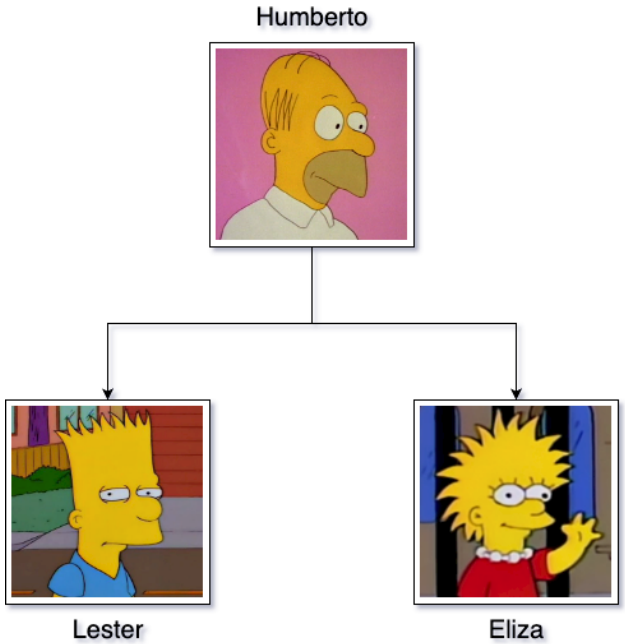
Lester y Eliza están muy aburridos. Para matar el tiempo, su ocurrente padre Humberto les propone un juego:

- Lester y Eliza tiran cada uno un dado, al mismo tiempo.
- Luego le cuentan a Humberto qué valor les dio el dado.
- Finalmente, Humberto se fija cuál es el número más grande y grita bien fuerte el nombre del ganador.

Ejercicio 1: Los Simonzón

Proponer un código en C que represente este juego respetando:

- Cada personaje debe estar representado por un proceso.
- Se debe respetar la genealogía.



- La comunicación entre procesos deberá realizarse con pipes.
- Los anuncios deberán realizarse imprimiendo a `stdout`.

Ejercicio 1: Pseudocódigo

Humberto:

- Crear a Lester.
- Crear a Eliza.
- Recibir número de Lester.
- Recibir número de Eliza.
- Anunciar al ganador.

Lester:

- Tirar el dado.
- Enviar resultado a Humberto.

Eliza:

- Tirar el dado.
- Enviar resultado a Humberto.

Ejercicio 1: Resolución

Nos tomamos unos minutos para pensarlo...

...y luego lo resolvemos en pizarrón.

Ejercicio 1: Solución

Importante

Todas las soluciones mostradas están simplificadas a efecto de poder incluirlas en la clase. Deberán tener **cuidado** y criterio a la hora de decidir si aplicar estas simplificaciones en sus resoluciones.

Es imprescindible que **lean minuciosamente el manual** para cada función que utilicen.

Ejercicio 1: Solución simplificada (1)

```c
#include <stdio.h>   // printf()
#include <stdlib.h>  // exit()
#include <unistd.h>  // fork() pipe() write() read()
#include "dado.h"    // tirar_dado()

// Constants 0 and 1 for READ and WRITE
enum { READ, WRITE };
// Constants 0 and 1 for LESTER and ELIZA
enum { LESTER, ELIZA };

// Variables globales
int pipes[2][2];
```

Ejercicio 1: Solución simplificada (2)

```
1  int main(int argc, char const* argv[]) {
2    // Creo los pipes
3    for (int i = 0; i < 2; i++) {
4      pipe(pipes[i]);
5    }
6
7    // Creo a Lester
8    int pid_lester = fork();
9    if (pid_lester == 0) {
10     lester();
11   }
12
13   // Creo a Eliza
14   int pid_eliza = fork();
15   if (pid_eliza == 0) {
16     eliza();
17   }
```

Ejercicio 1: Solución simplificada (3)

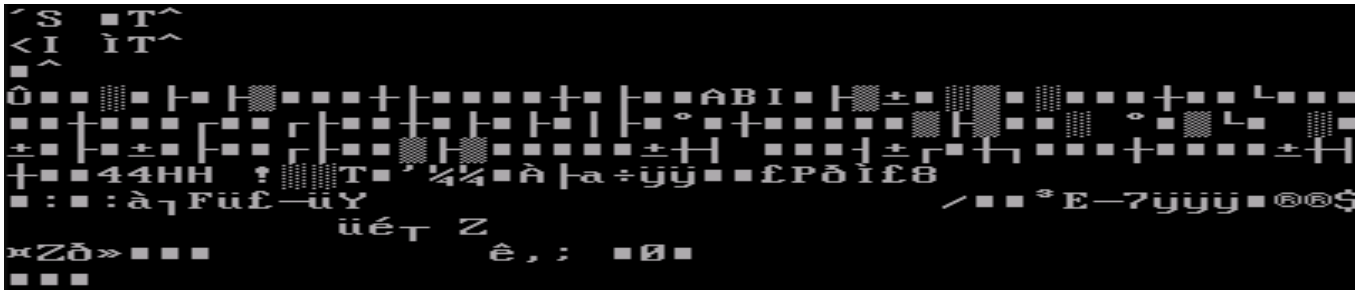
```
18    // Recibo el dado de Lester
19    int dado_lester;
20    read(pipes[LESTER][READ], &dado_lester, sizeof(dado_lester));
21
22    // Recibo el dado de Eliza
23    int dado_eliza;
24    read(pipes[ELIZA][READ], &dado_eliza, sizeof(dado_eliza));
25
26    // Anuncio al ganador
27    if (dado_lester == dado_eliza) {
28      printf("EMPATE");
29    } else if (dado_lester > dado_eliza) {
30      printf("GANADOR LESTER");
31    } else {
32      printf("GANADORA ELIZA");
33    }
34
35    return 0;
36  }
```

Ejercicio 1: Solución simplificada (4)

```
1  void lester() {
2    // Tiro el dado
3    int dado = tirar_dado();
4    // Le informo el resultado a Humberto
5    write(pipes[LESTER][WRITE], &dado, sizeof(dado));
6    exit(EXIT_SUCCESS);
7  }
8
9  void eliza() {
10   // Tiro el dado
11   int dado = tirar_dado();
12   // Le informo el resultado a Humberto
13   write(pipes[ELIZA][WRITE], &dado, sizeof(dado));
14   exit(EXIT_SUCCESS);
15 }
```

Temario

Ejercicio 2: Fantasma en el Shell



¡No se que toqué... y ahora no anda más el shell!

Ejercicio 18 de la Práctica 1: "Procesos y APIs del SO"

Escribir el código de un programa que se comporte de la misma manera que la ejecución del comando "**ls -al** | **wc -l**" en un intérprete de comandos.

No está permitido utilizar la función **system()**, y cada uno de los programas involucrados en la ejecución del comando deberá ejecutarse como un subproceso.

Ejercicio 2: Consideraciones preliminares

Primero, veamos qué hacen los comandos:

ls -al	Lista los archivos del directorio actual.
wc -l	Cuenta la cantidad de líneas del input.
cmd1 | cmd2	Ejecuta **cmd1**, y usa su output como input para ejecutar **cmd2**.
ls -al | wc -l	El comando completo cuenta la cantidad de archivos en el directorio actual.

Ejercicio 2: Pseudocódigo

Pasos a seguir...

- Creamos el pipe.
- Creamos los subprocesos.
- Queremos que todo lo que el subproceso 1 **escriba a stdout**, el subproceso 2 lo **lea desde stdin**. Para ello, usando dup2:
  - Conectamos **el descriptor de stdout** del subproceso 1 al **extremo de escritura** del pipe.
  - Conectamos **el descriptor de stdin** del subproceso 2 al **extremo de lectura** del pipe.
- Ejecutamos el comando "**ls -al**" en el subproceso 1.
- Ejecutamos el comando "**wc -l**" en el subproceso 2.

Ejercicio 2: Detectando el final del input

Todavía falta algo...

El subproceso 2 tiene que poder **detectar cuando el subproceso 1 terminó de escribir**. ¿Por qué en este caso es necesario esto?

Y, más importante, ¿cómo lo logramos?

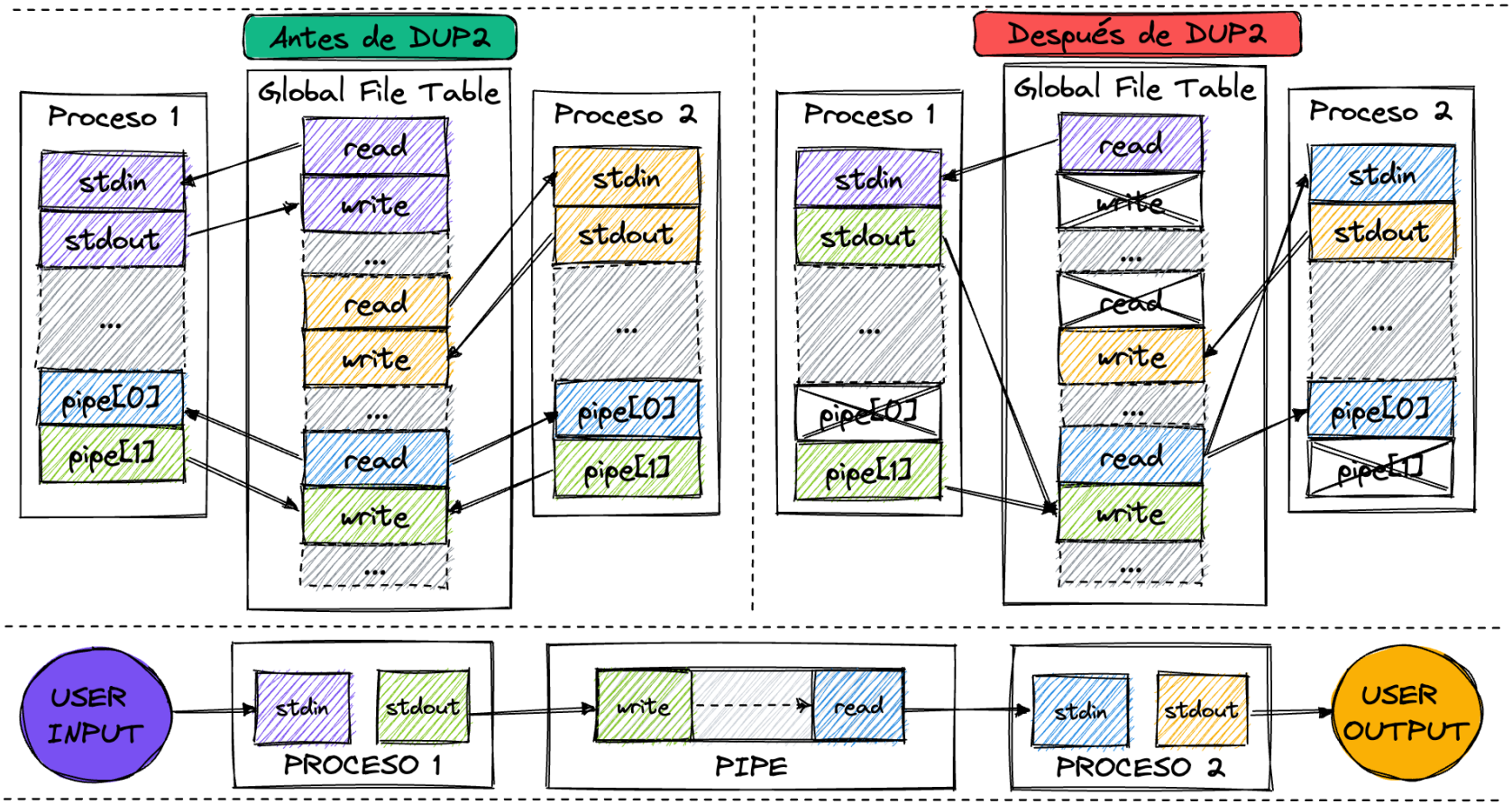
¡Haciendo un buen uso de los pipes!

Debemos cerrar los descriptores no utilizados en cada subproceso.

- El subproceso 1 jamás va a usar el extremo de lectura del pipe.
- El subproceso 2 jamás va a usar el extremo de escritura.

Cerrándolos previamente, nos aseguramos de que cuando el subproceso 1 termine de escribir a *"su stdout"* (que en realidad es el extremo de escritura del pipe), el sistema operativo detecte que **ya no existen más referencias hacia ese descriptor**, retornando un valor de **End-Of-File** cuando el subproceso 2 intente leer *"su stdin"* (que en realidad es el extremo de lectura del pipe).

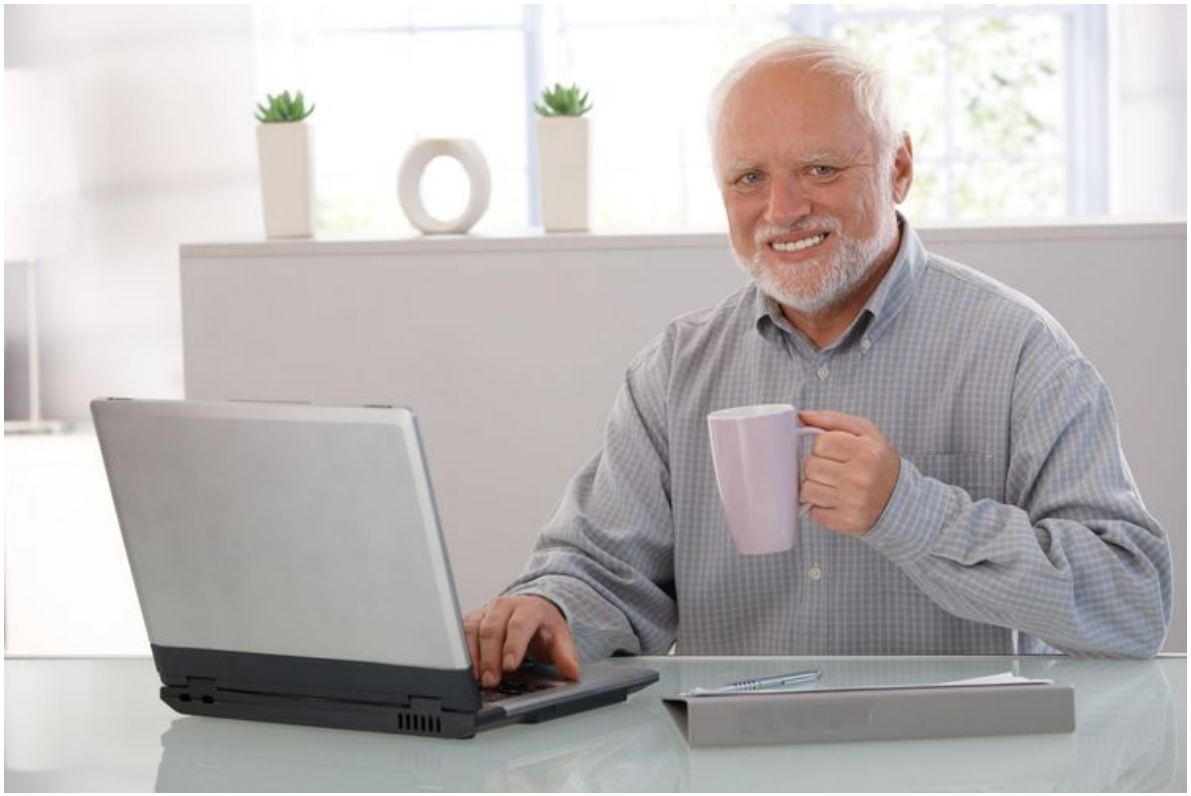
Ejercicio 2: Diagrama dup2



Esquema de comunicación entre los procesos utilizando un pipe y dup2

Ejercicio 2: Resolución

Nos tomamos unos minutos para pensarlo...



...y luego lo resolvemos en pizarrón.

Ejercicio 2: Solución

```c
void main() {
  int pipe_fd[2];
  pipe(pipe_fd);

  if(fork() == 0) {
    ejecutar_hijo_1(pipe_fd);
  }

  if(fork() == 0) {
    ejecutar_hijo_2(pipe_fd);
  }
}


void ejecutar_cmd(char* cmd, char* p) {
  execlp(cmd, cmd, p, NULL);
}
```

```c
void ejecutar_hijo_1(int pipe_fd[]) {
  // Cerrar lectura
  close(pipe_fd[0]);
  // Conectar escritura a stdout
  dup2(pipe_fd[1], STDOUT_FILENO);
  // Ejecutar programa
  ejecutar_cmd("ls", "-al");
}

void ejecutar_hijo_2(int pipe_fd[]) {
  // Cerrar escritura
  close(pipe_fd[1]);
  // Conectar lectura a stdin
  dup2(pipe_fd[0], STDIN_FILENO);
  // Ejecutar programa
  ejecutar_cmd("wc", "-l");
}
```

Bonus: Para pensar...

¿Qué tendríamos que **modificar al código** para lograr lo siguiente?

- Encadenar un tercer comando.
  - Por ejemplo: "**ls -al | grep e | wc -l**"
- Que el output se guarde en un archivo.
  - Por ejemplo: "**ls -al | wc -l > file.txt**"

Temario

Ejercicio 3: ¿Cien cabezas piensan mejor que una?

Ejercicio 3: Consigna

Escriba un programa que cuente los números pares en el rango de 2 a mil millones, utilizando para ello una función **bool esPar(long numero)** provista por la cátedra.
El programa dividirá el cálculo de forma equitativa entre varios procesos ejecutándose en paralelo. El número de procesos se especificará como parámetro en la línea de comandos.

Ejercicio 3: Consigna (2)

- El proceso primario creará el número especificado de procesos, y luego dividirá el rango de números en subrangos iguales y asignará un subrango a cada proceso secundario.
- Por ejemplo, si la cantidad de procesos p_i es 10, cada subrango tendrá aproximadamente cien millones de números.
  - p_1 calculará en el rango [2, 100.000.002).
  - p_2 calculará en el rango [100.000.002, 200.000.001).
  - ...
  - p_{10} calculará en el rango [900.000.002, 1.000.000.001).
- El proceso primario informará a cada proceso secundario el subrango que le corresponde usando pipes. Luego, los procesos secundarios deberán enviar sus recuentos al primario, también utilizando pipes.
- Finalmente, el primario deberá sumar todos los recuentos e imprimir el resultado total.

Ejercicio 3: Pseudocódigo

Proceso primario

- Crear los pipes
- Crear a los procesos secundarios
- Calcular e informar los rangos
- Recibir los resultados parciales
- Sumarlos e imprimir el total

Procesos secundarios

- Cerrar pipes
- Recibir los rangos
- Contar los pares
- Informar el resultado

Ejercicio 3: Solución simplificada (1)

```
1   #include <sys/wait.h>
2   #include <unistd.h>
3   #include <iostream>
4
5   #define RANGO_MIN 2L
6   #define RANGO_MAX 1000000001L
7
8   using namespace std;
9
10  int procesos;
11
12  bool esPar(long numero) {
13    return (numero & 1) == 0;
14  }
15
16  long contarPares(long minimo, long maximo) {
17    long cantidad = 0;
18    for (long i = minimo; i < maximo; ++i) {
19      if (esPar(i)) {
20        cantidad++;
21      }
22    }
23    return cantidad;
24  }
```

Ejercicio 3: Solución simplificada (2)

```
25  int main(int argc, char const* argv[]) {
26    procesos = atoi(argv[1]);
27    int pipes[procesos][2];
28
29    for (int i = 0; i < procesos; ++i) {
30      pipe(pipes[i]);
31    }
32
33    for (int i = 0; i < procesos; ++i) {
34      if (fork() == 0) {
35        ejecutarHijo(pipes, i);
36        return 0;
37      }
38    }
39
40    long cantidad = ((RANGO_MAX - RANGO_MIN) + (procesos - 1)) / procesos;
41    long inicio_rango = RANGO_MIN;
42    for (int i = 0; i < procesos; ++i) {
43      long fin_rango = min(inicio_rango + cantidad, RANGO_MAX);
44      write(pipes[i][1], &inicio_rango, sizeof(inicio_rango));
45      write(pipes[i][1], &fin_rango, sizeof(fin_rango));
46      inicio_rango += cantidad;
47    }
```

Ejercicio 3: Solución simplificada (3)

```
48    for (int i = 0; i < procesos; ++i) {
49      wait(NULL);
50    }
51
52    long resultado = 0;
53    for (int i = 0; i < procesos; ++i) {
54      long resultado_i;
55      read(pipes[i][0], &resultado_i, sizeof(resultado_i));
56      resultado += resultado_i;
57    }
58
59    cout << "Resultado total: " << resultado << endl;
60    return 0;
61  }
62
63  void ejecutarHijo(int pipes[][2], int i) {
64    long minimo;
65    read(pipes[i][0], &minimo, sizeof(minimo));
66
67    long maximo;
68    read(pipes[i][0], &maximo, sizeof(maximo));
69
70    long totalPares = contarPares(minimo, maximo);
71    write(pipes[i][1], &totalPares, sizeof(totalPares));
72  }
```

Temario

Recapitulando

Hoy vimos...

- Cómo crear una **topología** de procesos.
- Cómo usar **pipes** para comunicar procesos.
- Cómo usar **dup2** para duplicar un pipe.
- Por qué es importante **usar correctamente** los pipes.
- Para qué sirve la señal de **End-Of-File**.
- Cómo establecer una comunicación bidireccional:
  - Usando un solo pipe por cada par de procesos[a]
  - Usando dos pipes por cada par de procesos.

[a] **¡muy peligroso!**

¿Preguntas?