



Universitat d'Alacant  
Universidad de Alicante

**GRADO EN INGENIERÍA INFORMÁTICA**  
**CURSO ACADÉMICO 2025-2026**

# Sistemas Operativos

## GRUPO 8 - Práctica 1



JULIÁN HINOJOSA GIL - DNI 48795869-N

# Índice

<b>1. Gestión básica de procesos.</b>	<b>1</b>
1.1. Introducción . . . . .	1
1.2. Planteamiento . . . . .	1
1.3. Análisis de código fuente . . . . .	1
<b>2. Comunicación entre procesos: tuberías.</b>	<b>3</b>
2.1. Introducción . . . . .	3
2.2. Planteamiento . . . . .	3
2.3. Análisis de código fuente . . . . .	3
<b>3. Comunicación entre procesos: memoria compartida.</b>	<b>5</b>
3.1. Introducción . . . . .	5
3.2. Planteamiento . . . . .	5
3.3. Análisis de código fuente . . . . .	6

## 1. Gestión básica de procesos.

### 1.1. Introducción

Se pide realizar un programa en C que cree un árbol de procesos con la siguiente estructura:

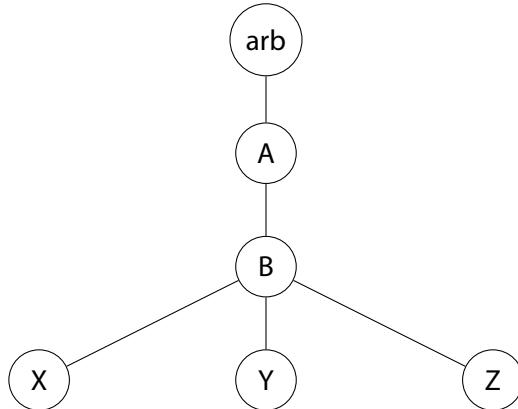


Figura 1: Árbol de procesos

El programa tambien recibira dos argumentos, el primero servira para determinar si se ejecuta el comando `pstree -l 1` (si el argumento es A o B) o `ls -la` (si el argumento es X o Y) y el segundo argumento sera el tiempo en segundos que esperara Z para mandar las señales.

### 1.2. Planteamiento

El código se plantea implementando una jerarquía de procesos mediante llamadas a `fork()`, creando 6 procesos como se muestra en la figura 1. La comunicación entre procesos se realiza mediante señales, usando `SIGUSR1` para ejecutar los comandos y `SIGUSR2` para despertar a los procesos que estén en estado de espera.

Tambien se definen dos funciones void manejadoras de señales para ejecutar los comandos `pstree -l 1` y `ls -la` y en cada proceso con la llamada a `signal()` se asignan a estos mismos, donde `SIGUSR1` es para el manejador de los comandos y `SIGUSR2` es para el manejador de despertar.

El proceso Z coordina el envío de señales a los demás procesos para que ejecuten los comandos o terminen su espera, dependiendo del argumento recibido, por tanto el flujo de ejecución de Z sigue un patrón, donde primero espera el tiempo definido por el segundo argumento, luego envia la señal `SIGUSR1` al proceso correspondiente para que ejecute el comando, espera un segundo mediante `alarm()` y finalmente envia la señal `SIGUSR2` a los procesos que estén en estado de espera.

Todo esta sincronizado mediante `pause()` y `wait()`, donde los procesos padres esperan con `wait(NULL)` a que sus hijos terminen y los procesos hijos esperan con `pause()` a que les llegue la señal para ejecutar el comando o terminar su espera. De esta forma se consigue garantizar el orden predecible desde las hojas del árbol hasta la raíz.

### 1.3. Análisis de código fuente

El ejercicio a realizar trata sobre la simulación de un árbol de procesos, tal y como se muestra en la figura 1 y dependiendo de los argumentos recibidos por el programa se ejecutarán los siguientes comandos:

```

1 // Maneja la señal SIGUSR1, que ejecuta el comando pstree
2 void pstree_command() {
3     pid_t pid;
4     if ((pid = fork()) == 0)
5         execlp("pstree", "pstree", "-l 1", NULL);
6     else
7         wait(NULL);
8 }
9
10 // Maneja la señal SIGUSR1, que ejecuta el comando ls
11 void ls_command() {
12     pid_t pid;
13     if ((pid = fork()) == 0)
14         execlp("ls", "ls", "-la", NULL);
15     else
16         wait(NULL);
17 }
```

Listing 1: Manejo de señales SIGUSR1 y SIGUSR2

Como se indica estas funciones pasaran a ser manejadores de las señales definidas por usuario **SIGUSR1**. Es importante destacar que dentro de los manejadores se utiliza **fork()** seguido de **execlp()** para ejecutar los comandos en un proceso hijo separado. Esto es necesario porque **execlp()** reemplaza completamente la imagen del proceso actual, por lo que si se ejecutara directamente en el manejador, el proceso original se perdería. Al crear un proceso hijo con **fork()**, permitimos que el padre continúe su ejecución normal tras ejecutar el comando en el hijo, esperando su finalización con **wait()**. La señal SIGUSR2 simplemente sirve para despertar a los procesos a través de un manejador vacío:

```

1 signal(SIGUSR1, pstree_command); // En el caso de A y B
2 signal(SIGUSR1, ls_command); // En el caso de X e Y
```

Listing 2: Registro de manejadores de señales

Para la creación de los procesos X, Y y Z simplemente usaremos un bucle for y accederemos a su ejecución con un switch en base a la iteración en cuestión, ademas previamente a entrar al switch guardaremos los pids de los procesos X e Y, ya que Z al ser hermanos de estos dos no podra acceder a ellos directamente:

```

1 process = fork(); // Se crea el proceso B
2 ...
3 for (int i = 1; i <= 3; i++) {
4     process = fork(); // Se crea los procesos X, Y y Z
5     ...
6     // Se utiliza despues para mandarles señales, ya que Z no conoce los Pid
7     // de X e Y
8     if (process != 0) {
9         // El proceso B guarda el Pid de X e Y
10        if (i == 1)
11            pid_X = process;
12        if (i == 2)
13            pid_Y = process;
14    } else { // Esta en ejecucion los 3 (X, Y, Z)
15        switch (i) {
16            ...
17        }
18    }
19}
```

Listing 3: Creación de procesos

Y por último mediante *kill()* mandamos las señales respectivas según los argumentos del programa, primero el comando a ejecutar y luego a despertar el resto de procesos dormidos para que acaben, es la misma estructura para A y B como para X e Y:

```

1 // Control de Argumentos
2 if (strcmp(argv[1], "A") == 0 || strcmp(argv[1], "B") == 0) {
3     if (strcmp(argv[1], "A") == 0) {
4         kill(pid_A, SIGUSR1); // Se manda la señal a A ya que es el argumento
5         recibido
6     } else {
7         kill(pid_B, SIGUSR1); // Se manda la señal a B ya que es el argumento
8         recibido
9     }
10
11     signal(SIGALRM, end_sleep); // Cuando suene la alarma, se despierta
12     alarm(1);
13     pause();
14
15     kill(pid_Y, SIGUSR2); // Ya que habia hecho pause antes
16     kill(pid_X, SIGUSR2); // Se manda la señal a X e Y para que
17     // terminen su sleep
18
19     if (strcmp(argv[1], "A") == 0) {
20         kill(pid_B, SIGUSR2); // Se manda la señal a B para que termine su
21         sleep
22     } else {
23         kill(pid_A, SIGUSR2); // Se manda la señal a A para que termine su
24         sleep
25     }
26 }
```

Listing 4: Control de argumentos y envío de señales

## 2. Comunicación entre procesos: tuberías.

### 2.1. Introducción

Se pide realizar un programa en C que, recibiendo dos argumentos (archivo de entrada y archivo de salida), copie el contenido del primer archivo en el segundo usando tuberías para simular la comunicación entre procesos.

### 2.2. Planteamiento

Se crea un array de enteros de tamaño 2 para almacenar los descriptores de las dos puntas de la tubería (lectura y escritura). Se crea una tubería con la llamada a *pipe()*. A continuación, se hace un *fork()* para crear un proceso hijo. El proceso padre se encarga de leer el archivo de entrada y escribir su contenido en la tubería, mientras que el proceso hijo lee de la tubería y escribe en el archivo de salida. Ambos procesos cierran los extremos de la tubería que no utilizan para evitar bloqueos.

La lecturas y escrituras se realizan de char en char hasta llegar al final del archivo (EOF), utilizando bucles *while* que leen y escriben un carácter a la vez.

### 2.3. Análisis de código fuente

El ejercicio pretende copiar los contenidos de un archivo a otro nuevo, todo esto como argumentos del programa. Creamos una arreglo de enteros que contenga los dos descriptores de pipe y creamos las tuberías con *pipe()*:

```

1 int pipes[2], file_in, file_out;
2 pipe(pipes);

```

Listing 5: Creación de pipes

Crearemos un fork para poder demostrar la comunicación entre padre e hijo por tuberías. Para la ejecución del padre cerraremos el pipe de lectura y abriremos el archivo origen como lectura, con un bucle while lo leeremos y a su vez escribiremos en el pipe de escritura:

```

1 if (fork() != 0) { // Ejecucion del padre
2     char read_char;
3     close(pipes[0]); // Cierra el extremo de lectura del pipe
4
5     file_in = open(argv[1], O_RDONLY); // Abre el archivo de entrada
6
7     // Leemos el archivo hasta que devuelva 0 y por tanto EOF
8     while ((read(file_in, &read_char, sizeof(char))) > 0) {
9         write(pipes[1], &read_char, sizeof(char)); // Escribimos en
10            // el pipe de escritura
11    }
12
13    // Cerramos los descriptores
14    ...
15 }

```

Listing 6: Proceso padre - lectura del archivo y escritura en pipe

Realizaremos el proceso inverso para el hijo, que se encarga de escribir el contenido en el archivo que hay como segundo argumento del programa:

```

1 else {
2     char write_char;
3
4     close(pipes[1]); // Cierra el extremo de escritura del pipe
5
6     file_out = creat(argv[2], 0666); // Crea el archivo de salida con
7           // permisos de lectura y escritura
8
9     // Al igual que en el padre, leemos hasta EOF
10    while((read(pipes[0], &write_char, sizeof(char))) > 0) {
11        write(file_out, &write_char, sizeof(char)); // Escribimos en el
12           // archivo de salida
13    }
14
15    // Cerramos los descriptores
16    ...
17 }

```

Listing 7: Proceso hijo - lectura del pipe y escritura en archivo

### 3. Comunicación entre procesos: memoria compartida.

#### 3.1. Introducción

Se pide realizar un programa en C que cree un árbol de procesos en base a dos argumentos X e Y, creando por un lado hijos de forma vertical y de forma horizontal respectivamente, usando memoria compartida para almacenar los pids y así poder mostrar la relación de los procesos.

#### 3.2. Planteamiento

El programa crea una estructura de procesos en dos dimensiones: una cadena vertical de X procesos y una horizontal de Y procesos, como se muestra en la figura 2. El proceso padre original (hijos) crea primero una cadena vertical de X procesos, donde cada hijo se convierte en padre del siguiente, formando la rama vertical. El último proceso de esta cadena vertical es el que luego crea Y procesos hermanos de forma horizontal.

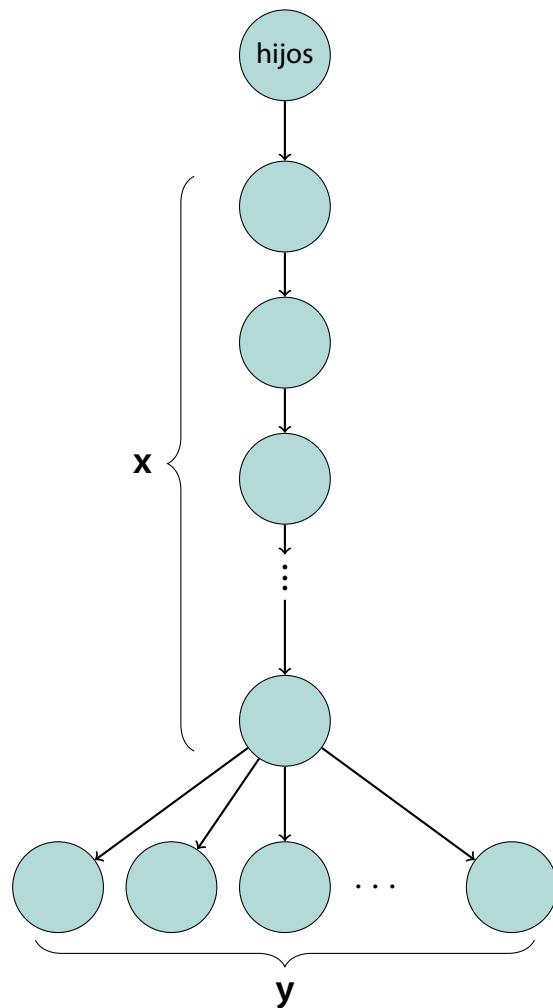


Figura 2: Estructura del árbol de procesos con ramas vertical (x) y horizontal (y)

Para lograr esto, se utiliza memoria compartida que permite a todos los procesos acceder a los PIDs de sus antecesores. Se crean dos segmentos de memoria compartida: uno para almacenar los PIDs de la cadena vertical (X) y otro para la cadena horizontal (Y). Esto permite que cada proceso pueda imprimir su genealogía completa.

### 3.3. Análisis de código fuente

Esta vez se trata de crear un árbol variable de procesos en base a los argumentos X e Y, creando por un lado hijos de forma vertical y de forma horizontal respectivamente.

Lo haremos usando memoria compartida para almacenar los pids y así poder mostrar la relación de los procesos. Empezaremos resaltando primero la creación de la memoria compartida una para X y otra para Y y asignándole sus respectivos punteros:

```

1 // Crear memoria compartida para X e Y
2 shm1 = shmget(IPC_PRIVATE, sizeof(int)*(x_childs+1), IPC_CREAT|0666);
3 shm2 = shmget(IPC_PRIVATE, sizeof(int)*y_childs, IPC_CREAT|0666);
4 // Punteros a la memoria compartida
5 X_pointer = (int *)shmat(shm1, 0, 0);
6 Y_pointer = (int *)shmat(shm2, 0, 0);

```

Listing 8: Creación de memoria compartida

Se guarda el pid del padre origen en la primera posición de X y se crea la rama vertical en base al primer argumento:

```

1 // Crear la rama vertical de x_childs procesos
2 for(i = 1; i <= x_childs; i++) {
3     process = fork();
4
5     if(process != 0) // El padre sale para que el hijo continue la cadena
6         break;
7     else {
8         printf("Soy el subhijo %d, mis padres son: ", getpid());
9
10        // Imprimir los PIDs de los padres desde el array X
11        for (child = 0; child < i; child++) {
12            printf("%d", X_pointer[child]);
13            if (child != i-1) printf(", ");
14        }
15        printf("\n");
16
17        // Guardar el PID del hijo actual en el array X
18        X_pointer[i] = getpid();
19    }
20}

```

Listing 9: Creación de rama vertical

Repetiremos el mismo proceso con y para crear la rama horizontal

```

1 for (i = 1; i <= y_childs; i++) {
2     process = fork();
3
4     if (process == 0) {
5         Y_pointer[i-1] = getpid();
6         sleep(20);
7         break;
8     }
9
10    // El padre espera a que terminen todos los hijos de Y
11    if (i == y_childs + 1)
12        for (i = 1; i <= y_childs; i++)
13            wait(NULL);
14}

```

Listing 10: Creación de rama horizontal