

## **Documentación Taller Procesos**

Guillermo Andrés Aponte Cárdenas

Daniel Felipe Ramirez Vargas

Pontificia Universidad Javeriana

Sistemas Operativos

Bogotá, 2025

## **Objetivos**

### **General**

Analizar el funcionamiento de los procesos en un sistema operativo tipo Linux mediante la implementación y sincronización de procesos padre, hijo y nieto, utilizando llamadas al sistema como `fork()`, `pipe()`, `wait()` y `usleep()` para comprender la comunicación y coordinación entre procesos concurrentes.

### **Específicos**

Implementar la creación de múltiples procesos usando la llamada al sistema `fork()` y analizar su comportamiento.

Emplear pipes para establecer comunicación entre procesos relacionados (padre–hijo–nieto).

Compilar y organizar los módulos del programa mediante un Makefile para automatizar la construcción y limpieza del proyecto.

## Resumen

En este taller se implementó un programa en lenguaje C que aplica los conceptos de creación y comunicación entre procesos, utilizando las funciones `fork()`, `pipe()` y `wait()`. El programa recibe como argumentos dos archivos que contienen arreglos de enteros y sus respectivos tamaños, los cuales son leídos y almacenados dinámicamente en memoria. A partir del proceso principal se genera una jerarquía conformada por un padre, un primer hijo (que a su vez crea un nieto) y un segundo hijo. Cada proceso cumple una función específica, el nieto calcula la suma del primer arreglo (`sumaA`), el segundo hijo calcula la suma del segundo arreglo (`sumaB`), el primer hijo suma ambos resultados obteniendo la suma total y el proceso padre lee los dos archivos enviados como parámetro dejando así a sus sucesores disponibilidad para leer el contenido de ambos, además, recibe e imprime todos los valores. La comunicación entre procesos se realiza mediante dos pipes. El primer pipe le permite al segundo hijo y al nieto, enviar la suma de los arreglos al primer hijo. El segundo pipe permite al primer hijo, segundo hijo y al nieto, enviar sus resultados al proceso padre. Se incluyen manejos de errores con `perror()` y liberación de memoria dinámica al finalizar la ejecución (Ver ilustración Foto 1).

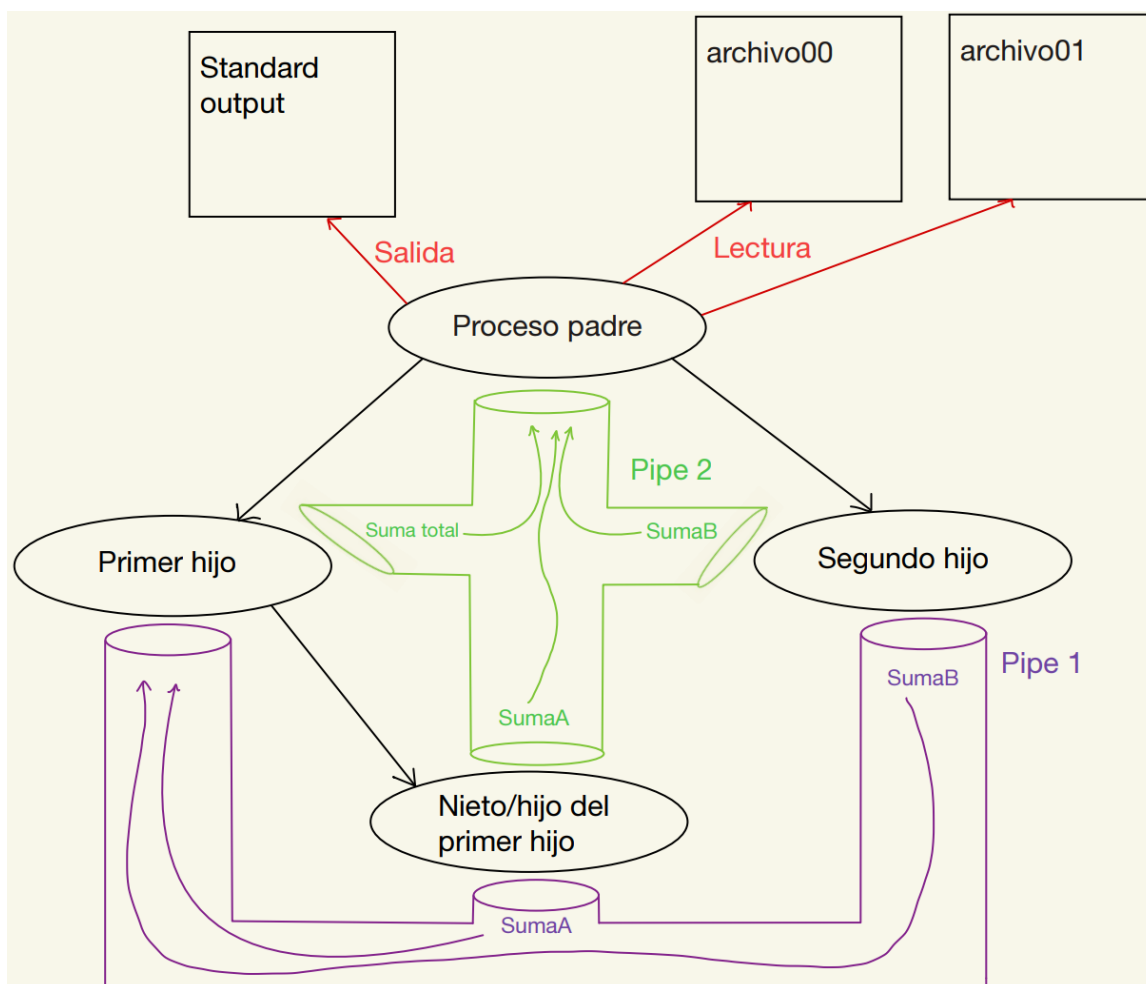


Foto 1. Ejemplo visual del Taller de Procesos

## Marco Teórico

### Función Fork

(*fork(2)* - *Linux manual page*, s. f.) “fork() crea un nuevo proceso duplicando el proceso que lo invoca.

El nuevo proceso se denomina proceso hijo. El proceso que lo invoca se denomina proceso padre.

El proceso hijo y el proceso padre se ejecutan en espacios de memoria separados. Al ejecutar fork(), ambos espacios de memoria tienen el mismo contenido. Las escrituras en memoria, las asignaciones de archivos y las anulaciones de asignaciones realizadas por uno de los procesos no afectan al otro.”

### Función Pipe

(*pipe(2)* - *Linux manual page*, s. f.) “pipe() crea una tubería, un canal de datos unidireccional que puede utilizarse para la comunicación entre procesos. La matriz pipefd se utiliza para devolver dos descriptores de archivo que hacen referencia a los extremos de la tubería. pipefd[0] se refiere al extremo de lectura de la tubería. pipefd[1] se refiere al extremo de escritura de la tubería. Los datos escritos en el extremo de escritura de la tubería se almacenan en el búfer del núcleo hasta que se leen desde el extremo de lectura de la tubería.”

### **Función usleep**

(*usleep(3)* - *Linux manual page*, s. f.) “La función `usleep()` suspende la ejecución del hilo que realiza la llamada durante la cantidad enviada de microsegundos. El tiempo de suspensión puede prolongarse ligeramente debido a la actividad del sistema, al tiempo empleado en procesar la llamada o a la granularidad de los temporizadores del sistema.”

### **Función Wait**

(*wait(2)* - *Linux manual page*, s. f.) “Se utilizan para esperar cambios de estado en un hijo del proceso que realiza la llamada y obtener información sobre dicho hijo cuyo estado ha cambiado. Se considera que un cambio de estado ocurre cuando: el hijo ha finalizado; el hijo fue detenido por una señal; o el hijo fue reanudado por una señal. En el caso de un hijo finalizado, realizar una espera permite al sistema liberar los recursos asociados con él; si no se realiza una espera, el hijo finalizado permanece en un estado "zombi".

### **Intercomunicación de Procesos**

(Silberschatz et al., 2020) “Los procesos que se ejecutan simultáneamente en el sistema operativo pueden ser independientes o cooperantes. Un proceso es independiente si no comparte datos con otros procesos que se ejecutan en el sistema. Un proceso coopera si puede afectar o ser afectado por los demás procesos que se ejecutan en el sistema. Claramente, cualquier proceso que comparte datos con otros procesos es un proceso cooperante.”

## **Proceso**

(Silberschatz et al., 2020) “Un proceso es un programa en ejecución. Un proceso necesita ciertos recursos, como tiempo de CPU, memoria, archivos y dispositivos de E/S, para realizar su tarea. Estos recursos suelen asignarse al proceso mientras se ejecuta.

Un proceso es la unidad de trabajo en la mayoría de los sistemas. Los sistemas constan de un conjunto de procesos: los procesos del sistema operativo ejecutan el código del sistema y los procesos de usuario ejecutan el código del usuario. Todos estos procesos pueden ejecutarse simultáneamente.

Los sistemas operativos modernos admiten procesos con múltiples hilos de control. En sistemas con múltiples núcleos de procesamiento de hardware, estos hilos pueden ejecutarse en paralelo.

Uno de los aspectos más importantes de un sistema operativo es cómo programa los hilos en los núcleos de procesamiento disponibles. Los programadores disponen de varias opciones para diseñar programadores de CPU.”

## Resultados

En este caso de prueba, utilizamos dos ficheros (Ver Foto 2 y Foto 3).

```
1  1 2 3 4 5 6 7 8 9 0 9 8 7 6 5 4 3 2 1 0
```

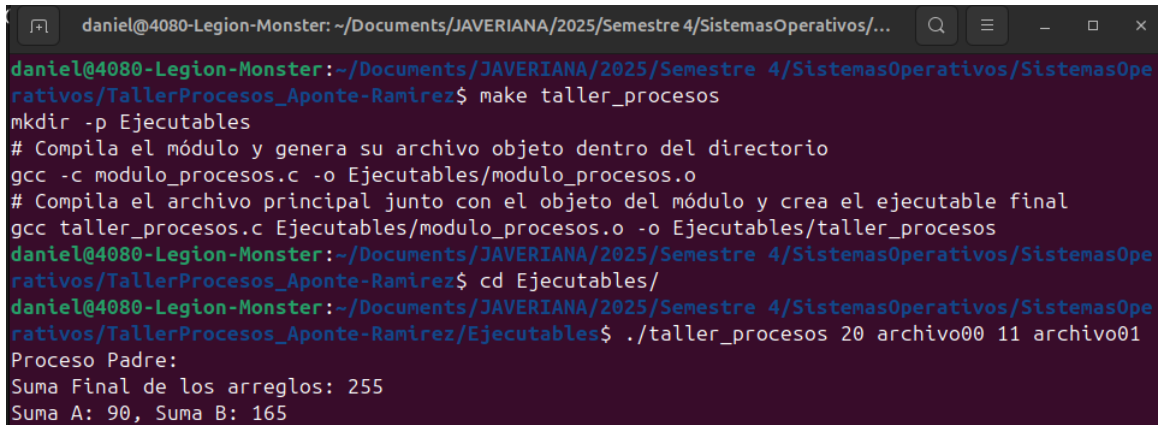
Foto 2. Contenido del fichero “archivo00”

```
1  10 11 12 13 14 15 16 17 18 19 20
2
```

Foto 3. Contenido del fichero “archivo01”

Luego, con ayuda de la compilación automatizada del fichero “Makefile” implementada por nosotros, ejecutamos la regla “taller\_procesos” la cual nos permite crear un directorio donde guardaremos los ejecutables, objetos y los ficheros que vamos a utilizar (archivo00 y archivo01). Seguido de esto, usaremos el comando “cd Ejecutables” para ingresar en el directorio y una vez allí, escribir el comando “./taller\_procesos 20 archivo00 11 archivo01” para visualizar el correcto comportamiento de los procesos (Ver Foto 4).





```
daniel@4080-Legion-Monster: ~/Documents/JAVERIANA/2025/Semestre 4/SistemasOperativos/...
daniel@4080-Legion-Monster:~/Documents/JAVERIANA/2025/Semestre 4/SistemasOperativos/SistemasOpe
rativos/TallerProcesos_Aponte-Ramirez$ make taller_procesos
mkdir -p Ejecutables
# Compila el módulo y genera su archivo objeto dentro del directorio
gcc -c modulo_procesos.c -o Ejecutables/modulo_procesos.o
# Compila el archivo principal junto con el objeto del módulo y crea el ejecutable final
gcc taller_procesos.c Ejecutables/modulo_procesos.o -o Ejecutables/taller_procesos
daniel@4080-Legion-Monster:~/Documents/JAVERIANA/2025/Semestre 4/SistemasOperativos/SistemasOpe
rativos/TallerProcesos_Aponte-Ramirez$ cd Ejecutables/
daniel@4080-Legion-Monster:~/Documents/JAVERIANA/2025/Semestre 4/SistemasOperativos/SistemasOpe
rativos/TallerProcesos_Aponte-Ramirez/Ejecutables$ ./taller_procesos 20 archivo00 11 archivo01
Proceso Padre:
Suma Final de los arreglos: 255
Suma A: 90, Suma B: 165
```

Foto 4. Correcta ejecución del taller

### Análisis de Resultados

Es importante aclarar que el funcionamiento del mecanismo usado para lectura de ambos pipes no es determinista ya que el orden en el que los procesos escriben en los pipes depende del planificador que se use en la máquina donde se ejecuten. En el caso de la lectura del pipe1, el primer hijo de la jerarquía no tiene problema alguno en su operación gracias a la conmutatividad de la suma, sin embargo, el proceso padre si debe mostrar en un orden específico los valores en pantalla por lo cual nosotros optamos por un mecanismo que, aunque útil, sigue sin ser determinista ya que en un sistema altamente cargado o con pocos núcleos lógicos de procesamiento puede no funcionar. Para escribir en el pipe2 (el que lee el padre), utilizamos la función `usleep()` para detener al segundo hijo 50 milisegundos y al primer hijo 100 milisegundos, de esta forma, en la gran

mayoría de casos, el orden de escritura en el pipe2 será

Nieto > SegundoHijo > PrimerHijo.

### Bibliografía

*fork(2)*—*Linux manual page*. (s. f.). Recuperado 28 de octubre de 2025, de

<https://man7.org/linux/man-pages/man2/fork.2.html>

*pipe(2)*—*Linux manual page*. (s. f.). Recuperado 28 de octubre de 2025, de

<https://man7.org/linux/man-pages/man2/pipe.2.html>

Silberschatz, A., Galvin, P., & Gagne, G. (2020). *Operating System Concepts* (Tenth).

Wiley.

*usleep(3)*—*Linux manual page*. (s. f.). Recuperado 28 de octubre de 2025, de

<https://man7.org/linux/man-pages/man3/usleep.3.html>

*wait(2)*—*Linux manual page*. (s. f.). Recuperado 28 de octubre de 2025, de

<https://man7.org/linux/man-pages/man2/wait.2.html>