Facultad de Ingeniería

Asignatura:

Sistemas Operativos

Profesor:

John Corredor Franco

Autores:

Juan Diego Ariza López

Diego Alejandro Mendoza

30 de octubre del 2025

Tabla de contenido

Objetivos

El presente taller tiene como objetivo principal aplicar los conceptos fundamentales de procesos y comunicación entre procesos en sistemas operativos Unix/Linux. Específicamente, se busca:

- Comprender y aplicar la creación de procesos mediante la llamada al sistema **fork().**
- Implementar jerarquías de procesos padre-hijo en múltiples niveles.
- Utilizar **pipes** como mecanismo de comunicación entre procesos (IPC).
- Manejar correctamente la memoria dinámica en programas con múltiples procesos.
- Sincronizar procesos mediante el uso de wait().

Descripción del problema

Se requiere desarrollar un programa en lenguaje C que lea dos archivos con arreglos de números enteros y calcule sus sumas utilizando una jerarquía de cuatro procesos. El programa debe recibir como argumentos de línea de comandos los tamaños de los arreglos y los nombres de los archivos que los contienen.

Formato de ejecución

El programa debe ejecutarse con la siguiente sintaxis:

./taller_procesos N1 archivo00 N2 archivo01

Donde:

- **N1**: cantidad de elementos a leer del primer archivo
- **archivo00**: archivo que contiene el primer arreglo de enteros
- **N2**: cantidad de elementos a leer del segundo archivo
- **archivo01**: archivo que contiene el segundo arreglo de enteros

Implementación

Estructura de Procesos

El programa implementa una jerarquía de cuatro procesos con las siguientes responsabilidades:

Proceso	Función
Grand hijo	Calcula la sumaA del arreglo en archivo00
Segundo hijo	Calcula la sumaB del arreglo en archivo01
Primer hijo	Calcula la suma total de ambos arreglos
Padre	Recibe e imprime los resultados de los tres procesos

La jerarquía de procesos se establece de la siguiente manera:
Padre ⟶ *Primer hijo* ⟶ *Segundo hijo* ⟶ *Grand hijo*

Funciones principales

El programa está organizado en tres funciones principales:

leer_archivo ()

Esta función se encarga de abrir un archivo, leer N números enteros separados por espacios y almacenarlos en un arreglo dinámico. Utiliza malloc() para la asignación de memoria y realiza validación de errores en la apertura del archivo y lectura de datos.

Calcular_suma ()

Función que recibe un arreglo de enteros y su tamaño, y retorna la suma de todos sus elementos. Es utilizada por cada uno de los procesos hijos para calcular sus respectivas sumas.

main ()

Función principal que coordina todo el flujo del programa: valida argumentos, lee los archivos, crea los pipes, genera los procesos mediante fork(), y maneja la comunicación entre ellos.

Comunicación entre Procesos

La comunicación entre procesos se implementa mediante pipes, que son canales unidireccionales de comunicación. Se utilizan tres pipes:

- **pipe_grandhijo[2]**: Comunica el Grand hijo con el Padre para enviar suma
- **pipe_hijo2[2]**: Comunica el Segundo hijo con el Padre para enviar sumaB

- **pipe_hijo1[2]**: Comunica el Primer hijo con el Padre para enviar suma_total

Cada pipe tiene dos extremos: el índice [0] se utiliza para lectura y el índice [1] para escritura. Los procesos hijos cierran el extremo de lectura ya que solo escriben, mientras que el proceso padre cierra los extremos de escritura porque solo lee.

Resultados y Pruebas

Se realizaron múltiples pruebas con diferentes tamaños de arreglos para verificar el correcto funcionamiento del programa. Los archivos de prueba contienen 50 números enteros cada uno, permitiendo probar con diferentes valores de N1 y N2.

Casos de Prueba

Prueba	N1	N2	sumaA	sumaB	Suma Total
1	5	5	150	125	275
2	10	8	325	179	504
3	20	15	675	400	1075
4	50	50	1725	1525	3250

En todas las pruebas realizadas, se verificó que la suma calculada por el Primer hijo (suma_total) coincide con la suma de sumaA + sumaB, confirmando la correcta comunicación entre procesos y la precisión de los cálculos.

Análisis de Resultados

Sincronización de Procesos: El uso de wait() asegura que los procesos padre esperen a sus hijos antes de terminar, evitando procesos zombies y garantizando que todos los cálculos se completen antes de imprimir los resultados finales.

Pipes como IPC: Los pipes demostraron ser un mecanismo eficiente para la comunicación entre procesos. La transmisión de datos de tipo int mediante write() y read() funcionó correctamente en todas las pruebas, sin pérdida de información.

Gestión de Memoria: La asignación dinámica de memoria mediante malloc() permitió manejar arreglos de diferentes tamaños. Es importante destacar que cada proceso tiene su propia copia de los arreglos después del fork(), por lo que la liberación de memoria debe hacerse en cada proceso.

Jerarquía de Procesos: La estructura jerárquica Padre → Primer hijo → Segundo hijo → Grand hijo funciona correctamente. Cada proceso ejecuta su tarea de forma independiente y comunica sus resultados al proceso padre mediante su respectivo pipe.

Conclusiones

1. Se logró implementar exitosamente un sistema de procesos concurrentes que utiliza fork() y pipes para comunicación entre procesos, cumpliendo con todos los objetivos planteados.
2. La comunicación mediante pipes es un mecanismo robusto y eficiente para IPC en sistemas Unix/Linux, especialmente útil cuando se necesita establecer canales de comunicación unidireccionales entre procesos relacionados.
3. El manejo correcto de la sincronización mediante wait() es crucial para evitar condiciones de carrera y asegurar que los procesos terminen en el orden apropiado.
4. La gestión de memoria dinámica en programas con múltiples procesos requiere especial atención, ya que cada proceso debe liberar su propia memoria antes de terminar.
5. Este ejercicio demuestra la importancia de comprender los conceptos fundamentales de procesos en sistemas operativos, conocimiento esencial para el desarrollo de aplicaciones concurrentes y sistemas distribuidos.