

Sistemas Operativos Taller 01

Viviana Gómez, Gabriel Jaramillo, Roberth Méndez, Luz Adriana Salazar, Guden Silva

April 2025

1 Resumen

El taller consiste en desarrollar un programa en lenguaje C que reciba dos ficheros (cada uno con un arreglo de enteros) y dos valores (cantidad de elementos de cada arreglo) como argumentos de la línea de comandos. El programa debe asignar memoria dinámicamente para almacenar los arreglos, leer los datos desde los ficheros y, posteriormente, crear cuatro procesos mediante `fork()` para realizar cálculos específicos:

- El proceso "gran hijo" calcula la suma de los elementos del primer arreglo.
- El proceso "segundo hijo" calcula la suma de los elementos del segundo arreglo.
- El proceso "primer hijo" calcula la suma de ambos arreglos.
- El proceso padre recibe los resultados a través de `pipe()` y los muestra en pantalla.

1.1 Planteamiento del Problema

El problema que el código en C debe resolver es el siguiente:

- **Entrada:** Dos ficheros que contienen listas de números enteros y dos valores enteros que indican la cantidad de elementos de cada fichero.
- **Objetivo:** Implementar un programa en C que, mediante la creación de procesos, realice operaciones de suma sobre los datos leídos de los ficheros.

El programa a desarrollar cuenta con los siguientes requisitos para su correcta implementación y funcionamiento:

- Lectura y manejo de ficheros para extraer datos en forma de arreglos.
- Uso de memoria dinámica para almacenar los arreglos.
- Creación y manejo de múltiples procesos utilizando `fork()`.

- Comunicación entre procesos (IPC) mediante `pipe()` para transmitir los resultados de las operaciones.
- Integración de todos estos componentes en un solo programa de forma correcta y eficiente.

1.2 Propuesta de Solución

Para resolver el problema, el código implementa lo siguiente:

- **Lectura y gestión de archivos:**
 - Se debe implementar una función que lea los dos ficheros de entrada.
 - Se debe procesar cada fichero para extraer los números enteros y almacenarlos en un vector asignado dinámicamente de tamaño N1 y N2, respectivamente.
 - Se debe Verificar que el número de elementos leídos coincida con los valores ingresados como argumentos.
- **Creación de procesos usando `fork()`:**
 - Gran hijo: Encargado de calcular la suma de todos los elementos del primer arreglo.
 - Segundo hijo: Calcula la suma de los elementos del segundo arreglo.
 - Primer hijo: Realiza la suma total combinando ambos arreglos.
 - Proceso padre: Coordina la recolección de resultados de los tres procesos hijos mediante comunicación por `pipe()` y muestra el resultado final en pantalla.
- **Liberar memoria:**
 - Liberar la memoria dinámica asignada una vez se hayan completado todas las operaciones.
 - Cerrar los descriptores de archivo abiertos y manejar correctamente el cierre de los procesos.

1.3 Método de Prueba

Para probar la funcionalidad del programa, se sigue el siguiente método de prueba:

1. Prueba de Lectura de Ficheros:
 - Probar con ficheros de diferentes tamaños y formatos (asegurarse de que los datos estén separados por espacios).
 - Verificar que la cantidad de elementos leídos corresponda a los valores N1 y N2 pasados como argumentos.

2. Prueba de Creación y Comunicación de Procesos:

- Ejecutar el programa y verificar que se creen los cuatro procesos.
- Comprobar que cada proceso realice la suma correspondiente y que los resultados sean correctamente enviados y recibidos mediante los pipes.
- Incluir pruebas de manejo de errores (por ejemplo, qué ocurre si un fichero no existe o si falla la creación de un proceso).

3. Prueba de Liberación de Recursos:

- Verificar que no existan fugas de memoria.
- Confirmar el correcto cierre de descriptores y procesos al finalizar la ejecución.

1.4 Resultados Esperados

Después de la ejecución del programa se espera lo siguiente:

- **Salida en consola:** El proceso padre debe mostrar:
 - La suma de los elementos del primer arreglo (calculada por el gran hijo).
 - La suma de los elementos del segundo arreglo (calculada por el segundo hijo).
 - La suma total combinada de ambos arreglos (calculada por el primer hijo).
- **Comunicación entre procesos:** Los datos deben ser correctamente enviados a través de los pipes y no deben existir pérdidas o errores en la transmisión de información.

2 Diseño

2.1 Diseño general

- El programa esta escrito en el lenguaje de programación C.
- El programa utiliza lo siguiente:
 - pipe(): para comunicación entre procesos.
 - fork(): para crear procesos hijos.
 - wait(): para sincronización.
- Las bibliotecas que se incluyen permiten lo siguiente:
 - Entrada/salida (stdio.h).

- Memoria dinámica (stdlib.h).
- Procesos y comunicación (unistd.h, sys/types.h, sys/wait.h).
- Se define una estructura llamada `vectorDinamico` que:
 - Actúa como un vector dinámico.
 - Almacena punteros genéricos (`void**`), aunque se usan específicamente para enteros.

2.2 Lectura y preparación de datos

- La función `main` verifica que haya 5 argumentos:
 - Ejecutable.
 - `N1`: capacidad para el primer vector.
 - `archivo00`: nombre del archivo con enteros.
 - `N2`: capacidad para el segundo vector.
 - `archivo01`: segundo archivo.
- El programa realiza lo siguiente:
 - Convierte `N1` y `N2` en enteros.
 - Inicializa los vectores `vector1` y `vector2` usando `vectorInicio`.
 - Llama a `leerFichero` para llenar cada vector desde su respectivo archivo.
- Se define una estructura llamada `vectorDinamico` que:
 - Actúa como un vector dinámico.
 - Almacena punteros genéricos (`void**`), aunque se usan específicamente para enteros.

2.3 Comunicación entre procesos

- Se crean dos **pipes**:
 - **Pipe principal** (`pipePrincipal[2]`): utilizado para la comunicación entre el proceso *padre* y el proceso *hijo sumaTotal*.
 - **Pipe interno** (`pipeInterno[2]`): utilizado exclusivamente por el proceso *hijo sumaTotal* para recibir los resultados de los procesos *sumaA* y *sumaB*.
- Se crean tres procesos hijos utilizando `fork()`:
 1. **Proceso hijo sumaTotal:**
 - Crea un *pipe interno* para comunicarse con sus propios hijos (*sumaA* y *sumaB*).

- Crea dos procesos hijos:
 - * **sumaA**: calcula la suma de los elementos del **vector1** y la envía por el **pipeInterno**.
 - * **sumaB**: calcula la suma del **vector2** y la envía también por el **pipeInterno**.
 - Espera a que finalicen ambos procesos (**sumaA** y **sumaB**).
 - Lee las dos sumas parciales desde el **pipeInterno**.
 - Calcula la **sumaTotal = sumaA + sumaB**.
 - Envía los tres resultados al proceso *padre* mediante el **pipePrincipal**.
2. **Proceso sumaA:**
- Calcula la suma de los valores del **vector1**.
 - Envía el resultado a través del **pipeInterno** al proceso *sumaTotal*.
3. **Proceso sumaB:**
- Calcula la suma de los valores del **vector2**.
 - Envía el resultado a través del **pipeInterno** al proceso *sumaTotal*.
- Cada proceso cierra los extremos del pipe que no utiliza:
 - Por ejemplo, un proceso que solo escribe en un pipe cierra el extremo de lectura y viceversa.
 - Esto evita bloqueos innecesarios y garantiza una comunicación unidireccional eficiente.
 - La comunicación se realiza con las funciones **write()** y **read()** para enviar y recibir los datos entre los procesos.

2.4 Sincronización y lectura de resultados

- El proceso **padre** (proceso principal) realiza la sincronización general:
 - Espera la finalización del proceso *sumaTotal* mediante **wait()**.
 - Cierra el extremo de escritura del **pipePrincipal**.
 - Lee los resultados enviados desde el proceso *sumaTotal*, en este orden:
 1. **sumaA** (desde **vector1**).
 2. **sumaB** (desde **vector2**).
 3. **sumaTotal** (la suma total de ambos vectores).
 - Finalmente, imprime los tres valores en la consola.

2.5 Limpieza de memoria

:

- Se libera la memoria reservada por los vectores usando `freeVector`.
- Esto previene fugas de memoria, una práctica esencial en C.
- El programa finaliza con `return 0`; para indicar ejecución exitosa.

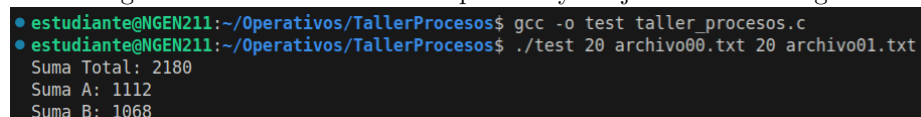
2.6 Diseño por módulos

: El código tiene las siguientes funciones auxiliares de forma separadas:

- `vectorInicio`: reserva e inicializa un vector dinámico.
- `leerFichero`: carga enteros desde un archivo al vector.
- `sumarVector`: calcula la suma de los valores del vector.
- `freeVector`: libera memoria.

3 Resultados

Figure 1: Resultados de la compilación y la ejecución del código



```
● estudiante@NGEN211:~/Operativos/TallerProcesos$ gcc -o test taller_procesos.c
● estudiante@NGEN211:~/Operativos/TallerProcesos$ ./test 20 archivo00.txt 20 archivo01.txt
Suma Total: 2180
Suma A: 1112
Suma B: 1068
```

4 Análisis de resultados

El taller demostró que `fork()` no solo sirve para duplicar procesos, sino también para dividir tareas, controlar la ejecución concurrente, facilitar la comunicación entre procesos y organizar la estructura del programa. Esto es clave para desarrollar software eficiente que interactúe correctamente con el sistema operativo. Se hacen las siguientes observaciones:

- **Creación de procesos con `fork()`**: El taller facilitó la comprensión práctica del uso de la función `fork()` para crear procesos en sistemas operativos basados en Unix o Linux. Esta función permite generar un proceso hijo idéntico al padre, y se evidenció que luego de ejecutarla, el flujo del programa se divide: uno para el padre y otro para el hijo, diferenciados por el valor retornado por `fork()`.

- **Asignación de tareas a procesos hijos:** Se aprendió a asignar distintas funciones a cada proceso hijo mediante la evaluación del valor retornado por `fork()`. Así, se puede programar a cada hijo para realizar tareas específicas, como sumar datos de archivos o hacer cálculos globales. Esto permite distribuir el trabajo y simular concurrencia en la ejecución.
- **Control y sincronización de procesos:** También se entendió la necesidad de controlar los procesos con `wait()`, que permite al proceso padre esperar la finalización de sus hijos. Esta sincronización es vital para evitar inconsistencias y garantizar que los resultados se obtengan correctamente.
- **Comunicación entre procesos con `pipe()`:** Dado que cada proceso creado por `fork()` tiene su propio espacio de memoria, se exploró el uso de `pipe()` como mecanismo de comunicación. Con él, los procesos hijos pueden enviar datos al padre, quien luego se encarga de presentarlos, permitiendo una interacción controlada entre procesos separados.
- **Jerarquías de procesos con múltiples `fork()`:** Se observó que un proceso hijo también puede crear su propio hijo usando `fork()`, lo que permite construir estructuras jerárquicas o árboles de procesos. Esto es útil en aplicaciones más complejas que requieren control en múltiples niveles.

5 Conclusiones

El análisis del código en C permite extraer varias conclusiones interesantes sobre su diseño y funcionalidad:

- **Creación de procesos y distribución de tareas:**
La función `fork()` permite la creación de procesos hijos y cada uno de estos procesos ejecuta tareas diferentes de manera independiente según su valor de retorno. La distribución de trabajo entre el proceso padre y los procesos hijos es fundamental para mejorar el rendimiento del programa ya que permite que las diferentes tareas se ejecuten al mismo tiempo a través del paralelismo o la concurrencia con el objetivo de aprovechar los recursos del sistema operativo.
- **Comunicación entre procesos mediante pipes:**
Debido a que los procesos no comparten memoria, pues que cada uno tiene su propia memoria independiente, es necesario utilizar mecanismos como los pipes para compartir información entre ellos. Esta función crea un canal de comunicación entre procesos relacionados, con un extremo de escritura y otro de lectura, lo que permite la transmisión de datos de manera eficiente.
- **Buenas prácticas en el uso de pipes:**
Una buena practica de programación al utilizar pipe es cerrar los extremos

que no se van a usar para evitar bloqueos en la comunicación entre procesos. Por ejemplo, para aquellos procesos que únicamente van a escribir, se debe cerrar el extremo de lectura, y para aquellos procesos que únicamente van a leer, se debe cerrar el extremo de escritura.

- **Sincronización de procesos con wait:**

Es importante hacer uso de la función wait para que el proceso padre espere a que cada proceso hijo termine antes de continuar su ejecución. Esto permite garantizar que los resultados de las tareas realizadas por los procesos hijos estén listos antes de ser utilizados y evita la creación de procesos zombie, los cuales ocurren cuando un proceso hijo finaliza pero su estado no es recogido por el padre.