INFORME TALLER FORK()

DANNA GABRIELA ROJAS BERNAL
JUAN DAVID DAZA CARO

JOHN CORREDOR FRANCO
PROFESOR DE LA ASIGNATURA SISTEMAS OPERATIVOS

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERÍA
INGENIERÍA DE SISTEMAS – SISTEMAS DE INFORMACIÓN
BOGOTÁ D.C.
2025

1. Objetivo

Crear un programa en C que permita calcular la suma parcial de los números enteros en un archivo de texto y la suma total de estos, haciendo uso de subprocesos por medio de fork() y efectuando comunicación entre procesos por medio de pipes.

2. Metodologia

Primero se identifican los parámetros a recibir por el programa.

N1: número de elementos (números enteros) contenidos en un fichero
archivo00: archivo de texto que contiene números separados por espacios
N2: número de elementos (números enteros) contenidos en un segundo fichero
Archivo01: archivo de texto que contiene números separados por espacios

Dados los argumentos que recibe el programa nos podemos dar cuenta de que podemos leer los contenidos de este utilizando la función estándar fscanf() de C, debido a que están separados por espacios. Y simplemente iterar la lectura N_i veces hasta leer cada archivo por completo.

La información leída se guarda en vectores de punteros con memoria dinámica, utilizando la función malloc()

El enunciado pide crear distintos procesos hijo para calcular la suma de los enteros contenidos en los archivos, para esto se va a utilizar la función fork() que es una system call del estándar POSIX la cual se puede acceder por medio de la librería <unistd.h>

Para comunicar la información procesada por cada uno de estos subprocesos se utilizara un pipe, el cual es un canal de comunicación entre procesos unidireccional que se accede por medio de la librería <unistd.h> solo funciona con procesos padre e hijo.

3. Solucion

El programa lee dos archivos con N1 y N2 enteros, crea tres procesos hijos que calculan las sumas (archivo1, archivo2 y suma total), y usa tres pipes para enviar cada resultado al proceso padre. El padre imprime las tres sumas y luego libera la memoria dinámica. Se usan fork, pipe, read, write, y waitpid para gestionar procesos y comunicación.

Primero se declararon las siguientes librerías stdio.h, stdlib.h, unistd.h, string.h, sys/wait.h las cuales proporcionan funciones de entrada y salida, la creación de

procesos, pipes y espera de procesos.

Luego se creó la función “leer archivo”que tiene como propósito abrir un archivo de texto, verificar que se accede correctamente y luego lee la cantidad específica de números enteros. Para almacenar los valores, la función reserva memoria dinámica utilizando malloc, permitiendo guardar los datos. Se crea un bucle en el que a medida de que se leen los números, se va guardando en el arreglo dinámico. Se cierra el archivo y se devuelve la dirección del arreglo, para que el programa principal pueda trabajar con los datos leídos.

```c
//funicion para leer archivo y verificar si se abrio
int* leer_archivo(char *nombre, int n) {
    FILE *arch = fopen(nombre, "r"); // abrir archivo en modo lectura
    if  (arch == NULL) {
        printf("Error al abrir el archivo.\n");
        exit(1);
    }

    int *arr = (int*) malloc(n * sizeof(int));//crea memoria dinamica para guardar  n numeros enteros
    for (int i = 0; i < n; i++) {
        fscanf(arch, "%d", &arr[i]);// lee numeros del archivo (&arr[i] dirreccion de memmoria donde se va  guardar el numero)
    }
    fclose(arch);
    return arr;
}
```

Imagen #1. Función leer archivo

Por otra parte, la función “suma_arreglo” recibe un arreglo de números enteros y calcula la suma de todos la cantidad de números que se pidan, por lo que recorre el arregla con un ciclo for, acumulando cada valor en la variable suma para luego devolver este parámetro a la función principal.

```c
//funcion con algoritmo basico de suma de arreglos
int suma_arreglo(int *arr, int n) {
    int suma = 0;
    for (int i = 0; i < n; i++){
          suma += arr[i];
        }
    return suma;
}
```

Imagen #2. Función suma_arreglo

En el Main se verifica que se haya ingresado correctamente los argumentos necesarios, la cantidad de números y el nombre de cada archivo

```c
 if (argc != 5) { //verifica el paso de argumentos
     printf("Uso: %s N1 archivo00 N2 archivo01\n", argv[0]);
  return 0;
     }
```

Imagen #3. Verifica el paso de argumentos

Luego convierte esos argumentos en valores entres ya que llega como una cadena de caracteres y llama a la función “leer_archivo” para cargar los datos desde ambos archivos en memoria dinámica.

```
    // Conversión de los argumentos tipo texto a enteros
int N1 = atoi(argv[1]);
int N2 = atoi(argv[3]);
    // Leer los archivos con la cantidad de datos indicada
int *arr1 = leer_archivo(argv[2], N1);
int *arr2 = leer_archivo(argv[4], N2);
```

Imagen #4. Conversión de argumentos y lectura de archivos

El programa crea tres pipes que permite la comunicación entre procesos y llama al fork para generar tres procesos hijos para cada tarea específica. El primer hijo calcula la suma del primer archivo, el segundo hijo hace lo mismo con el segundo archivo y el tercero obtiene la suma total combinando ambas.

```
    // Declaración de tres pipes para la comunicación entre procesos
     int pipefd1[2],pipefd2[2],pipefdTotal[2];

    //verificar que todos los pipes se creen correctamente
    if(pipe(pipefd1)==-1 || pipe(pipefd2) == -1 || pipe(pipefdTotal) == -1){
       perror("PIPE");
            exit(EXIT_FAILURE);
    }

    // Declaración de variables para los tres procesos
    pid_t pid1, pid2, pidTotal;
```

Imagen #5. Declaración de pipes, variables para los tres procesos y verificación de pipes

Mientras los hijos realizan los cálculos correspondientes, el padre se encarga de recibir los resultados por medio de los *pipes* utilizando las funciones read y write. Además, el padre usa waitpid para esperar a que cada hijo termine antes de continuar, garantizando la sincronización correcta entre los procesos. Una vez que ha recibido todas las sumas, el padre muestra en pantalla los resultados individuales y la suma total.

```c
//Porceso 1
        pid1 = fork();//crea primer proceso hijo
        if(pid1 == 0){
        close(pipefd1[0]); // cerrar lectura
        int suma01 = suma_arreglo(arr1, N1);
        write(pipefd1[1], &suma01, sizeof(int)); // enviar suma al padre
        close(pipefd1[1]); // cerrar escritura
        free(arr1);
        free(arr2);
        exit(0);//finaliza hijo
    }else{
        close(pipefd1[1]); // cerrar escritura (el padre solo lee)
        waitpid(pid1, NULL, 0); // esperar a que el hijo termine
        int sumaA;
        read(pipefd1[0], &sumaA, sizeof(int)); // leer lo que escribió el hijo
        close(pipefd1[0]); // cerrar lectura
        printf("Suma del archivo %s: %d\n", argv[2], sumaA);
```

Imagen #6. Calculo de proceso 1

Finalmente, libera la memoria dinámica utilizada y finaliza el programa correctamente.

```c
//liberar memoria utilizada en el proceso padre
  free(arr1);
  free(arr2);
  return 0;

}
```

Imagen #7. Libera memoria

Así que al momento de ejecutar el programa, su salida es la siguiente:

```
estudiante@NGEN466:~/Taller02$ ./taller_procesos 4 archivo00.txt 7 archivo01.txt
Suma del archivo archivo00.txt: 138
Suma del archivo archivo01.txt: 120
Suma total: 258
```

Imagen #8. Compilación de programa

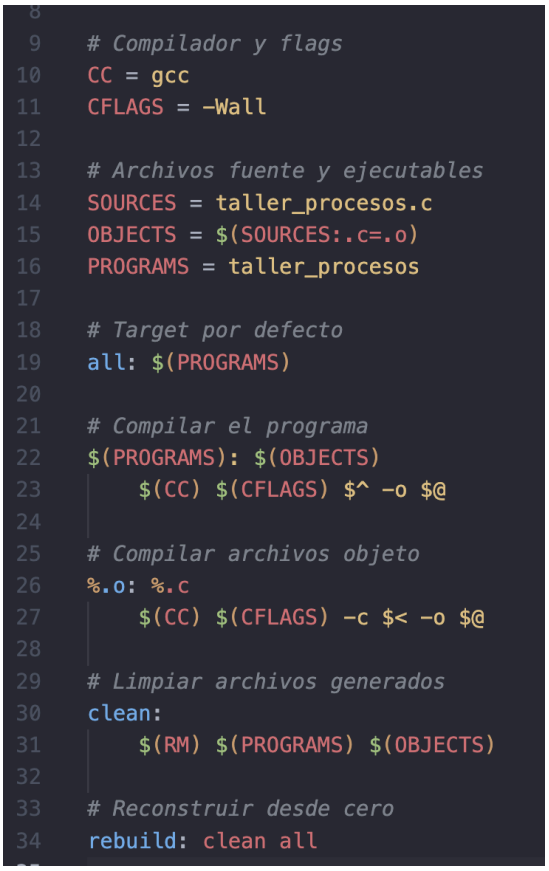
Para el anterior ejemplo se utilizó los siguientes archivos y para la ejecución del programa el siguiente Makefile:

```
estudiante@NGEN466:~/Taller02$ cat archivo00.txt
13 14 109 2 3 8 0 9 12 14 1 15 16 22
estudiante@NGEN466:~/Taller02$ cat archivo01.txt
1 2 45 15 24 31 2
```

Imagen #9. Contenido de los archivos

El makefile toma como fuente el único archivo de código fuente (taller_procesos.c) del taller. Primero compila el .c en un objeto y después crea el ejecutable, compilar de esta manera es una buena práctica, especialmente cuando se tienen múltiples archivos de código fuente, de esta manera se asegura de que el linker de compilador funcione de manera adecuada y de verificar errores en compilación de un .c específico.

Además se establece el target rebuild para hacer clean y make con un solo comando.



```
# Compilador y flags
CC = gcc
CFLAGS = -Wall

# Archivos fuente y ejecutables
SOURCES = taller_procesos.c
OBJECTS = $(SOURCES:.c=.o)
PROGRAMS = taller_procesos

# Target por defecto
all: $(PROGRAMS)

# Compilar el programa
$(PROGRAMS): $(OBJECTS)
    $(CC) $(CFLAGS) $^ -o $@

# Compilar archivos objeto
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Limpiar archivos generados
clean:
    $(RM) $(PROGRAMS) $(OBJECTS)

# Reconstruir desde cero
rebuild: clean all
```

Imagen #10. Makefile

4. Conclusiones

Las conclusiones con el desarrollo del taller son las siguientes:

- El programa logra crear procesos y comunicarse entre ellos
- Se hace un uso adecuado de fork() que permitió dividir las tareas de cálculo independientes entre distintos procesos.
- La comunicación con el pipe anónimo permite la comunicación entre procesos sin la necesidad de compartir memoria.
- La colocación y liberación de memoria con malloc() y free() permitió manejar memoria dinámica y liberarla en cada proceso. Dado a que cuando se utiliza

fork() el proceso hijo copia toda la información del proceso padre, incluyendo la información del Stack y Heap, así como los registros del proceso en CPU.