Fecha: 30/03/2025

Taller 2 Procesos

Autores: Jeronimo Chaparro Tenorio
Materia: Sistemas Operativos
Pontificia Universidad Javeriana

1. Resumen de lo solicitado:

En este taller se nos solicita realizar una rutina para lectura de los arreglos de enteros en ficheros, dentro de un programa principal en C.

2. Objetivo:

Se busca utilizar lo visto en clase de las funciones fork() y pipe() para utilizar procesos recurrentes y cumplir con lo solicitado.

3. Desarrollo:

Primero se buscó realizar las dos funciones que consideramos útiles para completar el taller, que son: leer_archivo y calcular_suma

leer_archivo

```c
//Función para leer un archivo y almacenar los números en un arreglo
void leer_archivo(const char *nombre, int *arr, int n) {
    FILE *file = fopen(nombre, "r");
    if (!file) {
        perror("Error abriendo archivo");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < n; i++) {
        fscanf(file, "%d", &arr[i]);
    }
    fclose(file);
}
```

leer_archivo se encarga de leer un archivo y guardar su información en un vector con memoria dinámica (por medio de punteros) de enteros.

calcular_suma

```c
//Función para calcular la suma de los elementos de un arreglo
int calcular_suma(int *arr, int n) {
    int suma = 0;
    for (int i = 0; i < n; i++) {
        suma += arr[i];
    }
    return suma;
}
```

calcular_suma calcula la suma de todos los valores dentro de un vector con memoria dinámica (por medio de punteros) de enteros y la almacena en una variable llamada suma.

La siguiente parte del programa constó de la creación de la función main() convirtiendo primero los argumentos del programa en variables a utilizar, luego declarando los vectores dinámicos y por último ingresando, a estos últimos, valores a través de la función leer_archivo.

```c
int main(int argc, char *argv[]) {
    if (argc != 5) {
        fprintf(stderr, "Uso: %s N1 archivo00 N2 archivo01\n", argv[0]);
        return EXIT_FAILURE;
    }
    //Se extraen de los argumentos de la línea de comandos los valores de N1, archivo00, N2 y archivo01
    int N1 = atoi(argv[1]);
    char *archivo00 = argv[2];
    int N2 = atoi(argv[3]);
    char *archivo01 = argv[4];

    //Vectores con memoria dinámica
    int *arr1 = (int *)malloc(N1 * sizeof(int));
    int *arr2 = (int *)malloc(N2 * sizeof(int));
    if (!arr1 || !arr2) {
        perror("Error al asignar memoria");
        return EXIT_FAILURE;
    }

    leer_archivo(archivo00, arr1, N1);
    leer_archivo(archivo01, arr2, N2);
```

Se creó una pipe fd para la comunicación entre procesos, dos variables enteras para identificar la suma (sumaA, sumaB o sumaTotal) y su valor y tres identificadores de procesos.

```c
//Se crea el pipe fd[2] para la comunicación entre procesos
//Siendo fd[0] el extremo de lectura y fd[1] el extremo de escritura
int fd[2];
pipe(fd);

int id, value;

//Se crean tres identificadores de procesos
pid_t pid1, pid2, pid3;
```

A continuación se crea un if else donde se crea el primer proceso hijo en ese mismo if se crea otro if else, donde se crea el segundo proceso hijo que realiza la sumaA, en el else se hace la sumaB por el primer proceso hijo. Ambas sumas se guardan en fd, antes escribir cada operación en el pipe fd se coloca el identificador de dicha operación.

```
//Se crea el primer proceso hijo pid1
if ((pid1 = fork()) == 0) {
    //Se crea el proceso nieto pid2
    if ((pid2 = fork()) == 0) {
        //pid2 hace la suma del archivo00
        int sumaA = calcular_suma(arr1, N1);
        id = 1; // Identificador para sumaA
        write(fd[1], &id, sizeof(int));
        //pid2 escribe la suma en el pipe fd[1]
        write(fd[1], &sumaA, sizeof(int));
        exit(0);
    } else {
        //pid1 espera a que pid2 termine
        wait(NULL);
        //pid1 hace la suma del archivo01
        int sumaB = calcular_suma(arr2, N2);
        id = 2; // Identificador para sumaB
        write(fd[1], &id, sizeof(int));
        //pid1 escribe la suma en el pipe fd[1]
        write(fd[1], &sumaB, sizeof(int));
        exit(0);
    }
```

En el else del primer if se crea otro if else, donde, en el if, se crea el proceso tres que hace la suma total y el proceso padre se encarga de mostrar todas las sumas realizadas por medio de fd. Antes escribir cada operación en el pipe fd se coloca el identificador de dicha operación.

```
} else {
    //Se crea el segundo proceso hijo pid3
    if ((pid3 = fork()) == 0) {
        wait(NULL);
        //pid3 espera a que pid1 termine
        int sumaTotal = calcular_suma(arr1, N1) + calcular_suma(arr2, N2);
        id = 3; // Identificador para sumaTotal
        write(fd[1], &id, sizeof(int));
        //pid3 escribe la suma en el pipe fd[1]
        write(fd[1], &sumaTotal, sizeof(int));
        exit(0);
    } else {
        //El proceso padre espera a que pid1 y pid3 terminen
        wait(NULL);
        wait(NULL);
        //El proceso padre lee las sumas del pipe fd[0]
        int sumaA = 0, sumaB = 0, sumaTotal = 0;

        for (int i = 0; i < 3; i++) {
            read(fd[0], &id, sizeof(int));
            read(fd[0], &value, sizeof(int));

            if (id == 1) sumaA = value;
            else if (id == 2) sumaB = value;
            else if (id == 3) sumaTotal = value;
        }
        //Se muestran los resultados de las sumas
        printf("Suma archivo00: %d\n", sumaA);
        printf("Suma archivo01: %d\n", sumaB);
        printf("Suma total: %d\n", sumaTotal);
    }
}
```

Por último se cierra el pipe fd y se libera la memoria de los vectores dinámicos.

```
//Se cierran los extremos del pipe fd[0] y fd[1]
close(fd[0]);
close(fd[1]);
//Se liberan los punteros de memoria
free(arr1);
free(arr2);
return EXIT_SUCCESS;
```

4. Conclusión:

Este programa en C implementa un sistema de procesamiento concurrente utilizando la creación de procesos con fork() y la comunicación entre ellos mediante pipe(). Su propósito es leer dos archivos de texto con arreglos de enteros, calcular sus sumas parciales y la

suma total utilizando múltiples procesos, y finalmente consolidar los resultados en el proceso padre.

El programa demuestra el uso eficiente de procesos concurrentes para dividir tareas y luego consolidar los resultados. Aunque es un esquema simple de procesamiento concurrente, es una base sólida para aplicaciones más avanzadas, como el procesamiento distribuido o la programación paralela.