Programación Concurrente, Junio 2023

Práctica Final

Martínez Buenrostro Jorge Rafael.
Universidad Autónoma Metropolitana
Unidad Iztapalapa, México
*molap*96*@gmail.com*

El código para esta práctica se encuentra en el siguiente enlace Online GDB

En esta práctica, se nos dio la opción de elegir uno de varios problemas para analizar y diseñar un código que resuelva la tarea de forma concurrente. En mi caso, seleccioné el problema 3, el cual consiste en desarrollar un programa concurrente que tome N archivos de números enteros previamente ordenados de manera ascendente y genere un único archivo con todos los números ordenados de forma ascendente.

La idea detrás de este programa es determinar inicialmente el número de archivos presentes en un directorio. A partir de esta información, se presentan dos opciones: en primer lugar, si la cantidad de archivos es par, se asigna a cada proceso hijo un par de archivos para que los ordene de forma ascendente en un nuevo archivo, el cual será guardado en el mismo directorio. Este proceso se repite hasta que solo quede un archivo que contenga todos los datos ordenados de forma ascendente. Por otro lado, si la cantidad de archivos es impar, el proceso padre se encargará de combinar el último archivo ordenado generado por los hijos con el archivo impar restante, asegurando así la correcta ordenación de los datos.

1. En la función `main()`, se realiza lo siguiente:

- Se declara una serie de variables, incluyendo el directorio a leer (`directory`), la cantidad de procesos de lectura (`readingProcesses`), un contador (`i`) y un identificador de proceso (`pid`).
- Se muestra una lista de directorios disponibles para que el usuario elija uno.
- Se solicita al usuario que ingrese el directorio que desea leer.
- Se intenta abrir el directorio especificado. Si no se puede abrir, se muestra un mensaje de error y se termina el programa.
- Se declara una estructura `dirent` para almacenar la información de cada archivo en el directorio.
- Se declara una variable para contar la cantidad de archivos encontrados (`fileCount`) y un arreglo para almacenar los pares de nombres de archivos (`filePairs`).
- Se itera sobre cada archivo en el directorio y se construye la ruta completa del archivo. Los nombres de los archivos se almacenan en el arreglo `filePairs`.
- Se cierra el directorio.

2. Luego, se realiza un ciclo `for` para procesar los pares de archivos en procesos hijos:

➢ Se obtienen los nombres de los archivos del par actual (`filename1` y `filename2`).
➢ Se crea un nuevo proceso hijo usando la función `fork()`.
➢ En el proceso hijo, se llama a la función `readFilesInDirectory()` para procesar los archivos.
➢ El proceso hijo sale (`exit(0)`).

3. Después de que todos los procesos hijos han terminado, se ejecuta un ciclo `while` para esperar a que todos los procesos hijos finalicen.

4. Luego, se verifica si quedan archivos en el directorio:

➢ Se abre el directorio nuevamente y se obtiene la lista de archivos restantes.
➢ Se cuenta la cantidad de archivos restantes (`remainingCount`).
➢ Si hay archivos restantes, se realiza lo siguiente:
➢ Se crea un buffer para almacenar los números de los archivos restantes (`buffer`) y una variable para contar el número total de elementos (`totalCount`).
➢ Se itera sobre cada archivo restante y se procesa de manera similar a los archivos anteriores. Los números se leen y se agregan al buffer.
➢ Luego, se ordenan los números en el buffer de forma ascendente.
➢ Se concatena los nombres de los archivos restantes para crear el nombre del archivo nuevo (`merged_filename`).
➢ Se crea un nuevo archivo con el nombre generado y se escriben los números ordenados en él.
➢ Se liberan los nombres de archivo restantes y se muestra un mensaje indicando el archivo nuevo creado.

5. Finalmente, se liberan los nombres de archivo originales y se retorna 0 para indicar que el programa finalizó correctamente.

Código del programa

```c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
```

```c
void readFilesInDirectory(const char* directory, const char* filename1, const char* filename2);

int main() {
    char directory[256];
    int readingProcesses, i;
    pid_t pid;

    printf("\nDirectorios disponibles:\n");
    system("ls -d */");
    // Pide al usuario el directorio
    printf("\nIngrese el directorio a leer: ");
    scanf("%s", directory);

    DIR* dir = opendir(directory);
    if(dir == NULL){
        printf("No se pudo abrir el directorio '%s'\n", directory);
        return 1;
    }

    struct dirent* entry;
    int fileCount = 0;
    char* filePairs[256];

    // Lee los nombres de los archivos en el directorio
    while ((entry = readdir(dir)) != NULL) {
        // Ignora los directorios "." y ".."
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        // Construye la ruta completa del archivo
        char file_path[256];
        snprintf(file_path, sizeof(file_path), "%s/%s", directory, entry->d_name);

        // Almacena los nombres de los archivos en el array filePairs
        filePairs[fileCount] = strdup(entry->d_name);
```

```
        fileCount++;
    }

    // Cierra el directorio
    closedir(dir);

    // Procesa los pares de archivos en los procesos hijos
    for (i = 0; i < fileCount; i += 2) {
        char* filename1 = filePairs[i];
        char* filename2 = filePairs[i + 1];

        // Crea un nuevo proceso hijo
        pid_t pid = fork();

        if (pid == -1) {
            printf("Error al crear el proceso hijo\n");
            return 1;
        } else if (pid == 0) {
            // Proceso hijo
            readFilesInDirectory(directory, filename1, filename2);
            exit(0);
        }
    }

    // Espera a que todos los procesos hijos terminen
    while (wait(NULL) > 0);

    // Procesa los archivos restantes en el proceso padre
    char* remainingFiles[256];
    int remainingCount = 0;

    DIR* remainingDir = opendir(directory);
    if (remainingDir == NULL) {
        printf("No se pudo abrir el directorio '%s'\n", directory);
        return 1;
    }
```

```
    struct dirent* remainingEntry;

    // Lee los nombres de los archivos restantes en el directorio
    while ((remainingEntry = readdir(remainingDir)) != NULL) {
        // Ignora los directorios "." y ".."
        if (strcmp(remainingEntry->d_name, ".") == 0 ||
strcmp(remainingEntry->d_name, "..") == 0) {
            continue;
        }

        // Construye la ruta completa del archivo restante
        char remaining_file_path[256];
        snprintf(remaining_file_path, sizeof(remaining_file_path), "%s/%s",
directory, remainingEntry->d_name);

        // Almacena los nombres de los archivos restantes en el array
remainingFiles
        remainingFiles[remainingCount] = strdup(remainingEntry->d_name);
        remainingCount++;
    }

    // Cierra el directorio de archivos restantes
    closedir(remainingDir);

    if (remainingCount > 0) {
        // Crea un buffer para almacenar todos los números de los archivos
restantes
        int buffer[256];
        int totalCount = 0;

        // Procesa cada archivo restante
        for (i = 0; i < remainingCount; i++) {
            char* filename = remainingFiles[i];
            char file_path[256];
            snprintf(file_path, sizeof(file_path), "%s/%s", directory, filename);

            printf("Proceso padre: Archivo restante a procesar: %s\n", file_path);
```

```
            // Abre y lee el contenido del archivo
            FILE* file = fopen(file_path, "r");

            if (file != NULL) {
                // Lee los números del archivo y los agrega al buffer
                int count = 0;
                while (fscanf(file, "%d", &buffer[totalCount + count]) != EOF) {
                    count++;
                }

                totalCount += count;

                fclose(file);

                // Borra el archivo original
                remove(file_path);
            } else {
                printf("No se pudo abrir el archivo %s\n", file_path);
            }
        }

        // Ordena los números en el buffer de forma ascendente
        int j, temp;
        for (i = 0; i < totalCount - 1; i++) {
            for (j = 0; j < totalCount - i - 1; j++) {
                if (buffer[j] > buffer[j + 1]) {
                    temp = buffer[j];
                    buffer[j] = buffer[j + 1];
                    buffer[j + 1] = temp;
                }
            }
        }

        // Concatena los nombres de los archivos restantes para crear el nombre del
archivo nuevo
        char merged_filename[256] = "";
```

```
        for (i = 0; i < remainingCount; i++) {
            strcat(merged_filename, remainingFiles[i]);
        }
        strcat(merged_filename, "_merged_sorted.txt");

        // Crea un nuevo archivo para guardar los números ordenados
        char merged_filepath[256];
        snprintf(merged_filepath, sizeof(merged_filepath), "%s/%s", directory,
merged_filename);
        FILE* merged_file = fopen(merged_filepath, "w");

        if (merged_file != NULL) {
            // Escribe los números ordenados en el archivo
            for (i = 0; i < totalCount; i++) {
                fprintf(merged_file, "%d\n", buffer[i]);
            }

            fclose(merged_file);
        } else {
            printf("No se pudo crear el archivo %s\n", merged_filepath);
        }

        // Libera la memoria utilizada para almacenar los nombres de archivo
restantes
        for (i = 0; i < remainingCount; i++) {
            free(remainingFiles[i]);
        }

        printf("Proceso padre: Archivo nuevo creado: %s\n", merged_filepath);
    }

    // Libera la memoria utilizada para almacenar los nombres de archivo
    for (i = 0; i < fileCount; i++) {
        free(filePairs[i]);
    }

    return 0;
```

```
}

void readFilesInDirectory(const char* directory, const char* filename1, const char*
filename2) {
    // Construye las rutas completas de los archivos
    char file_path1[256];
    snprintf(file_path1, sizeof(file_path1), "%s/%s", directory, filename1);

    char file_path2[256];
    snprintf(file_path2, sizeof(file_path2), "%s/%s", directory, filename2);

    // Procesa los archivos
    printf("Proceso hijo: Archivos a procesar: %s, %s\n", file_path1, file_path2);


    FILE* file1 = fopen(file_path1, "r");
    FILE* file2 = fopen(file_path2, "r");

    if (file1 != NULL && file2 != NULL) {
        // Ordena los contenidos de los archivos
        int buffer[256];
        int count = 0;

        while (fscanf(file1, "%d", &buffer[count]) != EOF) {
            count++;
        }

        while (fscanf(file2, "%d", &buffer[count]) != EOF) {
            count++;
        }

        fclose(file1);
        fclose(file2);

        // Ordena los números en el buffer de forma ascendente
        int i, j, temp;
        for (i = 0; i < count - 1; i++) {
```

```
            for (j = 0; j < count - i - 1; j++) {
                if (buffer[j] > buffer[j + 1]) {
                    temp = buffer[j];
                    buffer[j] = buffer[j + 1];
                    buffer[j + 1] = temp;
                }
            }
        }

        // Crea un nuevo archivo para guardar los números ordenados
        char new_filename[256];
        snprintf(new_filename, sizeof(new_filename), "%s/merged_sorted_%s_%s.txt",
directory, filename1, filename2);
        FILE* merged_file = fopen(new_filename, "w");

        if (merged_file != NULL) {
            // Escribe los números ordenados en el archivo
            for (i = 0; i < count; i++) {
                fprintf(merged_file, "%d\n", buffer[i]);
            }

            fclose(merged_file);
        } else {
            printf("No se pudo crear el archivo %s\n", new_filename);
        }

        // Borra los archivos originales
        remove(file_path1);
        remove(file_path2);

    } else {
        printf("No se pudieron abrir los archivos %s, %s\n", file_path1,
file_path2);
    }

    printf("Proceso hijo: Finalizado\n\n");
}
```