Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

Sistemas Operativos

INFORME PROYECTO #2: MULTITHREADS

Estudiantes: César Carios

Jhonatan Homsany

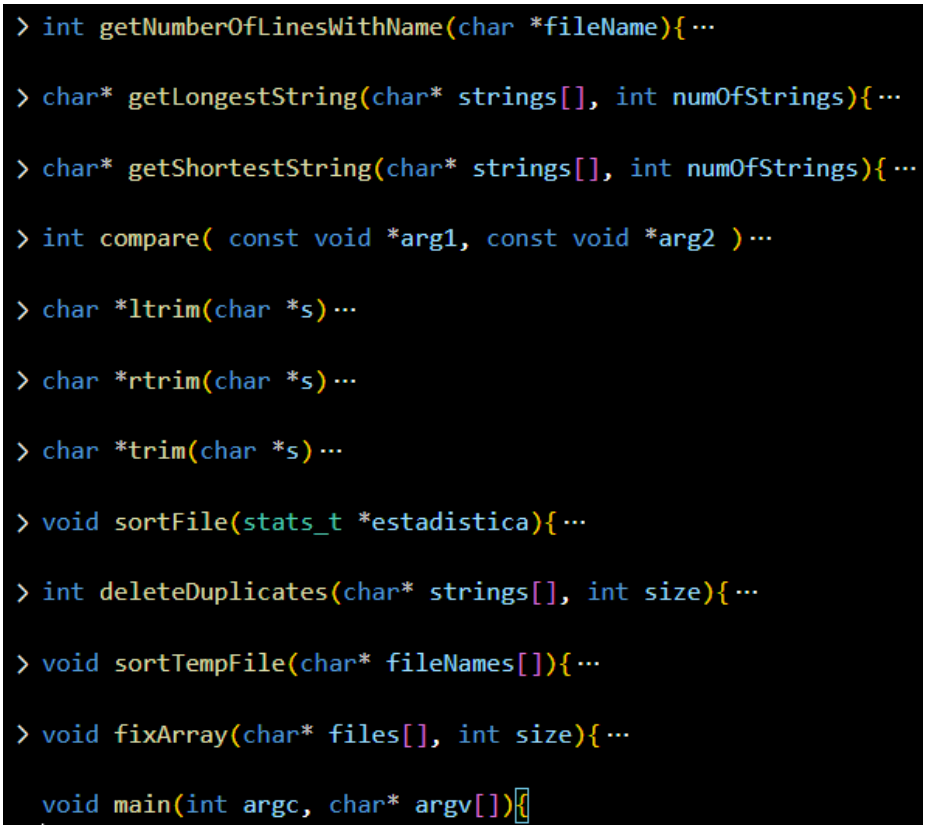
Estructura del código y principales funcionalidades

El código fuente de este proyecto consta de 3 variables globales y 12 funciones. Las variables globales son las siguientes:

```
int indexOfFirstFile = 0;
sem_t indexS;
int totalSorted = 0;
```

La primera variable hace lo que indica su nombre: funciona como índice/cursor en un arreglo de archivos cuya finalidad es llevar la cuenta de los archivos que han sido fusionados con otros. La segunda variable es un semáforo que nos ayudará para controlar el acceso a un recurso crítico, profundizaremos en él cuando sea oportuno. La tercera variable actúa como contador del total de líneas ordenadas.

Las funciones son las siguientes:



A continuación, se explicarán esas funciones una por una.

Funciones del código

- *Main:* En las primeras líneas del main se tiene lo siguiente:

```
sem_init(&indexS,0,1);

if (argc < 3){
    perror(0);
}

pthread_t workers[argc-1];
char* files[argc-1];
stats_t *estadisticas[argc-1];
```

Acá se inicializa el semáforo, en este caso es tipo mutex. Se valida con un condicional que se reciban al menos dos archivos por consola. Seguidamente se declaran los hilos que manejarán los archivos que se reciben por consola y por último tenemos un arreglo de structs para cada archivo que contiene como atributos sus estadísticas.

```
for(int i = 0; i < argc -1; i++){
    estadisticas[i] = malloc(sizeof(stats_t));
    estadisticas[i]->fileName = files[i];
    estadisticas[i]->sortedLines = 0;
    estadisticas[i]->longestString = NULL;
    estadisticas[i]->shortestString = NULL;
}

for(int i = 0; i < argc-1; i++){
    pthread_create(&workers[i], NULL, (void*) sortFile, estadisticas[i]);
}
for(int i = 0; i < argc-1; i++){
    pthread_join(workers[i], NULL);
}
```

Luego tenemos una estructura de control para inicializar los valores del struct, otra para crear la cantidad de hilos necesarios para manejar los archivos que se reciban y una último for para que cada hilo termine de ejecutar su trabajo.

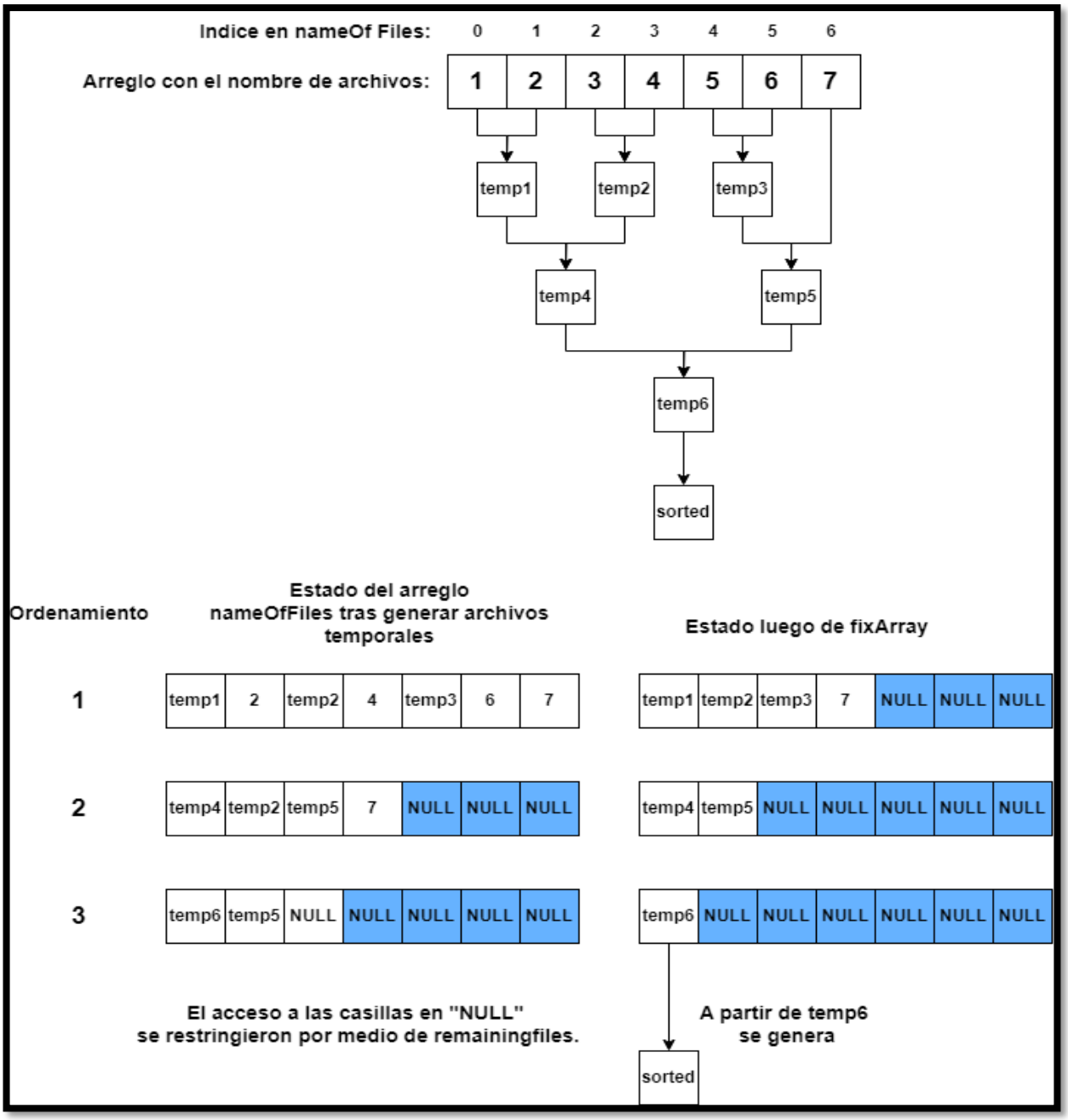
```
int remainingFiles = argc - 1;
pthread_t sortingThreads[remainingFiles/2];
char* nameOfFiles[remainingFiles];

for(int i = 0; i < argc-1; i++){
    nameOfFiles[i] = malloc(500 * sizeof(char));
    strcat(nameOfFiles[i], argv[i+1]);
    strcat(nameOfFiles[i], ".sorted");
}
```

Seguidamente tenemos la variable *remainingFiles* donde almacenamos cuántos hilos vamos a crear, en este caso si se tienen “n” cantidad de archivos, se crean n/2 hilos, por otro lado, *sortingThreads* es un arreglo donde se guardan los hilos encargados de ordenar y en *nameOfFiles* se guardan los nombres de los archivos a ordenar. Para guardar los nombres, se utiliza el bucle for que se muestra en la

imagen en el que se concatenan el nombre del archivo correspondiente en argv y el “.sorted”.

```
while(remainingFiles > 1){

    indexOfFirstFile = 0;

    for(int i = 0; i < remainingFiles/2; i++){
        pthread_create(&sortingThreads[i], NULL, (void*) sortTempFile, nameOfFiles);
        sleep(0);
        sem_wait(&indexS);
        indexOfFirstFile += 2;
        sem_post(&indexS);
    }

    for(int i = 0; i < remainingFiles/2; i++){
        pthread_join(sortingThreads[i], NULL);
    }

    remainingFiles-= remainingFiles/2;

    fixArray(nameOfFiles, argc-1);
}
```

Esta estructura de control es el motor de la fusión y el ordenamiento de los archivos, a continuación, se adjunta un diagrama para ilustrar lo que hace ese while:



En el ejemplo de la imagen se tienen 7 archivos, por lo que se crean 3 hilos (parte entera de remainingFiles/2). Los hilos se van reduciendo hasta que solo quedan dos temporales y se fusionan para crear el archivo final sorted.txt. En la segunda parte de la imagen se ilustra cómo se hizo el desplazamiento de los archivos dentro del arreglo para que se fusionaran de dos en dos. En el diagrama y en la imagen del código se llama a una función llamada *fixArray*, esta función será explicada más adelante.

```
int linesOfLastFile = getNumberOfLinesWithName(nameOfFiles[0])-1;
FILE* sorted = fopen("sorted.txt", "w+");
FILE* lastFile = fopen(nameOfFiles[0], "r");
char* string[linesOfLastFile];

for(int i = 0; i < linesOfLastFile; i++){
    string[i] = malloc(5000 * sizeof(char));
    fgets(string[i], 5000 * sizeof(char), lastFile);
}

qsort(string, linesOfLastFile, sizeof(char*), compare);

for(int i = 0; i < linesOfLastFile; i++){
    fprintf(sorted, "%s", string[i]);
}

printf("A total of %d strings were passed as input.\n", totalSorted);
printf("Longest string sorted: %s", getLongestString(string, linesOfLastFile-1));
printf("Shortest string sorted: %s", getShortestString(string, linesOfLastFile-1));
sem_destroy(&indexS);
fclose(sorted);
fclose(lastFile);
```

En estas últimas líneas del main, se abre el archivo sorted.txt para escribir en él lo que hay en el último archivo fusionado por los hilos, así que dicho archivo también se abre, pero solamente con permiso de lectura. Se obtienen los strings del último archivo fusionado, se ordena con la función *qsort* y luego se escribe en sorted.txt, Finalmente se imprimen las estadísticas pedidas: el total de líneas procesadas y el string más corto y más largo procesado.

- *Compare:* Como se pide ordenar los strings de forma lexicográfica decresciente, usamos la función strcasecmp, de esta forma lo ordenamos de manera lexicográfica creciente y tomando en cuenta la diferencia entre mayúsculas y minúsculas, además, multiplicamos la función por –1 para que se ordene de manera decreciente. Esta es la función que se pasa como parámetro al *qsort* del main.

```
int compare( const void *arg1, const void *arg2 )
{
   return -strcasecmp( * ( char** ) arg1, * ( char** ) arg2 );
}
```

- *GetNumberOfLinesWithName:* Esta función recibe el nombre de los archivos y cuenta la cantidad de líneas que tiene. Cada hilo la usa por separado con su propio archivo, esta función nos sirve para obtener el número total de líneas final.

```
int getNumberOfLinesWithName(char *fileName){

    int count = 0;
    FILE *fp = fopen(fileName, "r");
    char c;

    while(c = getc(fp)){
        if (c == '\n'){
            count += 1;
        }
        else if (c == EOF){
            count +=1;
            break;
        }
    }
    fclose(fp);

    return count;
}
```

- *GetLongestString:* Tal como indica su nombre, extrae el string más largo que ha sido ordenado. Esto lo hace en el archivo final: sorted.txt.

```
char* getLongestString(char* strings[], int numOfStrings){
    char* longestString = strings[0];

    for(int i = 1; i < numOfStrings; i++){
        if(strlen(longestString) < strlen(strings[i])){
            longestString = strings[i];
        }
        else if(strlen(longestString) == strlen(strings[i]) && strcasecmp(longestString, strings[i]) < 0){
            longestString = strings[i];
        }
    }

    return longestString;
}
```

Para implementar la extracción de la cadena más larga se declara un arreglo llamado *longestString* y usando la función *strlen()* se almacena allí la

cadena más larga leída a medida que se va avanzando en la lectura del sorted.txt

- *GetShortestString:* Se implementó de manera similar a la función para conseguir la cadena más larga, con la diferencia de que acá se usó la función *isSpace()* para no tomar en cuenta los espacios en blanco.

```
char* getShortestString(char* strings[], int numOfStrings){
    char* shortestString = strings[0];

    for(int i = 1; i < numOfStrings; i++){
        if(strlen(shortestString) > strlen(strings[i]) && isspace(*strings[i]) == 0){
            shortestString = strings[i];
        }
        else if(strlen(shortestString) == strlen(strings[i]) && strcasecmp(shortestString, strings[i]) < 0){
            shortestString = strings[i];
        }
    }

    return shortestString;
}
```

- *Familia de funciones Trim:* Para que no hubiera distinción entre cadenas que tuviesen los mismos caracteres a excepción de algún espacio al inicio o al final de ellas, se usaron las funciones *rtrim*, *ltrim* y *trim* para eliminar dichos espacios. De esa forma, las cadenas "casa", "_ casa" y "casa_ " son iguales para el programa. Las tres funciones reciben como parámetro la cadena que se está leyendo y le elimina los espacios indeseados, si es que los hay.

```
char *ltrim(char *s)
{
    while(isspace(*s)) s++;
    return s;
}

char *rtrim(char *s)
{
    char* back = s + strlen(s);
    while(isspace(*--back));
    *(back+1) = '\0';
    return s;
}

char *trim(char *s)
{
    return rtrim(ltrim(s));
}
```

- *SortedFile:* Esta función es la encargada de ordenar cada archivo individual recibido por consola. Inicialmente abre el archivo original y abre un archivo en modo de escritura donde se almacenan los strings ordenados (si este archivo no existe, lo crea). Luego, guarda la cantidad de líneas en el archivo usando la función getNumberOfLinesWithName(). Conociendo el número de líneas que se leerán, es posible almacenar cada una de ellas en un arreglo empleando un bucle for. Al leer cada string, nos aseguramos de incrementar una variable global que lleva el registro de la cantidad de strings leídos en total por cada uno de los hilos, dado que esta variable es un recurso compartido por los hilos, se emplearon semáforos para poder modificar su valor. En este paso es importante verificar si el string leído no corresponde a un espacio vacío, en cuyo caso, se elimina del arreglo y se evita que las variables contadoras aumenten. Tras tener el archivo almacenado en un arreglo, lo ordenamos empleando la función *qsort* con el método *compare* definido anteriormente. Por último, transcribimos cada una de las palabras al archivo ".sorted" que se debe generar y guardamos las estadísticas declaradas en el archivo pf1.h. Antes de que la función

finalice, cerramos los archivos abiertos e imprimimos el mensaje correspondiente. Además, el hilo trabajador finaliza.

```
void sortFile(stats_t *estadistica){
    FILE* original = fopen((*estadistica).fileName, "r");
    char* sortedFileName = malloc(1000);
    strcat(sortedFileName, (*estadistica).fileName);
    strcat(sortedFileName, ".sorted");
    FILE* ordenado = fopen(sortedFileName, "w+");
    char* stringArray[5000];
    unsigned int numOfLines = getNumberOfLinesWithName((*estadistica).fileName);

    for(int i = 0; i < numOfLines; i++){
        stringArray[i] = malloc(5000);
        fgets(stringArray[i], 5000, original);
        stringArray[i] = trim(stringArray[i]);
        if(strlen(stringArray[i]) == 0){
            numOfLines--;
            i--;
            sem_wait(&indexS);
            totalSorted--;
            sem_post(&indexS);
        }
        sem_wait(&indexS);
        totalSorted++;
        sem_post(&indexS);
    }
```

```
qsort(stringArray, numOfLines, sizeof(malloc(5000)), compare);
for(int i = 0; i < numOfLines; i++){
    fprintf(ordenado, "%s\n", stringArray[i]);
}

(*estadistica).sortedLines = numOfLines;
(*estadistica).shortestString = malloc(5000 * sizeof(char));
(*estadistica).shortestString = getShortestString(stringArray, numOfLines);
(*estadistica).longestString = malloc(5000 * sizeof(char));
(*estadistica).longestString = getLongestString(stringArray, numOfLines);

fclose(original);
fclose(ordenado);
printf("This worker thread writes %d lines to %c%s%c\n", numOfLines, '"', sortedFileName, '"');
pthread_exit(NULL);
```

- *DeleteDuplicates:* Como se puede inferir por su nombre, esta función elimina las palabras duplicadas en un mismo archivo. Esto se logra haciendo uso de *strcasecmp*, en cada archivo temporal creado (los archivos resultantes de la fusión con los hilos) se comparan cada una de sus cadenas y se elimina una de ellas si hay algún duplicado.

```
int deleteDuplicates(char* strings[], int size){
    int linesDeleted = 0;

    for(int i = 0; i < size; i++){
        for(int j = 0; j < i; j++){
            if(strcasecmp(strings[i], strings[j]) == 0){
                for (int k = i; k < size - 1; k++) {
                    strings[k] = strings[k + 1];
                }
                linesDeleted++;
                size--;
                i--;
                break;
            }
        }
    }

    return linesDeleted;
}
```

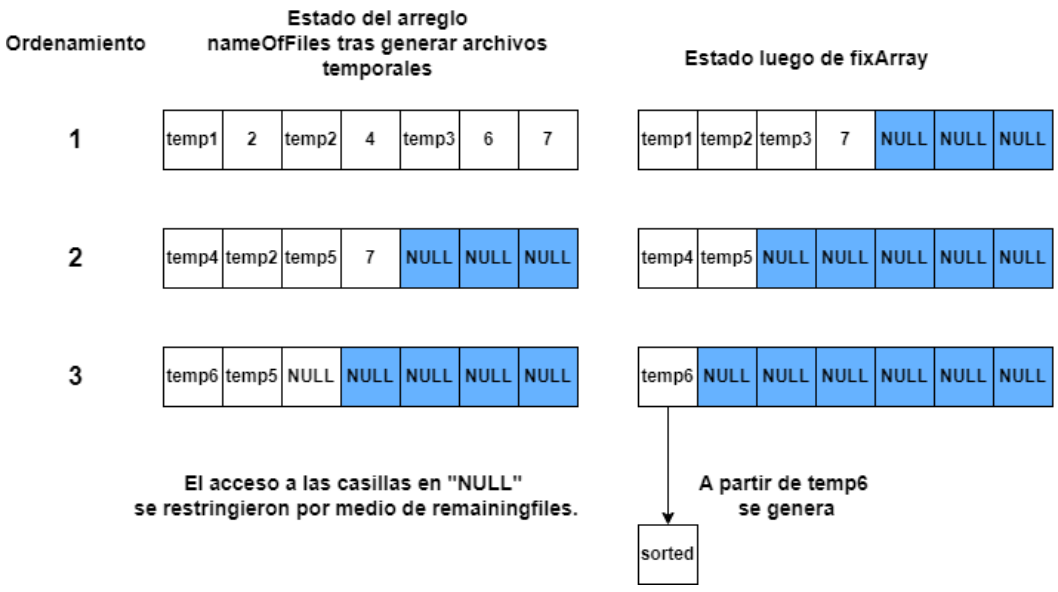
- *SortTempFile:* Para que cada hilo tenga conocimiento de los archivos con los que debe trabajar, utilizamos una variable global llamada indexOfFirstFile. La variable se encuentra inicializada en 0 e incrementa en 2 luego de que cada hilo es creado, así, el hilo obtiene que estará trabajando con los archivos que están en la posición indicada por esta variable y la posición que le sigue (indexOfFirstFile+1). Como el hilo principal se mantiene modificando el valor de esta variable antes de crear cada hilo, es importante mantener un control de concurrencia ya que una modificación anticipada de la variable podría provocar que el hilo trabaje sobre un archivo diferente, inclusive, dos hilos podrían estar trabajando con los mismos archivos, para ello se utilizaron semáforos. Para poder llevar a cabo la tarea de crear los archivos temporales y guardarlos en el arreglo que contiene los nombres de los archivos, se utilizó la función tmpnam, la cual genera un nombre para el archivo temporal y lo elimina del computador una vez este es cerrado.

Tras tener abiertos los diferentes archivos con los que trabajaremos, almacenamos la longitud de los archivos que vamos a unir. Luego, guardamos en un arreglo de strings las palabras en cada uno de ellos con la ayuda de un bucle for. Posteriormente, realizamos la tarea de eliminar los duplicados en dicho arreglo, para ello usamos la función deleteDuplicates() explicada anteriormente. Tras esto, es necesario imprimir en el archivo temporal los resultados, para ello necesitamos la cantidad de líneas a escribir, computada como la suma de los dos archivos a unir menos la cantidad de lineas duplicadas. Finalmente, imprimimos los resultados, modificamos el arreglo con los nombres de archivos temporales, liberamos el semaforo y cerramos los archivos.

```
void sortTempFile(char* fileNames[]){

    sem_wait(&indexS);

    char* stringFile[5000];
    FILE* firstFile = fopen(fileNames[indexOfFirstFile], "r");
    FILE* secondFile = fopen(fileNames[indexOfFirstFile+1], "r");
    char tempFileName[L_tmpnam];
    tmpnam(tempFileName);
    FILE* temp = fopen(tempFileName, "w+");

    int linesFirstFile = getNumberOfLinesWithName(fileNames[indexOfFirstFile])-1;
    int linesSecondFile = getNumberOfLinesWithName(fileNames[indexOfFirstFile+1])-1;

    for(int i = 0; i < linesFirstFile + linesSecondFile; i++){
        if(i < linesFirstFile){
            stringFile[i] = malloc(5000);
            fgets(stringFile[i], 5000, firstFile);
        }
        else{
            stringFile[i] = malloc(5000);
            fgets(stringFile[i], 5000, secondFile);
        }
    }
```

```c
int linesDeleted = deleteDuplicates(stringFile, linesFirstFile + linesSecondFile);
int linesFinalFile = linesFirstFile+linesSecondFile-linesDeleted;

for(int i = 0; i < linesFinalFile; i++){
    if(stringFile[i] != NULL){
        if(strlen(stringFile[i]) > 0 && (*stringFile[i]) != '\n'){
            fprintf(temp, "%s", stringFile[i]);
        }
    }
}

printf("Merged %d lines and %d lines into %d lines.\n", linesFirstFile, linesSecondFile, linesFinalFile);

fileNames[indexOfFirstFile] = strdup(tempFileName);

sem_post(&indexS);
fclose(firstFile);
fclose(secondFile);
fclose(temp);
pthread_exit(NULL);
```

- *FixArray:* Tras hacer la unión de dos archivos del tipo ".sorted" es necesario acomodar el arreglo de manera que cuando se vuelvan a crear los hilos, cada uno obtenga los archivos correspondientes. Para solucionar el problema, se implementó la función fixArray(). Esta función se encarga de eliminar los archivos en las posiciones impares del arreglo y de desplazar los archivos en posiciones posteriores de manera que todos los archivos queden en posiciones contiguas del arreglo. Retomemos una imagen que se ha visto anteriormente en este documento para reflejar el estado del arreglo con los nombres de los archivos tras unir dos archivos y tras hacer el llamado a fixArray:



Como se puede ver, los archivos en las posiciones impares del arreglo se convierten en NULL y se desplazan al final del arreglo. En iteraciones futuras, es posible asegurarse que esas casillas con punteros a direcciones nulas no serán referenciadas debido a la forma en la que se decrementa la variable remainingFiles en el ciclo donde se llama a esta función.

```
void fixArray(char* files[], int size){

    int desplazamiento = 0;
    char* aux = malloc(5000 * sizeof(char));

    for(int i = 0; i < size; i++){
        if(i % 2 == 0){
            aux = files[i];
            files[i] = NULL;
            files[i-desplazamiento] = aux;
            desplazamiento++;
        }
        else{
            files[i] = NULL;
        }
    }
}
```

Consideraciones Finales

Este proyecto se desarrolló en el subsistema de Linux en Windows (WSL), usando el distrito de Ubuntu, concretamente en su versión 22.04.4 LTS, por lo que compila sin problemas en dicho distrito. Sin embargo, presenta problemas al intentar compilar en Lubuntu y en Mint, se recomienda solo compilar en Ubuntu.

Caso de Prueba

A continuación, se prueba el archivo comprimido suministrado por el grupo docente en el canal de telegram.

- Compilación con el comando make:

```
ccarios@LAPTOP-OK5E59NR:~/Sistemas Operativos Linux/Multithreads/Multithreads-SO$ make
gcc -c pf1.c
pf1.c: In function ‘main’:
pf1.c:218:9: warning: implicit declaration of function ‘error’; did you mean ‘perror’? [-Wimplicit-function-declaration]
  218 |         error(0);
      |         ^~~~~
      |         perror
pf1.c:260:13: warning: implicit declaration of function ‘sleep’ [-Wimplicit-function-declaration]
  260 |             sleep(0);
      |             ^~~~~
gcc -o pf1 pf1.o
/usr/bin/ld: pf1.o: in function `sortTempFile':
pf1.c:(.text+0x8ca): warning: the use of `tmpnam' is dangerous, better use `mkstemp'
```

- Ejecución del programa con los 6 archivos de texto suministrados:

```
ccarios@LAPTOP-OK5E59NR:~/Sistemas Operativos Linux/Multithreads/Multithreads-SO$ ./pf1 a1.txt a2.txt a3.txt a4.txt a5.txt a6.txt
This worker thread writes 89 lines to "a1.txt.sorted"
This worker thread writes 83 lines to "a5.txt.sorted"
This worker thread writes 55 lines to "a4.txt.sorted"
This worker thread writes 280 lines to "a6.txt.sorted"
This worker thread writes 115 lines to "a3.txt.sorted"
This worker thread writes 311 lines to "a2.txt.sorted"
Merged 89 lines and 311 lines into 399 lines.
Merged 115 lines and 55 lines into 170 lines.
Merged 83 lines and 280 lines into 363 lines.
Merged 399 lines and 170 lines into 569 lines.
Merged 569 lines and 363 lines into 931 lines.
A total of 933 strings were passed as input.
Longest string sorted: Posterioristically
Shortest string sorted: Tao
```