

PRÁCTICA 5: ADMINISTRADOR DE PROCESOS EN LINUX Y WINDOWS (2)

ALUMNO: BASTIDA PRADO JAIME ARMANDO

PROFESOR: CORTÉS GALICIA JORGE

MATERIA: SISTEMAS OPERATIVOS

GRUPO: 2CM9

Mayo 2018

Índice

1. Competencias	3
2. Desarrollo	3
2.1. Punto 1	3
2.2. Punto 5	4
2.2.1. Linux	4
2.2.2. Funcionamiento	4
2.2.3. Código	4
2.2.4. Windows	6
2.2.5. Funcionamiento	6
2.2.6. Código Padre	7
2.2.7. Código Hijo	7
2.3. Punto 6	9
2.3.1. Linux	9
2.3.2. Funcionamiento	9
2.3.3. Observaciones	11
2.3.4. Código	11
2.3.5. Windows	18
2.3.6. Funcionamiento	18
2.3.7. Observaciones	19
2.3.8. Código	20
2.4. Punto 7	26
2.4.1. Linux	26
2.4.2. Funcionamiento	26
2.4.3. Observaciones	27
2.4.4. Código	27
2.4.5. Windows	30
2.4.6. Funcionamiento	30
2.4.7. Observaciones	31
2.4.8. Código	31
3. Análisis Crítico	34
4. Conclusión	34

1. Competencias

El alumno aprende a familiarizarse con el administrador de procesos del sistema operativo en Linux y Windows a través de la creación de nuevos procesos ligeros (hilos) para el desarrollo de aplicaciones concurrentes sencillas.

2. Desarrollo

2.1. Punto 1

A través de la ayuda en línea que proporciona Linux, investigue el funcionamiento de las funciones: **pthread_create()**, **pthread_join()**, **pthread_self()**, **pthread_exit()**, **scandir()**, **stat()**. Explique los argumentos y retorno de cada función.

- **int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)**

La función retorna 0 de haber sido exitosa la llamada y un número de error de haber sido fallida.

El argumento thread es la variable que contiene el ID del hilo.

El argumento attr es un apuntador a una estructura de tipo pthread_attr_t cuyos contenidos son usados a la hora de crear el hilo, si el valor de este argumento es NULL, los atributos son puestos en default.

El argumento start_routine es la función que ejecutará el hilo.

El último argumento arg es un apuntador a los argumentos los cuales serán pasados a la función que ejecutará el hilo.

- **int pthread_join(pthread_t thread, void **retval)**

La función retorna 0 de haber sido exitosa la llamada y un número de error de haber sido fallida.

El argumento thread es la variable que contiene el ID del hilo al cual va a esperar la función a terminar.

El argumento retval contendrá después de la llamada el valor de retorno del hilo al cual se espero a unir.

- **pthread_t pthread_self(void)**

La función retorna el ID del hilo que mandó a llamar la función.

- **void pthread_exit(void *retval)**

El argumento retval es el valor que el hilo regresará antes de terminar y que estará disponible para cualquier otro hilo que se una con él.

- **int scandir(const char *dirp, struct dirent **namelist)**

La función regresa el número de entradas dentro del directorio escaneado seleccionadas, si falla regresa -1.

El argumento dirp contiene el directorio el cual se va a escanear.

El argumento namelist contendrá los nombres de todas las entradas dentro del directorio.

- **int stat(const char *pathname, struct stat *statbuf)**

De ser exitosa la llamada la función devuelve 0, -1 caso contrario.

El argumento pathname contiene la dirección del archivo.

El argumento statbuf es un apuntador a una estructura del tipo struct stat que almacenará información sobre el archivo especificado en pathname dentro de la estructura.

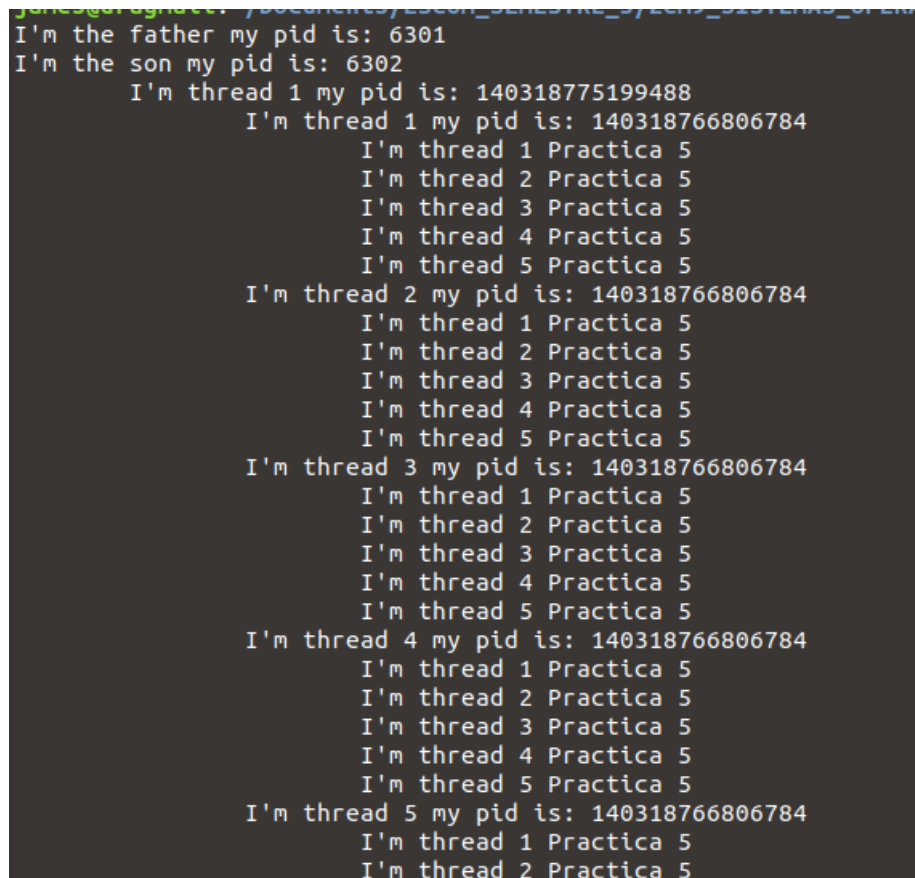
2.2. Punto 5

Programe una aplicación (tanto en Linux como en Windows), que cree un proceso hijo a partir de un proceso padre, el hijo creado a su vez creará 15 hilos. A su vez cada uno de los 15 hilos creará 10 hilos más. A su vez cada uno de los 10 hilos creará 5 hilos más. Cada uno de los hilos creados imprimirá en pantalla "Práctica 5" si se trata de un hilo terminal o los identificadores de los hilos creados si se trata de un proceso o hilo padre.

2.2.1. Linux

2.2.2. Funcionamiento

Corremos el programa e inmediatamente después podremos ver la impresión de cada hilo o proceso creado con su ID o mensaje "Practica 5" si se trata de un hilo terminal.



```
jones@ubuntu: ~/Documents/Leccion_5/Leccion_5/Practica_5
I'm the father my pid is: 6301
I'm the son my pid is: 6302
    I'm thread 1 my pid is: 140318775199488
        I'm thread 1 my pid is: 140318766806784
            I'm thread 1 Practica 5
            I'm thread 2 Practica 5
            I'm thread 3 Practica 5
            I'm thread 4 Practica 5
            I'm thread 5 Practica 5
        I'm thread 2 my pid is: 140318766806784
            I'm thread 1 Practica 5
            I'm thread 2 Practica 5
            I'm thread 3 Practica 5
            I'm thread 4 Practica 5
            I'm thread 5 Practica 5
        I'm thread 3 my pid is: 140318766806784
            I'm thread 1 Practica 5
            I'm thread 2 Practica 5
            I'm thread 3 Practica 5
            I'm thread 4 Practica 5
            I'm thread 5 Practica 5
        I'm thread 4 my pid is: 140318766806784
            I'm thread 1 Practica 5
            I'm thread 2 Practica 5
            I'm thread 3 Practica 5
            I'm thread 4 Practica 5
            I'm thread 5 Practica 5
        I'm thread 5 my pid is: 140318766806784
            I'm thread 1 Practica 5
            I'm thread 2 Practica 5
```

Figura 1:

2.2.3. Código

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/wait.h>

void *thread(void *arg);
void *thread1(void *arg);
void *thread2(void *arg);
```

```

int main(void)
{
    pid_t pid;
    pthread_t id_thread;
    int *no;

    if((pid = fork()) == 0)
    {
        pid = getpid();
        printf("I'm the son my pid is: %d\n", pid);
        for(int i = 0; i < 15; i++)
        {
            *no = i + 1;
            pthread_create(&id_thread, NULL, thread, no);
            pthread_join(id_thread, NULL);
        }
        exit(EXIT_SUCCESS);
    }
    else
    {
        pid = getpid();
        printf("I'm the father my pid is: %d\n", pid);
        wait(0);
    }

    return 0;
}

void *thread(void *arg)
{
    int *no = arg;
    pthread_t id_thread = pthread_self();
    printf("\tI'm thread %d my pid is: %d\n", *no, id_thread);

    for(int i = 0; i < 10; i++)
    {
        *no = i + 1;
        pthread_create(&id_thread, NULL, thread1, no);
        pthread_join(id_thread, NULL);
    }
    return NULL;
}

void *thread1(void *arg)
{
    int *no = arg;
    pthread_t id_thread = pthread_self();
    printf("\t\tI'm thread %d my pid is: %d\n", *no, id_thread);

    for(int i = 0; i < 5; i++)
    {
        *no = i + 1;
        pthread_create(&id_thread, NULL, thread2, no);
        pthread_join(id_thread, NULL);
    }
    return NULL;
}

void *thread2(void *arg)
{
    int *no = arg;
    pthread_t id_thread = pthread_self();
    printf("\t\t\tI'm thread %d Practica 5\n", *no);

    return NULL;
}

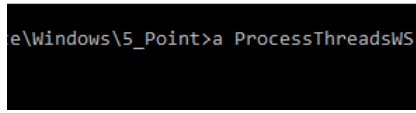
```

2.2.4. Windows

2.2.5. Funcionamiento

Para correr el programa deberemos teclear el nombre de la aplicación (proceso padre), en este caso *a* y dejar un espacio para después teclear el nombre de la segunda aplicación, es decir el proceso hijo, porque como sabemos en Windows la única forma de crear procesos es por sustitución de código. Este proceso hijo es el que mandará a llamar los hilos. Después de seguir la instrucciones anteriores damos enter e inmediatamente después podremos ver la impresión de cada hilo o proceso creado con su ID o mensaje "Practica 5" si se trata de un hilo terminal.

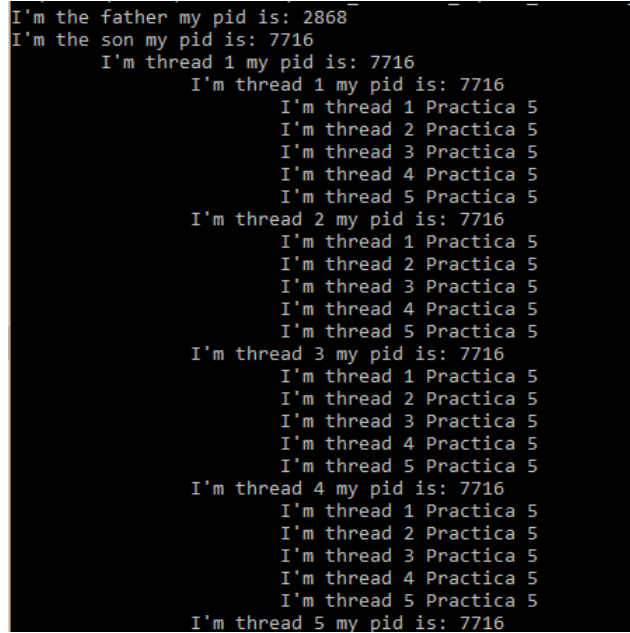
A continuación se puede apreciar como correr el programa:



```
e\Windows\5_Point>a ProcessThreadsWS
```

Figura 2:

La salida correspondiente en pantalla



```
I'm the father my pid is: 2868
I'm the son my pid is: 7716
  I'm thread 1 my pid is: 7716
    I'm thread 1 my pid is: 7716
      I'm thread 1 Practica 5
      I'm thread 2 Practica 5
      I'm thread 3 Practica 5
      I'm thread 4 Practica 5
      I'm thread 5 Practica 5
    I'm thread 2 my pid is: 7716
      I'm thread 1 Practica 5
      I'm thread 2 Practica 5
      I'm thread 3 Practica 5
      I'm thread 4 Practica 5
      I'm thread 5 Practica 5
    I'm thread 3 my pid is: 7716
      I'm thread 1 Practica 5
      I'm thread 2 Practica 5
      I'm thread 3 Practica 5
      I'm thread 4 Practica 5
      I'm thread 5 Practica 5
    I'm thread 4 my pid is: 7716
      I'm thread 1 Practica 5
      I'm thread 2 Practica 5
      I'm thread 3 Practica 5
      I'm thread 4 Practica 5
      I'm thread 5 Practica 5
    I'm thread 5 my pid is: 7716
```

Figura 3:

2.2.6. Código Padre

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

int main(int argc, char *argv[])
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    DWORD pid;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // Creaci n proceso hijo
    if(!CreateProcess(NULL, argv[1], NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
    {
        printf("Error: _cannot_invoke_CreateProcess(%d)\n", GetLastError());
        return 0;
    }

    // Proceso padre
    pid = GetCurrentProcessId();
    printf("I'm the _father _my _pid _is: _%d\n", pid);
    WaitForSingleObject(pi.hProcess, INFINITE);

    // Terminaci n controlada del proceso e hilo asociado de ejecuci n
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    return 0;
}
```

2.2.7. Código Hijo

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

DWORD WINAPI thread(LPVOID arg);
DWORD WINAPI thread1(LPVOID arg);
DWORD WINAPI thread2(LPVOID arg);

int main(void)
{
    DWORD pid, id_thread;
    HANDLE han_thread;
    int *no, j;

    pid = GetCurrentProcessId();
    printf("I'm the _son _my _pid _is: _%d\n", pid);
    for(int i = 0; i < 15; i++)
    {
        // *no = i + 1;
        j = i + 1;
        han_thread = CreateThread(NULL, 0, thread, &j, 0, &id_thread);
        WaitForSingleObject(han_thread, INFINITE);
    }

    return 0;
}

DWORD WINAPI thread(LPVOID arg)
{
    int *no = arg, j;
```

```

DWORD id_thread = GetCurrentProcessId();
HANDLE han_thread;

printf("\tI'm_thread_%d_my_pid_is:_%d\n", *no, id_thread);

for(int i = 0; i < 10; i++)
{
    // *no = i + 1;
    j = i + 1;
    han_thread = CreateThread(NULL, 0, thread1, &j, 0, &id_thread);
    WaitForSingleObject(han_thread, INFINITE);
}
return 0;
}

DWORD WINAPI thread1(LPVOID arg)
{
    int *no = arg, j;
    DWORD id_thread = GetCurrentProcessId();
    HANDLE han_thread;

    printf("\t\tI'm_thread_%d_my_pid_is:_%d\n", *no, id_thread);

    for(int i = 0; i < 5; i++)
    {
        // *no = i + 1;
        j = i + 1;
        han_thread = CreateThread(NULL, 0, thread2, &j, 0, &id_thread);
        WaitForSingleObject(han_thread, INFINITE);
    }
    return 0;
}

DWORD WINAPI thread2(LPVOID arg)
{
    int *no = arg;
    DWORD id_thread = GetCurrentProcessId();
    printf("\t\t\tI'm_thread_%d_Practica_5\n", *no);

    return 0;
}

```


2.3. Punto 6

Programa la misma aplicación del punto 5 de la práctica 4 pero utilizando hilos (tanto en Linux como en Windows) en vez de procesos. Compare ambos programas (el creado en la práctica 4 y el creado en esta práctica) y dé sus observaciones tanto de funcionamiento como de los tiempos de ejecución resultantes.

2.3.1. Linux

2.3.2. Funcionamiento

Corremos el programa y enseguida veremos en pantalla el orden en el que fueron llamados los hilos:

```
SUBTRACTION
TRANSPOSE
INVERSE
SUM
MULTIPLICATION
THREAD 6
```

Figura 4:

Posteriormente la salida en pantalla correspondiente de cada operación: La suma y resta de matrices:

```

              SUM
7    12   12    6    5    3    9    9   12    3
4    7    7    6   13    4    2    9    7    5
5   12   12   15    3   10    7   16   10    7
8   11    8   11    3   14    4   12   10   10
11  3    5    8   10    9    6    1   11   14
5   10   10   10   5    2    9    9    3    8
13   7    8    7   13   10    9    5    9    8
13   7    5   13   14    8    8   13   17    9
18   8    9    4   14    7   10    6    7    2
8   15    3    7    4   10    3    7    9    5

              SUBTRACTION
5    4    6   -2    1    3   -5   -7    4    3
-2   -3   -5    0   -1    0   -2    7    7    5
5   -6    4   -1   -1    6    3    2    6   -3
-4    7    8    1    1    0    4    4    0   -6
5   -3    5   -4   -2   -3   -2    1   -3    0
-5    2    0    6   -1   -2    3    3    3   -6
1    5   -6   -3   -5   -2    3   -3    7    4
-5    3   -1   -1   -4    4    4   -1    1   -1
0   -4   -1   -4   -2    5    8    0    5   -2
-6   -3   -1    5    0    6    3   -7    5   -3
```

Figura 5:

La multiplicación y transpuestas:

MULTIPLICATION									
180	249	180	206	212	100	113	196	97	140
223	158	98	162	201	100	66	147	144	159
296	245	211	272	278	153	126	272	167	238
261	239	208	253	274	151	110	227	145	217
153	196	110	152	134	86	103	192	108	135
196	142	133	195	206	143	63	155	123	155
228	209	178	171	235	118	119	186	102	134
299	218	193	231	289	163	119	214	156	211
198	175	187	212	250	126	156	195	125	149
189	147	141	132	173	111	69	109	62	147

TRANPOSE OF MATRIX 1									
6	1	5	2	8	0	7	4	9	1
1	2	3	9	0	6	6	5	2	6
5	3	8	8	5	5	1	2	4	1
2	9	8	6	2	8	2	6	0	6
8	0	5	2	4	2	4	5	6	2
0	6	5	8	2	0	4	6	6	8
7	6	1	2	4	4	6	6	9	3
4	5	2	6	5	6	6	6	3	0
9	2	4	0	6	6	9	3	6	7
1	6	1	6	2	8	3	0	7	1

TRANPOSE OF MATRIX 2									
1	3	0	6	3	5	6	9	9	7
3	5	9	2	3	4	1	2	6	9

Figura 6:

Las inversas:

INVERSE OF MATRIX 1									
6	1	5	2	8	0	7	4	9	1
1	2	3	9	0	6	6	5	2	6
5	3	8	8	5	5	1	2	4	1
2	9	8	6	2	8	2	6	0	6
8	0	5	2	4	2	4	5	6	2
0	6	5	8	2	0	4	6	6	8
7	6	1	2	4	4	6	6	9	3
4	5	2	6	5	6	6	6	3	0
9	2	4	0	6	6	9	3	6	7
1	6	1	6	2	8	3	0	7	1

INVERSE OF MATRIX 2									
1	3	0	6	3	5	6	9	9	7
3	5	9	2	3	4	1	2	6	9
0	9	4	0	0	5	7	3	5	2
6	2	0	5	6	2	5	7	4	1
3	3	0	6	6	3	9	9	8	2
5	4	5	2	3	2	6	2	1	2
6	1	7	5	9	6	3	2	1	0
9	2	3	7	9	2	2	7	3	7
9	6	5	4	8	1	1	3	1	2
7	9	2	1	2	2	0	7	2	4

Figura 7:

Los archivos creados por cada operación dentro del directorio de trabajo:

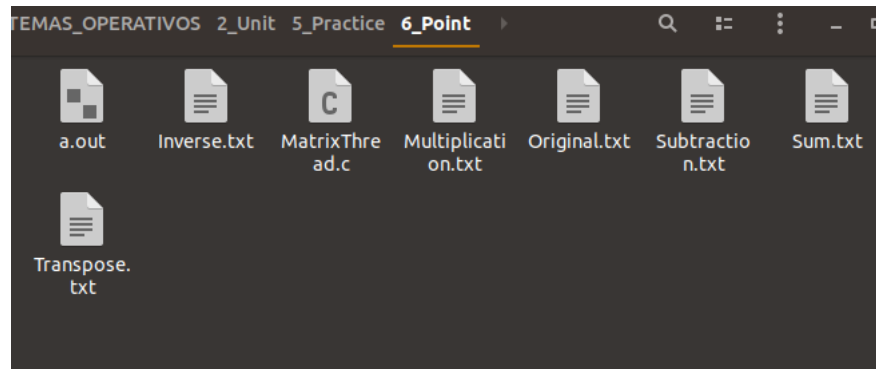


Figura 8:

Y el tiempo de ejecución:

```
Execution time: 0.003162 seconds
```

Figura 9:

2.3.3. Observaciones

El funcionamiento de esta práctica es muy similar al de procesos concurrentes normales excepto que se tienen que agregar ciertos pasos como las funciones que ejecutarán los hilos, apuntadores para pasar los argumentos a dichas funciones, etc.

A pesar de haberse programado con hilos las funciones de la aplicación se puede apreciar una diferencia aunque no muy grande si notable de tiempos de ejecución si comparamos la programación de esta aplicación con hilos y con procesos, siendo así que con procesos el tiempo de ejecución es de aproximadamente 0.001369 segundos mientras que con hilos es de aproximadamente 0.003162 segundos.

2.3.4. Código

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pthread.h>

#define SIZE 10

typedef unsigned char BYTE;

int determinant(int **matrix, int size);
void *sum(void *arg);
void *subtraction(void *arg);
void *multiplication(void *arg);
void *transpose(void *arg);
void *inverse(void *arg);

typedef struct
{
    int **matrix1;
    int **matrix2;
} Matrices;
```

```

int main(void)
{
    pid_t son_pid;
    pthread_t id_thread[5];
    BYTE i = 0, j = 0, k = 0, l = 0, m = 0;
    int **matrix1, **matrix2, result_matrix[SIZE][SIZE];
    clock_t begin, end;
    FILE *write_fp, *read_fp;

    Matrices argv;
    Matrices *argvp;

    //Create the matrices
    matrix1 = malloc(SIZE * sizeof(int *));
    for(i = 0; i < SIZE; i++)
        matrix1[i] = malloc(SIZE * sizeof(int));

    matrix2 = malloc(SIZE * sizeof(int *));
    for(i = 0; i < SIZE; i++)
        matrix2[i] = malloc(SIZE * sizeof(int));

    //Fill the matrices
    srand((unsigned) time(NULL));
    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
        {
            matrix1[i][j] = rand() % 10;
            matrix2[i][j] = rand() % 10;
        }

    //Copy the reference of the two matrices into the struct argv for further
    //argument to the threads
    argv.matrix1 = matrix1;
    argv.matrix2 = matrix2;

    //Copy the reference of the struct into argvp
    argvp = &argv;

    //Print the original two matrices
    write_fp = fopen("Original.txt", "w");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            fprintf(write_fp, "%d_", matrix1[i][j]);
        fprintf(write_fp, "\n");
    }
    fprintf(write_fp, "\n");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            fprintf(write_fp, "%d_", matrix2[i][j]);
        fprintf(write_fp, "\n");
    }
    fclose(write_fp);

    begin = clock();
    //Call 5 processes to run
    for(i = 0; i < 5; i++)
    {
        switch(i)
        {
            case 0:
                pthread_create(&id_thread[0], NULL, sum, argvp);
                break;
            case 1:
                pthread_create(&id_thread[1], NULL, subtraction,
                    argvp);
                break;
            case 2:

```

```

        pthread_create(&id_thread[2], NULL, multiplication,
                        argvp);
        break;
    case 3:
        pthread_create(&id_thread[3], NULL, transpose,
                        argvp);
        break;
    case 4:
        pthread_create(&id_thread[4], NULL, inverse, argvp);
        break;
    }
}
for(i = 0; i < 4; i++)
    pthread_join(id_thread[i], NULL);
printf("THREAD_6\n");

read_fp = fopen("Sum.txt", "rb");
for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        fscanf(read_fp, "%d", &result_matrix[i][j]);
fclose(read_fp);

printf("\t\t\tSUM\n");
for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
        printf("%5d", result_matrix[i][j]);
    printf("\n");
}

read_fp = fopen("Subtraction.txt", "rb");
for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        fscanf(read_fp, "%d", &result_matrix[i][j]);
fclose(read_fp);

printf("\n\t\t\tSUBTRACTION\n");
for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
        printf("%5d", result_matrix[i][j]);
    printf("\n");
}

read_fp = fopen("Multiplication.txt", "rb");
for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        fscanf(read_fp, "%d", &result_matrix[i][j]);
fclose(read_fp);

printf("\n\t\t\tMULTIPLICATION\n");
for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
        printf("%5d", result_matrix[i][j]);
    printf("\n");
}

read_fp = fopen("Transpose.txt", "rb");
for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        fscanf(read_fp, "%d", &matrix1[i][j]);
for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        fscanf(read_fp, "%d", &matrix2[i][j]);
fclose(read_fp);

printf("\n\t\t\tTRANSPOSE_OF_MATRIX_1\n");
for(i = 0; i < SIZE; i++)

```



```

        fclose(write_fp);
    }

    void *subtraction(void *arg)
    {
        Matrices *argv = arg;
        int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE];
        FILE *write_fp, *read_fp;

        for(int i = 0; i < SIZE; i++)
            for(int j = 0; j < SIZE; j++)
            {
                matrix1[i][j] = argv->matrix1[i][j];
                matrix2[i][j] = argv->matrix2[i][j];
            }

        printf("SUBTRACTION\n");
        write_fp = fopen("Subtraction.txt", "w");
        for(int j = 0; j < SIZE; j++)
            for(int k = 0; k < SIZE; k++)
                matrix1[j][k] = matrix1[j][k] - matrix2[j][k];

        for(int j = 0; j < SIZE; j++)
        {
            for(int k = 0; k < SIZE; k++)
                fprintf(write_fp, "%d_", matrix1[j][k]);
            fprintf(write_fp, "\n");
        }
        fclose(write_fp);
    }

    void *multiplication(void *arg)
    {
        Matrices *argv = arg;
        int accumulator = 0, result_matrix[SIZE][SIZE];
        int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE];
        FILE *write_fp, *read_fp;

        for(int i = 0; i < SIZE; i++)
            for(int j = 0; j < SIZE; j++)
            {
                matrix1[i][j] = argv->matrix1[i][j];
                matrix2[i][j] = argv->matrix2[i][j];
            }

        printf("MULTIPLICATION\n");
        write_fp = fopen("Multiplication.txt", "w");
        for(int j = 0; j < SIZE; j++)
        {
            for(int k = 0; k < SIZE; k++)
            {
                accumulator = 0;
                for(int l = 0, m = 0; l < SIZE && m < SIZE; l++, m++)
                    accumulator += matrix1[j][l] * matrix2[m][k];
                result_matrix[j][k] = accumulator;
            }
        }

        for(int j = 0; j < SIZE; j++)
        {
            for(int k = 0; k < SIZE; k++)
                fprintf(write_fp, "%d_", result_matrix[j][k]);
            fprintf(write_fp, "\n");
        }
        fclose(write_fp);
    }

    void *transpose(void *arg)
    {

```

```

Matrices *argv = arg;
int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE];
FILE *write_fp, *read_fp;

for(int i = 0; i < SIZE; i++)
    for(int j = 0; j < SIZE; j++)
    {
        matrix1[i][j] = argv->matrix1[i][j];
        matrix2[i][j] = argv->matrix2[i][j];
    }

printf("TRANSPOSE\n");
write_fp = fopen("Transpose.txt", "w");
for(int j = 0; j < SIZE; j++)
    for(int k = 0; k < SIZE; k++)
        matrix1[j][k] = matrix1[k][j];

for(int j = 0; j < SIZE; j++)
    for(int k = 0; k < SIZE; k++)
        matrix2[j][k] = matrix2[k][j];

for(int j = 0; j < SIZE; j++)
{
    for(int k = 0; k < SIZE; k++)
        fprintf(write_fp, "%d ", matrix1[j][k]);
    fprintf(write_fp, "\n");
}

fprintf(write_fp, "\n");
for(int j = 0; j < SIZE; j++)
{
    for(int k = 0; k < SIZE; k++)
        fprintf(write_fp, "%d ", matrix2[j][k]);
    fprintf(write_fp, "\n");
}
fclose(write_fp);
}

void *inverse(void *arg)
{
    Matrices *argv = arg;
    int det = 0;
    FILE *write_fp, *read_fp;
    int **matrix1, **matrix2;

    //Create the matrices
    matrix1 = malloc(SIZE * sizeof(int *));
    for(int i = 0; i < SIZE; i++)
        matrix1[i] = malloc(SIZE * sizeof(int));

    matrix2 = malloc(SIZE * sizeof(int *));
    for(int i = 0; i < SIZE; i++)
        matrix2[i] = malloc(SIZE * sizeof(int));

    for(int i = 0; i < SIZE; i++)
        for(int j = 0; j < SIZE; j++)
        {
            matrix1[i][j] = argv->matrix1[i][j];
            matrix2[i][j] = argv->matrix2[i][j];
        }

    printf("INVERSE\n");
    write_fp = fopen("Inverse.txt", "w");

    //Calculate the determinant of matrix1
    det = determinant(matrix1, SIZE);

    for(int j = 0; j < SIZE; j++)
    {
        for(int k = 0; k < SIZE; k++)

```



```

        fprintf(write_fp, "%d_", det * argv->matrix1[j][k]);
        fprintf(write_fp, "\n");
    }

    //Calculate the determinant of matrix2
    det = determinant(matrix2, SIZE);
    fprintf(write_fp, "\n");
    for(int j = 0; j < SIZE; j++)
    {
        for(int k = 0; k < SIZE; k++)
            fprintf(write_fp, "%d_", det * argv->matrix2[j][k]);
        fprintf(write_fp, "\n");
    }

    fclose(write_fp);
}

int determinant(int **matrix, int size)
{
    int det = 0;
    int **new_matrix;

    new_matrix = malloc(SIZE * sizeof(int *));
    for(int i = 0; i < size - 1; i++)
        new_matrix[i] = malloc((size - 1) * sizeof(int));

    if(size == 2)
        return matrix[0][0] * matrix[1][1] - matrix[1][0] * matrix[0][1];

    for(int k = 0; k < size; k++)
    {
        for(int i = 0; i < size - 1; i++)
            for(int j = 0; j < size - 1; j++)
                if(j != k)
                    new_matrix[i][j] = matrix[i + 1][j];

        for(int j = k; j < size - 1; j++)
            for(int i = 0; i < size - 1; i++)
                new_matrix[i][j] = new_matrix[i][j + 1];

        det += matrix[0][k] * determinant(new_matrix, size - 1);
    }

    return det;
}

```

2.3.5. Windows

2.3.6. Funcionamiento

Corremos el programa y enseguida veremos en pantalla el orden en el que fueron llamados los hilos:

```
SUM
SUBTRACTION
TRANSPOSE
MULTIPLICATION
INVERSE
THREAD 6
```

Figura 10:

Posteriormente la salida en pantalla correspondiente de cada operación: La suma y resta de matrices:

```

SUM
 9   7   9   9   10   3   10   10   10   10
 8  16  10  16  14  13  12  16  13  10
15  12   5   7  10   2  11  14  13  11
 7   8  10  11   5   8   1   2   7  11
13  17  13   9   9  10  11  14   1   4
 5  17  15   9  13   5   4   8   4   6
13   4  15  12  11  12   6   8  12   9
 9  10  10   9  15  11   5   7   2   8
 4  12  12   8  11   5  12   5   9   7
17  14  17  11   9  10  15  10   2  14

SUBTRACTION
-7   3   5  -5  -2  -3  -4   6   8  -8
-2   2  -2   0   4  -3   6  -2  -1  -4
 3  -6   3   5   2  -2  -7   2  -3  -1
 5  -4  -4   1   3   2   1  -2   5  -7
-3   1   5  -7   7  -4   5   0   1   4
 3  -1   1  -3   5  -1   0  -6   4  -4
-3   2   3  -2   1   0   2  -4   0  -7
 3   4   0  -1  -1   3   3  -7   2  -6
 4   2   2   0   3  -3  -2  -1  -9  -5
-1  -4  -1  -3   1   4   1   4   0   0
```

Figura 11:

La multiplicación y transpuestas:

```

MULTIPLICATION
166  225  181  200  184  180  181  225  194  245
298  357  337  382  253  306  209  355  217  389
250  255  228  278  211  198  208  221  130  317
145  215  174  198  125  137  155  153  112  223
328  321  257  326  248  265  231  336  202  333
225  278  181  241  154  205  195  245  176  231
215  293  211  246  180  191  214  244  182  284
221  272  207  264  162  201  172  244  142  255
225  229  169  225  147  187  154  224  144  229
336  347  302  350  270  249  251  322  175  394

TRANSPOSE OF MATRIX 1
 1   3   9   6   5   4   5   6   4   8
 3   9   3   2   9   8   3   7   7   5
 9   3   4   3   9   8   9   5   7   8
 6   2   3   6   1   3   5   4   4   4
 5   9   9   1   8   9   6   7   7   5
 4   8   8   3   9   2   6   7   1   7
 5   3   9   5   6   6   4   4   5   8
 6   7   5   4   7   7   4   0   2   7
 4   7   7   4   7   1   5   2   0   1
 8   5   8   4   5   7   8   7   1   7

TRANSPOSE OF MATRIX 2
 8   5   6   1   8   1   8   3   0   9
 5   7   9   6   8   9   1   3   5   9
 6   9   1   7   4   7   6   5   5   9
 1   6   7   5   8   6   7   5   4   7
```

Figura 12:

Las inversas:

INVERSE OF MATRIX 1									
1	5	7	2	4	0	3	8	9	1
3	9	4	8	9	5	9	7	6	3
9	3	4	6	6	0	2	8	5	5
6	2	3	6	4	5	1	0	6	2
5	9	9	1	8	3	8	7	1	4
4	8	8	3	9	2	2	1	4	1
5	3	9	5	6	6	4	2	6	1
6	7	5	4	7	7	4	0	2	1
4	7	7	4	7	1	5	2	0	1
8	5	8	4	5	7	8	7	1	7

INVERSE OF MATRIX 2									
56	14	14	49	42	21	49	14	7	63
35	49	42	56	35	56	21	63	49	49
42	63	7	7	28	14	63	42	56	42
7	42	49	35	7	21	0	14	7	63
56	56	28	56	7	49	21	49	0	0
7	63	49	42	28	21	14	49	0	35
56	7	42	49	35	42	14	42	42	56
21	21	35	35	56	28	7	49	0	49
0	35	35	28	28	28	49	21	63	42
63	63	63	49	28	21	49	21	7	49

Figura 13:

Los archivos creados por cada operación dentro del directorio de trabajo:

PC > Documents > ESCOM_SEMESTRE_5 > LCMY_SISTEMAS_OPERATIVOS > L_Unit > 5_Practice > Win				
Name	Date modified	Type	Size	
a.exe	4/20/2018 7:37 PM	Application	49 KB	
Inverse.txt	4/21/2018 11:30 PM	Text Document	1 KB	
MatrixThreadW.c	4/20/2018 7:37 PM	C File	10 KB	
Multiplication.txt	4/21/2018 11:30 PM	Text Document	1 KB	
Original.txt	4/21/2018 11:30 PM	Text Document	1 KB	
Subtraction.txt	4/21/2018 11:30 PM	Text Document	1 KB	
Sum.txt	4/21/2018 11:30 PM	Text Document	1 KB	
Transpose.txt	4/21/2018 11:30 PM	Text Document	1 KB	

Figura 14:

Y el tiempo de ejecución:

```
Execution time: 0.178000 seconds
```

Figura 15:

2.3.7. Observaciones

EL tiempo de ejecución de la aplicación en Windows es mucho mayor al de Linux, siendo este de aproximadamente 0.178000 segundos mientras que en Linux fue de aproximadamente 0.003162, esto demuestra claramente que en Linux la programación con hilos es mucho más rápida que en Windows.

2.3.8. Código

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>

#define SIZE 10

typedef unsigned char BYTE;

int determinant(int **matrix, int size);
DWORD WINAPI sum(LPVOID arg);
DWORD WINAPI subtraction(LPVOID arg);
DWORD WINAPI multiplication(LPVOID arg);
DWORD WINAPI transpose(LPVOID arg);
DWORD WINAPI inverse(LPVOID arg);

typedef struct
{
    int **matrix1;
    int **matrix2;
} Matrices;

int main(void)
{
    DWORD id_thread[5];
    HANDLE han_thread[5];
    BYTE i = 0, j = 0, k = 0, l = 0, m = 0;
    int **matrix1, **matrix2, result_matrix[SIZE][SIZE];
    clock_t begin, end;
    FILE *write_fp, *read_fp;

    Matrices argv;
    Matrices *argvp;

    //Create the matrices
    matrix1 = malloc(SIZE * sizeof(int *));
    for(i = 0; i < SIZE; i++)
        matrix1[i] = malloc(SIZE * sizeof(int));

    matrix2 = malloc(SIZE * sizeof(int *));
    for(i = 0; i < SIZE; i++)
        matrix2[i] = malloc(SIZE * sizeof(int));

    //Fill the matrices
    srand((unsigned) time(NULL));
    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
        {
            matrix1[i][j] = rand() % 10;
            matrix2[i][j] = rand() % 10;
        }

    //Copy the reference of the two matrices into the struct argv for further
    //argument to the threads
    argv.matrix1 = matrix1;
    argv.matrix2 = matrix2;

    //Copy the reference of the struct into argvp
    argvp = &argv;

    //Print the original two matrices
    write_fp = fopen("Original.txt", "w");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            fprintf(write_fp, "%d", matrix1[i][j]);
        fprintf(write_fp, "\n");
    }
```

```

fprintf(write_fp, "\n");
for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
        fprintf(write_fp, "%d", matrix2[i][j]);
    fprintf(write_fp, "\n");
}
fclose(write_fp);

begin = clock();
//Call 5 processes to run
for(i = 0; i < 5; i++)
{
    switch(i)
    {
        case 0:
            han_thread[0] = CreateThread(NULL, 0, sum, argv,
                                           0, &id_thread[0]);
            break;
        case 1:
            han_thread[1] = CreateThread(NULL, 0, subtraction,
                                           argv, 0, &id_thread[1]);
            break;
        case 2:
            han_thread[2] = CreateThread(NULL, 0,
                                           multiplication, argv, 0, &id_thread[2]);
            break;
        case 3:
            han_thread[3] = CreateThread(NULL, 0, transpose,
                                           argv, 0, &id_thread[3]);
            break;
        case 4:
            han_thread[4] = CreateThread(NULL, 0, inverse,
                                           argv, 0, &id_thread[4]);
            break;
    }
}
for(i = 0; i < 5; i++)
    WaitForSingleObject(han_thread[i], INFINITE);
printf("THREAD_6\n");

read_fp = fopen("Sum.txt", "rb");
for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        fscanf(read_fp, "%d", &result_matrix[i][j]);
fclose(read_fp);

printf("\t\t\t\tSUM\n");
for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
        printf("%5d", result_matrix[i][j]);
    printf("\n");
}

read_fp = fopen("Subtraction.txt", "rb");
for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        fscanf(read_fp, "%d", &result_matrix[i][j]);
fclose(read_fp);

printf("\n\t\t\t\tSUBTRACTION\n");
for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
        printf("%5d", result_matrix[i][j]);
    printf("\n");
}

read_fp = fopen("Multiplication.txt", "rb");

```

```

    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
            fscanf(read_fp, "%d", &result_matrix[i][j]);
    fclose(read_fp);

    printf("\n\t\t\t\tMULTIPLICATION\n");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            printf("%5d", result_matrix[i][j]);
        printf("\n");
    }

    read_fp = fopen("Transpose.txt", "rb");
    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
            fscanf(read_fp, "%d", &matrix1[i][j]);
    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
            fscanf(read_fp, "%d", &matrix2[i][j]);
    fclose(read_fp);

    printf("\n\t\t\t\tTRANSPPOSE_OF_MATRIX_1\n");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            printf("%5d", matrix1[i][j]);
        printf("\n");
    }
    printf("\n\t\t\t\tTRANSPPOSE_OF_MATRIX_2\n");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            printf("%5d", matrix2[i][j]);
        printf("\n");
    }

    read_fp = fopen("Inverse.txt", "rb");
    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
            fscanf(read_fp, "%d", &matrix1[i][j]);
    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
            fscanf(read_fp, "%d", &matrix2[i][j]);
    fclose(read_fp);

    printf("\n\t\t\t\tINVERSE_OF_MATRIX_1\n");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            printf("%5d", matrix1[i][j]);
        printf("\n");
    }
    printf("\n\t\t\t\tINVERSE_OF_MATRIX_2\n");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            printf("%5d", matrix2[i][j]);
        printf("\n");
    }
    end = clock();

    printf("\nExecution_time:_%f_seconds\n", (double) (end - begin) /
        CLOCKS_PER_SEC);

    //Finally terminate main process
    return 0;
}
DWORD WINAPI sum(LPVOID arg)

```

```

{
    Matrices *argv = arg;
    FILE *write_fp, *read_fp;
    int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE];

    for(int i = 0; i < SIZE; i++)
        for(int j = 0; j < SIZE; j++)
        {
            matrix1[i][j] = argv->matrix1[i][j];
            matrix2[i][j] = argv->matrix2[i][j];
        }

    printf("SUM\n");
    write_fp = fopen("Sum.txt", "w");
    for(int j = 0; j < SIZE; j++)
        for(int k = 0; k < SIZE; k++)
            matrix1[j][k] = matrix1[j][k] + matrix2[j][k];

    for(int j = 0; j < SIZE; j++)
    {
        for(int k = 0; k < SIZE; k++)
            fprintf(write_fp, "%d_", matrix1[j][k]);
        fprintf(write_fp, "\n");
    }
    fclose(write_fp);
}

DWORD WINAPI subtraction(LPVOID arg)
{
    Matrices *argv = arg;
    int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE];
    FILE *write_fp, *read_fp;

    for(int i = 0; i < SIZE; i++)
        for(int j = 0; j < SIZE; j++)
        {
            matrix1[i][j] = argv->matrix1[i][j];
            matrix2[i][j] = argv->matrix2[i][j];
        }

    printf("SUBTRACTION\n");
    write_fp = fopen("Subtraction.txt", "w");
    for(int j = 0; j < SIZE; j++)
        for(int k = 0; k < SIZE; k++)
            matrix1[j][k] = matrix1[j][k] - matrix2[j][k];

    for(int j = 0; j < SIZE; j++)
    {
        for(int k = 0; k < SIZE; k++)
            fprintf(write_fp, "%d_", matrix1[j][k]);
        fprintf(write_fp, "\n");
    }
    fclose(write_fp);
}

DWORD WINAPI multiplication(LPVOID arg)
{
    Matrices *argv = arg;
    int accumulator = 0, result_matrix[SIZE][SIZE];
    int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE];
    FILE *write_fp, *read_fp;

    for(int i = 0; i < SIZE; i++)
        for(int j = 0; j < SIZE; j++)
        {
            matrix1[i][j] = argv->matrix1[i][j];
            matrix2[i][j] = argv->matrix2[i][j];
        }

    printf("MULTIPLICATION\n");

```

```

write_fp = fopen("Multiplication.txt", "w");
for(int j = 0; j < SIZE; j++)
{
    for(int k = 0; k < SIZE; k++)
    {
        accumulator = 0;
        for(int l = 0, m = 0; l < SIZE && m < SIZE; l++, m++)
            accumulator += matrix1[j][l] * matrix2[m][k];
        result_matrix[j][k] = accumulator;
    }
}

for(int j = 0; j < SIZE; j++)
{
    for(int k = 0; k < SIZE; k++)
        fprintf(write_fp, "%d", result_matrix[j][k]);
    fprintf(write_fp, "\n");
}
fclose(write_fp);
}

DWORD WINAPI transpose(LPVOID arg)
{
    Matrices *argv = arg;
    int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE];
    FILE *write_fp, *read_fp;

    for(int i = 0; i < SIZE; i++)
        for(int j = 0; j < SIZE; j++)
        {
            matrix1[i][j] = argv->matrix1[i][j];
            matrix2[i][j] = argv->matrix2[i][j];
        }

    printf("TRANSPPOSE\n");
    write_fp = fopen("Transpose.txt", "w");
    for(int j = 0; j < SIZE; j++)
        for(int k = 0; k < SIZE; k++)
            matrix1[j][k] = matrix1[k][j];

    for(int j = 0; j < SIZE; j++)
        for(int k = 0; k < SIZE; k++)
            matrix2[j][k] = matrix2[k][j];

    for(int j = 0; j < SIZE; j++)
    {
        for(int k = 0; k < SIZE; k++)
            fprintf(write_fp, "%d", matrix1[j][k]);
        fprintf(write_fp, "\n");
    }

    fprintf(write_fp, "\n");
    for(int j = 0; j < SIZE; j++)
    {
        for(int k = 0; k < SIZE; k++)
            fprintf(write_fp, "%d", matrix2[j][k]);
        fprintf(write_fp, "\n");
    }
    fclose(write_fp);
}

DWORD WINAPI inverse(LPVOID arg)
{
    Matrices *argv = arg;
    int det = 0;
    FILE *write_fp, *read_fp;
    int **matrix1, **matrix2;

    //Create the matrices
    matrix1 = malloc(SIZE * sizeof(int *));

```



```

    for(int i = 0; i < SIZE; i++)
        matrix1[i] = malloc(SIZE * sizeof(int));

    matrix2 = malloc(SIZE * sizeof(int *));
    for(int i = 0; i < SIZE; i++)
        matrix2[i] = malloc(SIZE * sizeof(int));

    for(int i = 0; i < SIZE; i++)
        for(int j = 0; j < SIZE; j++)
        {
            matrix1[i][j] = argv->matrix1[i][j];
            matrix2[i][j] = argv->matrix2[i][j];
        }

    printf("INVERSE\n");
    write_fp = fopen("Inverse.txt", "w");

    //Calculate the determinant of matrix1
    det = determinant(matrix1, SIZE);

    for(int j = 0; j < SIZE; j++)
    {
        for(int k = 0; k < SIZE; k++)
            fprintf(write_fp, "%d", det * argv->matrix1[j][k]);
        fprintf(write_fp, "\n");
    }

    //Calculate the determinant of matrix2
    det = determinant(matrix2, SIZE);
    fprintf(write_fp, "\n");
    for(int j = 0; j < SIZE; j++)
    {
        for(int k = 0; k < SIZE; k++)
            fprintf(write_fp, "%d", det * argv->matrix2[j][k]);
        fprintf(write_fp, "\n");
    }

    fclose(write_fp);
}

int determinant(int **matrix, int size)
{
    int det = 0;
    int **new_matrix;

    new_matrix = malloc(SIZE * sizeof(int *));
    for(int i = 0; i < size - 1; i++)
        new_matrix[i] = malloc((size - 1) * sizeof(int));

    if(size == 2)
        return matrix[0][0] * matrix[1][1] - matrix[1][0] * matrix[0][1];

    return det = rand() % 10;
    for(int k = 0; k < size; k++)
    {
        for(int i = 0; i < size - 1; i++)
            for(int j = 0; j < size - 1; j++)
                if(j != k)
                    new_matrix[i][j] = matrix[i + 1][j];

        for(int j = k; j < size - 1; j++)
            for(int i = 0; i < size - 1; i++)
                new_matrix[i][j] = new_matrix[i][j + 1];

        det += matrix[0][k] * determinant(new_matrix, size - 1);
    }
}

```

2.4. Punto 7

Programe una aplicación (tanto en Linux como en Windows) que copie los archivos y directorios contenidos dentro de una ruta específica. Por cada directorio que se encuentre al momento de copiar, se deberá de crear un hilo que se encargará de copiar los archivos existentes en ese directorio. Nuevamente, si se encuentra otro directorio se creará otro hilo, así sucesivamente. Todos los hilos deberán de correr concurrentemente. Las rutas de origen y destino de copia se aceptarán por línea de comando.

2.4.1. Linux

2.4.2. Funcionamiento

A la hora de correr el programa le pasamos como argumentos por línea de comando a la aplicación la ruta de origen y destino respectivamente:

```
james@dragmaili:~/Documents/ESCOM_SEMESTRE_5/2CM9_SISTEMAS_OPERATIVOS/2_Unit/5_Practice/7_Point$ ./a.out /home/james/Documents/ESCOM_SEMESTRE_5/2CM9_SISTEMAS_OPERATIVOS/2_Unit/5_Practice/7_Point/ADirectory /home/james/Documents/ESCOM_SEMESTRE_5/2CM9_SISTEMAS_OPERATIVOS/2_Unit/5_Practice/7_Point/BDirectory
```

Figura 16:

Después de eso veremos la lista de entradas que la aplicación encontró dentro del directorio origen y que copió dentro del directorio destino:

```
Directory1
File3.txt
File1.txt
File2.txt
```

Figura 17:

Los archivos y directorios dentro del directorio origen *ADirectory*:

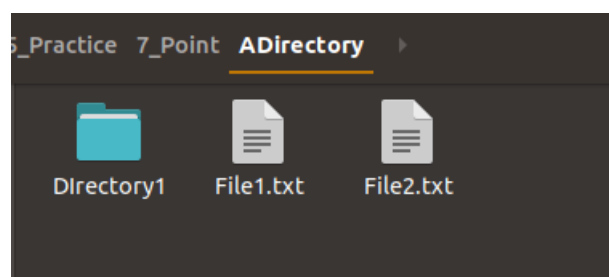


Figura 18:

Comprobamos que los archivos y directorios dentro del directorio origen están en el nuevo directorio *BDirectory*:

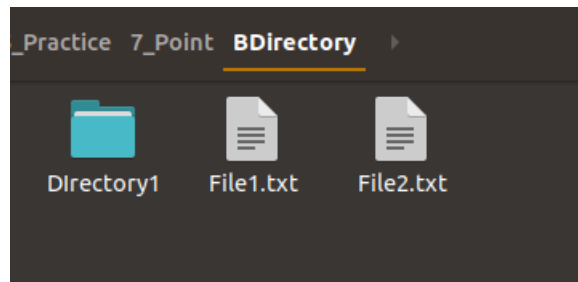


Figura 19:

2.4.3. Observaciones

En este caso la programación con hilos resultó muy cómoda, ya que solo se tuvo que programar una función recursiva que se encarga de hacer todo el proceso, claro cada vez que se llama se crea un hilo nuevo y esto permite una rapidez muy buena a la hora de necesitar un buen rendimiento.

2.4.4. Código

```
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <pthread.h>

void *copyDir(void *arg);

typedef struct
{
    char source_path[FILENAME_MAX];
    char dest_path[FILENAME_MAX];
} Paths;

int main(int argc, char *argv[])
{
    pthread_t id_thread;
    Paths paths;
    Paths *arg;

    printf("\n");
    //Test if the call of the program was done right
    if(argc != 3)
    {
        printf("Usage: ~ program_name ~ source_directory ~\n");
        exit(EXIT_FAILURE);
    }

    //Save the paths of the source and destination paths into the structure
    strcpy(paths.source_path, argv[1]);
    strcpy(paths.dest_path, argv[2]);
    //Make a pointer to the structure
    arg = &paths;

    //Call the first thread with the pointer as argument to the thread that
    will use the paths
```

```

        pthread_create(&id_thread, NULL, copyDir, arg);
        pthread_join(id_thread, NULL);

        return 0;
    }

    void *copyDir(void *arg)
    {
        Paths *dir_path = arg;
        char source_path[FILENAME_MAX], dest_path[FILENAME_MAX];
        struct dirent **namelist;
        struct stat stat_buf;
        pthread_t id_thread;
        int n, i, j, fd, chunk_size, fd1;
        char buffer[100], dest_path_fname[FILENAME_MAX];
        Paths paths;
        Paths *argv;

        //Copy the source and dest path into the new source and dest paths
        strcpy(source_path, dir_path->source_path);
        strcpy(dest_path, dir_path->dest_path);

        //Scan the source directory and check if it could be scanned
        n = scandir(source_path, &namelist, NULL, alphasort);
        if(n == -1)
        {
            printf("Error: _cannot_scan_source_directory\n");
            exit(EXIT_FAILURE);
        }

        //Start at 2 because the first two entries are "." and ".." always
        for(j = 2; j < n; j++)
        {
            //Change to source directory to be able to use relative pathnames
            if((chdir(source_path)) == -1)
            {
                printf("Error: _cannot_open_source_directory\n");
                exit(EXIT_FAILURE);
            }
            printf("%s\n", namelist[j]->d_name);
            if((stat(namelist[j]->d_name, &stat_buf)) == -1)
            {
                printf("Error: _cannot_check_\"%s\" _file_status\n", namelist[j]->d_name);
                exit(EXIT_FAILURE);
            }

            //Check whether the entry is a regular file or directory
            if(S_ISREG(stat_buf.st_mode))
            {
                //printf("-Regular file\n");
                //Open the file to read from
                if((fd = open(namelist[j]->d_name, O_RDONLY)) == -1)
                {
                    printf("Error: _cannot_open_\"%s\" \n", namelist[j]->d_name);
                    exit(EXIT_FAILURE);
                }
                //Clear the dest_path_fname and fill it with the pathname
                //of the new file to create
                for(i = 0; i < FILENAME_MAX; i++)
                    dest_path_fname[i] = '\0';
                strcpy(dest_path_fname, dest_path);
                strcat(dest_path_fname, "/" );
                strcat(dest_path_fname, namelist[j]->d_name);

                //Create the file to write to
                if((fd1 = creat(dest_path_fname, S_IRWXU)) == -1)
                {

```

```

        printf("Error:_cannot_create_\\" %s\\"\\n",
               dest_path_fname);
        exit(EXIT_FAILURE);
    }
    //Read chunks of 100 bytes from the file and write them to
    //the file destination until there's no longer more bytes
    //to read
    while((chunk_size = read(fd, buffer, 100)) > 0)
    {
        /*for(i = 0; i < chunk_size; i++)
           printf("%c", buffer[i]);
        printf("\\n");*/
        write(fd1, buffer, chunk_size);
    }
    close(fd1);
    close(fd);
}
else if(S_ISDIR(stat_buf.st_mode))
{
    //printf("-Directory\\n");
    //Clear the dest_path_fname and fill it with the path name
    //of the new directory to create
    for(i = 0; i < FILENAME_MAX; i++)
        dest_path_fname[i] = '\\0';
    strcpy(dest_path_fname, dest_path);
    strcat(dest_path_fname, "/");
    strcat(dest_path_fname, namelist[j]->d_name);

    if((mkdir(dest_path_fname, S_IRWXU)) == -1)
    {
        printf("Error:_cannot_create_directory_\\" %s\\"\\n",
               dest_path_fname);
        exit(EXIT_FAILURE);
    }
    //Save the new destination path name into the member of the
    //structure
    strcpy(paths.dest_path, dest_path_fname);
    //Clear the dest_path_fname and fill it with the path name
    //of the new source path name
    for(i = 0; i < FILENAME_MAX; i++)
        dest_path_fname[i] = '\\0';
    strcpy(dest_path_fname, source_path);
    strcat(dest_path_fname, "/");
    strcat(dest_path_fname, namelist[j]->d_name);
    //Save the paths of the source and destination paths into
    //the structure
    strcpy(paths.source_path, dest_path_fname);

    //Make a pointer to the structure
    argv = &paths;
    //Call the next thread
    pthread_create(&id_thread, NULL, copyDir, argv);
    //pthread_join(id_thread, NULL);
}
}
return NULL;
}

```

2.4.5. Windows

2.4.6. Funcionamiento

A la hora de correr el programa le pasamos como argumentos por línea de comando a la aplicación la ruta de origen y destino respectivamente:

```
C:\Users\James\Documents\ESCOM_SEMESTRE_5\2CM9_SISTEMAS_OPERATIVOS\2_Unit\5_Practice\Windows\7_Point>a C:/Users/James/Do
cuments/ESCOM_SEMESTRE_5/2CM9_SISTEMAS_OPERATIVOS/2_Unit/5_Practice/Windows/7_Point/ADirectory C:/Users/James/Documents/
ESCOM_SEMESTRE_5/2CM9_SISTEMAS_OPERATIVOS/2_Unit/5_Practice/Windows/7_Point/BDirectory
```

Figura 20:

Después de eso veremos la lista de entradas que la aplicación encontró dentro del directorio origen y que copió dentro del directorio destino:

```
Directory1
File1.txt
File3.txt
File2.txt
```

Figura 21:

Los archivos y directorios dentro del directorio origen *ADirectory*:

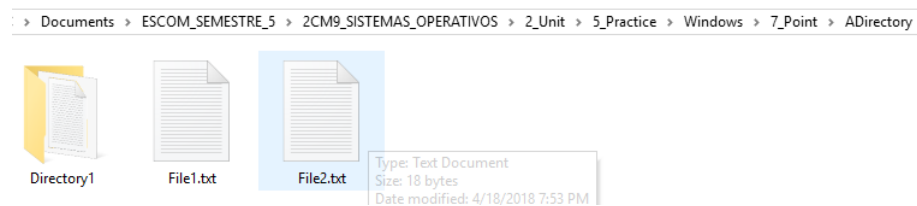


Figura 22:

Comprobamos que los archivos y directorios dentro del directorio origen están en el nuevo directorio *BDirectory*:

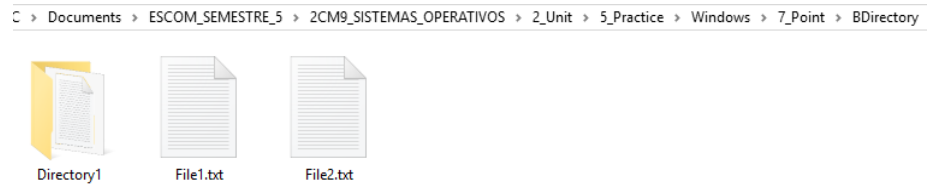


Figura 23:

2.4.7. Observaciones

Programar la aplicación en Windows requirió utilizar funciones de prácticas previamente vistas dado que, al no poderse utilizar la función *scandir()* en este sistema operativo, se tuvo que buscar otra función en este caso dos; *opendir()* y *readdir()* que ayudan a cumplir la función de *scandir()*.

2.4.8. Código

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <Windows.h>
#include <sys/stat.h>
#include <dirent.h>
#include <stdbool.h>

DWORD WINAPI copyDir(LPVOID arg);

typedef struct
{
    char source_path[FILENAME_MAX];
    char dest_path[FILENAME_MAX];
} Paths;

int main(int argc, char *argv[])
{
    DWORD id_thread;
    HANDLE han_thread;
    Paths paths;
    Paths *arg;

    printf("\n");
    //Test if the call of the program was done right
    if(argc != 3)
    {
        printf("Usage: _ program_name _ source_directory _\n");
        exit(EXIT_FAILURE);
    }

    //Save the paths of the source and destination paths into the structure
    strcpy(paths.source_path, argv[1]);
    strcpy(paths.dest_path, argv[2]);
    //Make a pointer to the structure
    arg = &paths;

    //Call the first thread with the pointer as argument to the thread that
    //will use the paths
    han_thread = CreateThread(NULL, 0, copyDir, arg, 0, &id_thread);
    WaitForSingleObject(han_thread, INFINITE);
}
```

```

        return 0;
    }
}

DWORD WINAPI copyDir(LPVOID arg)
{
    Paths *dir_path = arg;
    char source_path[FILENAME_MAX], dest_path[FILENAME_MAX];
    DWORD id_thread;
    HANDLE han_thread;
    int n, i, j;
    long unsigned int chunk_size;
    char buffer[100], dest_path_fname[FILENAME_MAX];
    Paths paths;
    Paths *argv;
    DIR *directory_stream;
    struct dirent *file_pointer;
    HANDLE fd, fd1;

    //Copy the source and dest path into the new source and dest paths
    strcpy(source_path, dir_path->source_path);
    strcpy(dest_path, dir_path->dest_path);

    //Open the source directory
    if((directory_stream = opendir(source_path)) == NULL)
    {
        printf("Error: _cannot_open_the_directory.\n");
        exit(EXIT_FAILURE);
    }

    //To avoid checking ".." entry
    file_pointer = readdir(directory_stream);
    //To avoid "." entry
    file_pointer = readdir(directory_stream);

    //Start at 2 because the first two entries are "." and ".." always
    while((file_pointer = readdir(directory_stream)) != NULL)
    {
        //Change to source directory to be able to use relative pathnames
        if((SetCurrentDirectory(source_path)) == -1)
        {
            printf("Error: _cannot_open_source_directory\n");
            exit(EXIT_FAILURE);
        }
        printf("%s\n", file_pointer->d_name);

        //Check whether the entry is a regular file or directory
        if(file_pointer->d_type == 0)
        {
            //printf("- Regular file\n");
            //Open the file to read from
            if((fd = CreateFile(file_pointer->d_name, GENERIC_READ,
                FILE_SHARE_READ, NULL, OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL, NULL)) == INVALID_HANDLE_VALUE)
            {
                printf("Error: _cannot_open_\"%s\".\n", file_pointer->d_name);
                exit(EXIT_FAILURE);
            }
            //Clear the dest_path_fname and fill it with the pathname
            //of the new file to create
            for(i = 0; i < FILENAME_MAX; i++)
                dest_path_fname[i] = '\0';
            strcpy(dest_path_fname, dest_path);
            strcat(dest_path_fname, "/");
            strcat(dest_path_fname, file_pointer->d_name);

            //Create the file to write to
            if((fd1 = CreateFile(dest_path_fname, GENERIC_READ |
                GENERIC_WRITE, FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
                FILE_ATTRIBUTE_NORMAL, NULL)) == INVALID_HANDLE_VALUE)

```



```

        {
            printf("Error:_cannot_create_\\" %s\\"\\n",
                dest_path_fname);
            exit(EXIT_FAILURE);
        }
        //Read chunks of 100 bytes from the file and write them to
        the file destination until there's no longer more bytes
        to read
        while(1)
        {
            /*for(i = 0; i < chunk_size; i++)
                printf("%c", buffer[i]);
            printf("\\n");*/
            ReadFile(fd, buffer, 100, &chunk_size, NULL);
            if(chunk_size == 0)
                break;
            WriteFile(fd1, buffer, chunk_size, NULL, NULL);
        }
        CloseHandle(fd1);
        CloseHandle(fd);
    }
    else if(file_pointer->d_type == 16)
    {
        //printf("-Directory\\n");
        //Clear the dest_path_fname and fill it with the path name
        of the new directory to create
        for(i = 0; i < FILENAME_MAX; i++)
            dest_path_fname[i] = '\\0';
        strcpy(dest_path_fname, dest_path);
        strcat(dest_path_fname, "/" );
        strcat(dest_path_fname, file_pointer->d_name);

        if((CreateDirectory(dest_path_fname, NULL)) == 0)
        {
            printf("Error:_cannot_create_directory_\\" %s\\"\\n",
                dest_path_fname);
            exit(EXIT_FAILURE);
        }
        //Save the new destination path name into the member of the
        structure
        strcpy(paths.dest_path, dest_path_fname);
        //Clear the dest_path_fname and fill it with the path name
        of the new source path name
        for(i = 0; i < FILENAME_MAX; i++)
            dest_path_fname[i] = '\\0';
        strcpy(dest_path_fname, source_path);
        strcat(dest_path_fname, "/" );
        strcat(dest_path_fname, file_pointer->d_name);
        //Save the paths of the source and destination paths into
        the structure
        strcpy(paths.source_path, dest_path_fname);

        //Make a pointer to the structure
        argv = &paths;
        //Call the next thread
        han_thread = CreateThread(NULL, 0, copyDir, argv, 0, &
            id_thread);
        //WaitForSingleObject(han_thread, INFINITE);
    }
}

return 0;
}

```

3. Análisis Crítico

La práctica fue más ligera que las anteriores y al tener más tiempo para realizarla se pudo hacer un código más optimizado y claro, además de que muchas partes de código ya estaban hechas solo había que hacer la implementación de los hilos aunque tampoco fue una tarea trivial.

4. Conclusión

La programación con hilos resulta muy eficiente al igual que con los procesos normales, solo que estos ocupan menos recursos del equipo de cómputo y realizan las mismas tareas en un tiempo muy aproximado al de la programación con procesos, aunque requiere de un nuevo tipo de lógica y de pequeños errores que hay que cuidar no cometer.