PRÁCTICA 6: COMUNICACIÓN INTER PROCESOS (IPC) EN LINUX Y WINDOWS

ALUMNO: BASTIDA PRADO JAIME ARMANDO

PROFESOR: CORTÉS GALICIA JORGE

MATERIA: SISTEMAS OPERATIVOS

GRUPO: 2CM9

Mayo 2018

Índice

1. Competencias

El alumno comprende el funcionamiento de las tuberías (pipes) sin nombre y de la memoria compartida como mecanismos de comunicación entre procesos tanto en el sistema operativo Linux como Windows para el desarrollo de aplicaciones concurrentes con soporte de comunicación.

2. Desarrollo

2.1. Punto 1

A través de la ayuda en línea que proporciona Linux, investigue el funcionamiento de las funciones: **pipe()**, **shmget()**, **shmat()**. Explique los argumentos y retorno de cada función.

- **int pipe(int pipefd[2]);**

  El argumento que se le pasa a la función es un arreglo de dos enteros, el primero para el descriptor de lectura de la pipa y otro para escritura.

  Si la llamada a la función es exitosa regresa 0, si hay error -1.

- **int pthread_join(pthread_t thread, void **retva l)**

  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 4

Programe una aplicación que cree un proceso hijo a partir de un proceso padre, el proceso padre enviará al proceso hijo, a través de una tubería, dos matrices de 10x10 a multiplicar por parte del hijo, mientras tanto el proceso hijo creará un hijo de él, al cual enviará dos matrices de 10x10 a sumar en el procso hijo creado, nuevamente el envío de estos valores será a través de una tubería. Una vez calculado el resultado de la suma, el proceso hijo del hijo devolverá la matriz resultante a su abuelo (vía tubería). A su vez, el proceso hijo devolverá la matriz resultante de la multiplicación que realizó a su padre. Finalmente, el proceso padre obtendrá la matriz inversa de cada una de las matrices recibidas y el resultado lo guardará en un archivo para cada matriz inversa obtenida. Programe esta aplicación tanto para Linux como para Windows utilizando las tuberías de cada sistema operativo.

2.2.1. Linux

2.2.2. Funcionamiento

Al correr el programa este nos indicará en pantalla cuando este enviando el resultado de cada maatriz respectiva a un archivo:

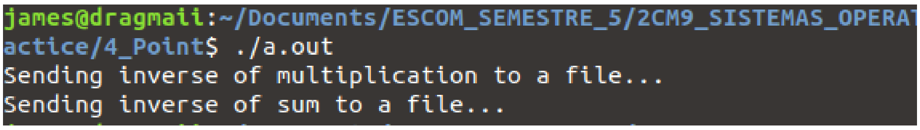


Figura 1:

Posteriormente podremos ver los archivos dentro del directorio de trabajo:

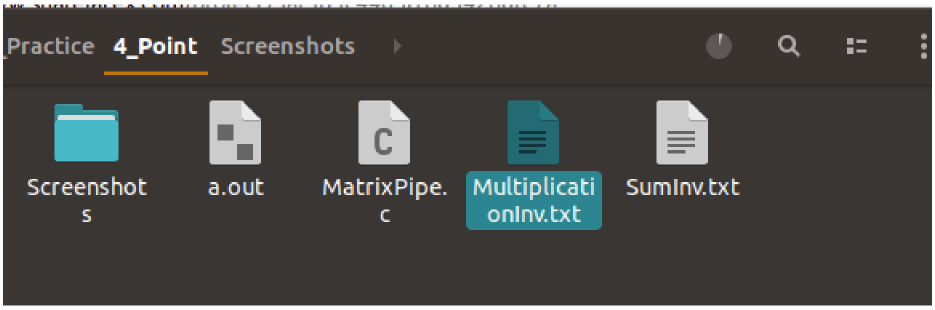
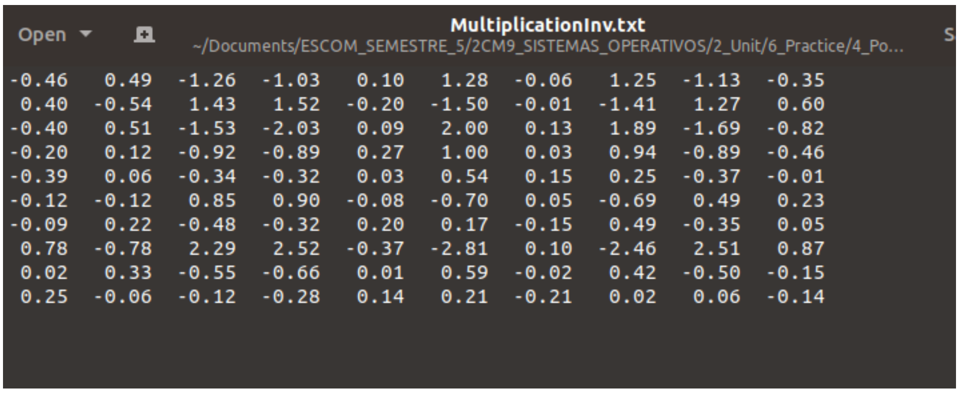


Figura 2:

Y su contenido:



-0.46	0.49	-1.26	-1.03	0.10	1.28	-0.06	1.25	-1.13	-0.35
0.40	-0.54	1.43	1.52	-0.20	-1.50	-0.01	-1.41	1.27	0.60
-0.40	0.51	-1.53	-2.03	0.09	2.00	0.13	1.89	-1.69	-0.82
-0.20	0.12	-0.92	-0.89	0.27	1.00	0.03	0.94	-0.89	-0.46
-0.39	0.06	-0.34	-0.32	0.03	0.54	0.15	0.25	-0.37	-0.01
-0.12	-0.12	0.85	0.90	-0.08	-0.70	0.05	-0.69	0.49	0.23
-0.09	0.22	-0.48	-0.32	0.20	0.17	-0.15	0.49	-0.35	0.05
0.78	-0.78	2.29	2.52	-0.37	-2.81	0.10	-2.46	2.51	0.87
0.02	0.33	-0.55	-0.66	0.01	0.59	-0.02	0.42	-0.50	-0.15
0.25	-0.06	-0.12	-0.28	0.14	0.21	-0.21	0.02	0.06	-0.14

Figura 3:

2.2.3. Código

```
//Program that uses process inter communication to process two matrices
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <stdbool.h>

#define SIZE 10
#define BUF_SIZE 1000

bool inverse(int A[SIZE][SIZE], float inverse[SIZE][SIZE]);
void adjoint(int A[SIZE][SIZE], int adj[SIZE][SIZE]);
void getCofactor(int A[SIZE][SIZE], int temp[SIZE][SIZE], int p, int q, int n);
int determinant(int A[SIZE][SIZE], int n);

int main(void)
{
        int pipefd[2], i, j, offset;
        char *wbuffer, *rbuffer;
        int **matrix1, **matrix2, result_matrix[SIZE][SIZE];
        float inv[SIZE][SIZE];
        FILE *write_fp;

        srand((unsigned) time(NULL));

        //Asign memory space for both matrices and buffers
        matrix1 = malloc(SIZE * sizeof(int *));
        matrix2 = malloc(SIZE * sizeof(int *));
        wbuffer = malloc(BUF_SIZE);
        rbuffer = malloc(BUF_SIZE);
        for(i = 0; i < SIZE; i++)
        {
                matrix1[i] = malloc(SIZE * sizeof(int));
                matrix2[i] = malloc(SIZE * sizeof(int));
        }

        //Fill the matrices with random values
        for(i = 0; i < SIZE; i++)
                for(j = 0; j < SIZE; j++)
                {
                        matrix1[i][j] = rand() %5;
                        matrix2[i][j] = rand() %5;
                }

        //Create the first pipe
        if(pipe(pipefd) == -1)
        {
                perror("pipe");
                exit(EXIT_FAILURE);
        }

        //Create child process
        if(fork() == 0)
        {
                int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE], result_matrix[SIZE][
                    SIZE], i, j, offset = 0, pipefd1[2], accumulator;
                char *wbuffer, *rbuffer;

                //Allocate memory for buffers
                wbuffer = malloc(BUF_SIZE);
                rbuffer = malloc(BUF_SIZE);

                //Read matrix1 and matrix2 from first pipe
                read(pipefd[0], rbuffer, BUF_SIZE);
```

```
        //Read into matrix1 from rbuffer
        for(i = 0; i < SIZE; i++)
                for(j = 0; j < SIZE; j++)
                {
                        sscanf(rbuffer, " %d%n", &matrix1[i][j], &offset);
                        rbuffer += offset;
                }
        //Read into matrix2 from rbuffer
        for(i = 0; i < SIZE; i++)
                for(j = 0; j < SIZE; j++)
                {
                        sscanf(rbuffer, " %d%n", &matrix2[i][j], &offset);
                        rbuffer += offset;
                }
        //Create the second pipe
        if(pipe(pipefd1) == -1)
        {
                perror("pipe");
                exit(EXIT_FAILURE);
        }
        //Create child process
        if(fork() == 0)
        {
                //Close unused write end
                close(pipefd1[1]);

                int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE], result_matrix
                    [SIZE][SIZE], i, j, offset = 0;
                char *wbuffer, *rbuffer;

                //Allocate memory for buffers
                wbuffer = malloc(BUF_SIZE);
                rbuffer = malloc(BUF_SIZE);

                //Read matrix1 and matrix2 from pipe
                read(pipefd1[0], rbuffer, BUF_SIZE);

                //Read into matrix1 from rbuffer
                for(i = 0; i < SIZE; i++)
                        for(j = 0; j < SIZE; j++)
                        {
                                sscanf(rbuffer, " %d%n", &matrix1[i][j], &
                                    offset);
                                rbuffer += offset;
                        }
                //Read into matrix2 from rbuffer
                for(i = 0; i < SIZE; i++)
                        for(j = 0; j < SIZE; j++)
                        {
                                sscanf(rbuffer, " %d%n", &matrix2[i][j], &
                                    offset);
                                rbuffer += offset;
                        }
                //Do sum
                for(i = 0; i < SIZE; i++)
                        for(j = 0; j < SIZE; j++)
                                result_matrix[i][j] = matrix1[i][j] +
                                    matrix2[i][j];

                //Write the resultx of sum into wbuffer
                for(i = 0; i < SIZE; i++)
                {
                        for(j = 0; j < SIZE; j++)
                                sprintf(wbuffer + strlen(wbuffer), "%d ",
                                    result_matrix[i][j]);
                        sprintf(wbuffer + strlen(wbuffer), "\n");
                }
                //Write to first pipe the result of sum
                write(pipefd[1], wbuffer, strlen(wbuffer));
```

```
                        exit(EXIT_SUCCESS);
                }
                else
                {
                        //Write the matrix1 into wbuffer
                        for(i = 0; i < SIZE; i++)
                        {
                                for(j = 0; j < SIZE; j++)
                                        sprintf(wbuffer + strlen(wbuffer), "%d ",
                                            matrix1[i][j]);
                                sprintf(wbuffer + strlen(wbuffer), "\n");
                        }
                        //Write the matrix2 into wbuffer
                        for(i = 0; i < SIZE; i++)
                        {
                                for(j = 0; j < SIZE; j++)
                                        sprintf(wbuffer + strlen(wbuffer), "%d ",
                                            matrix2[i][j]);
                                sprintf(wbuffer + strlen(wbuffer), "\n");
                        }
                        //Write matrix1 and matrix2 to the second pipe
                        write(pipefd1[1], wbuffer, strlen(wbuffer));

                        //Do multiplication
                        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;
                                }
                        }
                        //Clean wbuffer
                        for(i = 0; i < BUF_SIZE; i++)
                                wbuffer[i] = '\0';
                        //Write multiplication into wbuffer
                        for(i = 0; i < SIZE; i++)
                        {
                                for(j = 0; j < SIZE; j++)
                                        sprintf(wbuffer + strlen(wbuffer), "%d ",
                                            result_matrix[i][j]);
                                sprintf(wbuffer + strlen(wbuffer), "\n");
                        }
                        //Write to first pipe the result of multiplication
                        write(pipefd[1], wbuffer, strlen(wbuffer));
                }
                wait(0);
                exit(EXIT_SUCCESS);
        }
        //Father
        else
        {
                //Write the matrix1 into wbuffer
                for(i = 0; i < SIZE; i++)
                {
                        for(j = 0; j < SIZE; j++)
                                sprintf(wbuffer + strlen(wbuffer), "%d ", matrix1[i
                                    ][j]);
                        sprintf(wbuffer + strlen(wbuffer), "\n");
                }
                //Write the matrix2 into wbuffer
                for(i = 0; i < SIZE; i++)
                {
                        for(j = 0; j < SIZE; j++)
                                sprintf(wbuffer + strlen(wbuffer), "%d ", matrix2[i
                                    ][j]);
```

```
                        sprintf(wbuffer + strlen(wbuffer), "\n");
                }
                //Write matrix1 and matrix2 to the first pipe
                write(pipefd[1], wbuffer, strlen(wbuffer));

                //Read from first pipe result of multiplication and sum
                //wait(0);
                read(pipefd[0], rbuffer, BUF_SIZE);

                //Read into result_matrix the result of multiplication from rbuffer
                offset = 0;
                for(i = 0; i < SIZE; i++)
                        for(j = 0; j < SIZE; j++)
                        {
                                sscanf(rbuffer, " %d%n", &result_matrix[i][j], &
                                    offset);
                                rbuffer += offset;
                        }
                //Calculate inverse of multiplication and send it to a file
                write_fp = fopen("MultiplicationInv.txt", "w");
                printf("Sending inverse of multiplication to a file...\n");
                if(inverse(result_matrix, inv))
                {
                        for(i = 0; i < SIZE; i++)
                        {
                                for(j = 0; j < SIZE; j++)
                                        fprintf(write_fp, "%6.2f ", inv[i][j]);
                                fprintf(write_fp, "\n");
                        }
                }
                fclose(write_fp);

                //Read into result_matrix the result of sum from rbuffer
                for(i = 0; i < SIZE; i++)
                        for(j = 0; j < SIZE; j++)
                        {
                                sscanf(rbuffer, " %d%n", &result_matrix[i][j], &
                                    offset);
                                rbuffer += offset;
                        }
                //Calculate inverse of sum and send it to a file
                write_fp = fopen("SumInv.txt", "w");
                printf("Sending inverse of sum to a file...\n");
                if(inverse(result_matrix, inv))
                {
                        for(i = 0; i < SIZE; i++)
                        {
                                for(j = 0; j < SIZE; j++)
                                        fprintf(write_fp, "%6.2f ", inv[i][j]);
                                fprintf(write_fp, "\n");
                        }
                }
                fclose(write_fp);
        }

        return 0;
}

// Function to calculate and store inverse, returns false if
// matrix is singular
bool inverse(int A[SIZE][SIZE], float inverse[SIZE][SIZE])
{
    // Find determinant of A[][]
    int det = determinant(A, SIZE);
    if (det == 0)
    {
        printf("Singular matrix, can't find its inverse\n");
        return false;
    }
```

```
    // Find adjoint
    int adj[SIZE][SIZE];
    adjoint(A, adj);

    // Find Inverse using formula "inverse(A) = adj(A)/det(A)"
    for (int i=0; i<SIZE; i++)
        for (int j=0; j<SIZE; j++)
            inverse[i][j] = adj[i][j]/(float) det;

    return true;
}

// Function to get adjoint of A[N][N] in adj[N][N].
void adjoint(int A[SIZE][SIZE], int adj[SIZE][SIZE])
{
    if (SIZE == 1)
    {
        adj[0][0] = 1;
        return;
    }

    // temp is used to store cofactors of A[][]
    int sign = 1, temp[SIZE][SIZE];

    for (int i=0; i<SIZE; i++)
    {
        for (int j=0; j<SIZE; j++)
        {
            // Get cofactor of A[i][j]
            getCofactor(A, temp, i, j, SIZE);

            // sign of adj[j][i] positive if sum of row
            // and column indexes is even.
            sign = ((i+j)%2==0)? 1: -1;

            // Interchanging rows and columns to get the
            // transpose of the cofactor matrix
            adj[j][i] = (sign)*(determinant(temp, SIZE-1));
        }
    }
}

// Function to get cofactor of A[p][q] in temp[][]. n is current
// dimension of A[][]
void getCofactor(int A[SIZE][SIZE], int temp[SIZE][SIZE], int p, int q, int n)
{
    int i = 0, j = 0;

    // Looping for each element of the matrix
    for (int row = 0; row < n; row++)
    {
        for (int col = 0; col < n; col++)
        {
            //  Copying into temporary matrix only those element
            //  which are not in given row and column
            if (row != p && col != q)
            {
                temp[i][j++] = A[row][col];

                // Row is filled, so increase row index and
                // reset col index
                if (j == n - 1)
                {
                    j = 0;
                    i++;
                }
            }
        }
    }
}
```

```c
/* Recursive function for finding determinant of matrix.
   n is current dimension of A[][]. */
int determinant(int A[SIZE][SIZE], int n)
{
    int D = 0; // Initialize result

    //  Base case : if matrix contains single element
    if (n == 1)
        return A[0][0];

    int temp[SIZE][SIZE]; // To store cofactors

    int sign = 1;  // To store sign multiplier

     // Iterate for each element of first row
    for (int f = 0; f < n; f++)
    {
        // Getting Cofactor of A[0][f]
        getCofactor(A, temp, 0, f, n);
        D += sign * A[0][f] * determinant(temp, n - 1);

        // terms are to be added with alternate sign
        sign = -sign;
    }

    return D;
}
```

2.2.4. Windows

2.2.5. Funcionamiento

Al correr el programa este nos indicará en pantalla cuando este enviando el resultado de cada maatriz respectiva a un archivo:

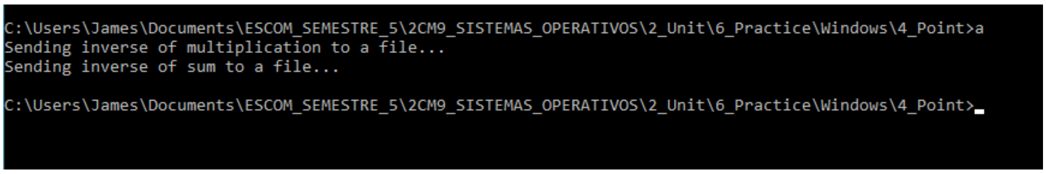


Figura 4:

Posteriormente podremos ver los archivos dentro del directorio de trabajo:

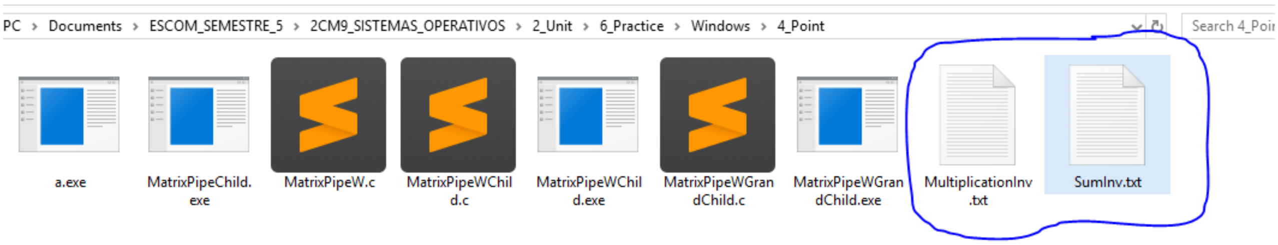


Figura 5:

Y su contenido:

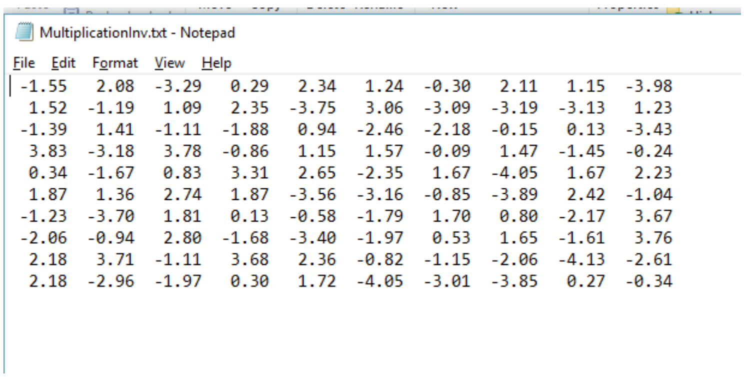


Figura 6:

2.2.6. Código Padre

```
//Program that uses process inter communication to process two matrices
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdbool.h>
#include <windows.h>

#define SIZE 10
#define BUF_SIZE 1000

bool inverse(int A[SIZE][SIZE], float inverse[SIZE][SIZE]);
void adjoint(int A[SIZE][SIZE], int adj[SIZE][SIZE]);
void getCofactor(int A[SIZE][SIZE], int temp[SIZE][SIZE], int p, int q, int n);
int determinant(int A[SIZE][SIZE], int n);

int main(void)
{
	int i, j, offset;
	char *wbuffer, *rbuffer;
	int **matrix1, **matrix2, result_matrix[SIZE][SIZE];
	float inv[SIZE][SIZE];
	FILE *write_fp;
	DWORD written, read;
	HANDLE pipefd[2];
	PROCESS_INFORMATION piChild;
	STARTUPINFO siChild;
	SECURITY_ATTRIBUTES pipeSec = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};

	srand((unsigned) time(NULL));

	//Asign memory space for both matrices and buffers
	matrix1 = malloc(SIZE * sizeof(int *));
	matrix2 = malloc(SIZE * sizeof(int *));
	wbuffer = malloc(BUF_SIZE);
	rbuffer = malloc(BUF_SIZE);
	for(i = 0; i < SIZE; i++)
	{
		matrix1[i] = malloc(SIZE * sizeof(int));
		matrix2[i] = malloc(SIZE * sizeof(int));
	}

	//Fill the matrices with random values
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
		{
			matrix1[i][j] = rand() % 5;
			matrix2[i][j] = rand() % 5;
		}

	//Create the first pipe
	//Obtencion de informaci n para la inicializaci n del proceso hijo
	GetStartupInfo(&siChild);
	//Creacion de la tuber a sin nombre
	CreatePipe(&pipefd[0], &pipefd[1], &pipeSec, 0);

	//Write the matrix1 into wbuffer
	for(i = 0; i < SIZE; i++)
	{
		for(j = 0; j < SIZE; j++)
			sprintf(wbuffer + strlen(wbuffer), "%d ", matrix1[i][j]);
		sprintf(wbuffer + strlen(wbuffer), "\n");
	}
	//Write the matrix2 into wbuffer
	for(i = 0; i < SIZE; i++)
	{
		for(j = 0; j < SIZE; j++)
			sprintf(wbuffer + strlen(wbuffer), "%d ", matrix2[i][j]);
```

```
		sprintf(wbuffer + strlen(wbuffer), "\n");
	}
	//Write matrix1 and matrix2 to the first pipe
	WriteFile(pipefd[1], wbuffer, strlen(wbuffer), &written, NULL);

	//Hereda el proceso hijo los manejadores de la tuber a del proceso padre
	siChild.hStdInput = pipefd[0];
	siChild.hStdError = GetStdHandle(STD_ERROR_HANDLE);
	siChild.hStdOutput = pipefd[1];
	siChild.dwFlags = STARTF_USESTDHANDLES;
	if(!CreateProcess(NULL, "MatrixPipeWChild", NULL, NULL, TRUE, 0, NULL, NULL
	    , &siChild, &piChild))
		fprintf(stderr, "Error\n");

	//Read from first pipe result of multiplication and sum
	//Wait necessary this time
	WaitForSingleObject(piChild.hProcess, INFINITE);
	ReadFile(pipefd[0], rbuffer, BUF_SIZE, &read, NULL);

	//Result of sum always is written first on the pipe at least on Windows
	//Read into result_matrix the result of multiplication from rbuffer
	offset = 0;
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
		{
			sscanf(rbuffer, "␣%d%n", &result_matrix[i][j], &offset);
			rbuffer += offset;
		}
	//Calculate inverse of multiplication and send it to a file
	write_fp = fopen("SumInv.txt", "w");
	printf("Sending␣inverse␣of␣multiplication␣to␣a␣file...\n");
	if(inverse(result_matrix, inv))
	{
		for(i = 0; i < SIZE; i++)
		{
			for(j = 0; j < SIZE; j++)
				fprintf(write_fp, "%6.2f␣", inv[i][j]);
			fprintf(write_fp, "\n");
		}
	}
	fclose(write_fp);

	//Read into result_matrix the result of sum from rbuffer
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
		{
			sscanf(rbuffer, "␣%d%n", &result_matrix[i][j], &offset);
			rbuffer += offset;
		}
	//Calculate inverse of sum and send it to a file
	write_fp = fopen("MultiplicationInv.txt", "w");
	printf("Sending␣inverse␣of␣sum␣to␣a␣file...\n");
	if(inverse(result_matrix, inv))
	{
		for(i = 0; i < SIZE; i++)
		{
			for(j = 0; j < SIZE; j++)
				fprintf(write_fp, "%6.2f␣", inv[i][j]);
			fprintf(write_fp, "\n");
		}
	}
	fclose(write_fp);

	CloseHandle(pipefd[0]);
	CloseHandle(pipefd[1]);
	CloseHandle(piChild.hThread);
	CloseHandle(piChild.hProcess);

	return 0;
}
```

```
// Function to calculate and store inverse, returns false if
// matrix is singular
bool inverse(int A[SIZE][SIZE], float inverse[SIZE][SIZE])
{
    // Find determinant of A[][]
    int det = determinant(A, SIZE);
    if (det == 0)
    {
        printf("Singular matrix, can't find its inverse\n");
        return false;
    }

    // Find adjoint
    int adj[SIZE][SIZE];
    adjoint(A, adj);

    // Find Inverse using formula "inverse(A) = adj(A)/det(A)"
    for (int i=0; i<SIZE; i++)
        for (int j=0; j<SIZE; j++)
            inverse[i][j] = adj[i][j]/(float) det;

    return true;
}

// Function to get adjoint of A[N][N] in adj[N][N].
void adjoint(int A[SIZE][SIZE], int adj[SIZE][SIZE])
{
    if (SIZE == 1)
    {
        adj[0][0] = 1;
        return;
    }

    // temp is used to store cofactors of A[][]
    int sign = 1, temp[SIZE][SIZE];

    for (int i=0; i<SIZE; i++)
    {
        for (int j=0; j<SIZE; j++)
        {
            // Get cofactor of A[i][j]
            getCofactor(A, temp, i, j, SIZE);

            // sign of adj[j][i] positive if sum of row
            // and column indexes is even.
            sign = ((i+j)%2==0)? 1: -1;

            // Interchanging rows and columns to get the
            // transpose of the cofactor matrix
            adj[j][i] = (sign)*(determinant(temp, SIZE-1));
        }
    }
}

// Function to get cofactor of A[p][q] in temp[][]. n is current
// dimension of A[][]
void getCofactor(int A[SIZE][SIZE], int temp[SIZE][SIZE], int p, int q, int n)
{
    int i = 0, j = 0;

    // Looping for each element of the matrix
    for (int row = 0; row < n; row++)
    {
        for (int col = 0; col < n; col++)
        {
            //  Copying into temporary matrix only those element
            //  which are not in given row and column
            if (row != p && col != q)
            {
```

```
                    temp[i][j++] = A[row][col];

                    // Row is filled, so increase row index and
                    // reset col index
                    if (j == n - 1)
                    {
                        j = 0;
                        i++;
                    }
                }
            }
        }
}

/* Recursive function for finding determinant of matrix.
   n is current dimension of A[][]. */
int determinant(int A[SIZE][SIZE], int n)
{
    int D = 0; // Initialize result

    //  Base case : if matrix contains single element
    if (n == 1)
        return A[0][0];

    int temp[SIZE][SIZE]; // To store cofactors

    int sign = 1;  // To store sign multiplier

     // Iterate for each element of first row
    for (int f = 0; f < n; f++)
    {
        // Getting Cofactor of A[0][f]
        getCofactor(A, temp, 0, f, n);
        D += sign * A[0][f] * determinant(temp, n - 1);

        // terms are to be added with alternate sign
        sign = -sign;
    }

    return D;
}
```

2.2.7. Código Hijo

```
//Program that uses process inter communication to process two matrices
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdbool.h>
#include <windows.h>

#define SIZE 10
#define BUF_SIZE 1000

int main(void)
{
        int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE], result_matrix[SIZE][SIZE], i,
            j, offset = 0, accumulator;
        char *wbuffer, *rbuffer;
        DWORD written, read;
        HANDLE pipefd1[2], pipefd[2];
        PROCESS_INFORMATION piChild;
        STARTUPINFO siChild;
        SECURITY_ATTRIBUTES pipeSec = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};

        //Get descriptors of first pipe
        pipefd[0] = GetStdHandle(STD_INPUT_HANDLE);
```

```
pipefd[1] = GetStdHandle(STD_OUTPUT_HANDLE);

//Allocate memory for buffers
wbuffer = malloc(BUF_SIZE);
rbuffer = malloc(BUF_SIZE);

//Read matrix1 and matrix2 from first pipe
ReadFile(pipefd[0], rbuffer, BUF_SIZE, &read, NULL);

//Read into matrix1 from rbuffer
for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
        {
                sscanf(rbuffer, "␣%d%n", &matrix1[i][j], &offset);
                rbuffer += offset;
        }
//Read into matrix2 from rbuffer
for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
        {
                sscanf(rbuffer, "␣%d%n", &matrix2[i][j], &offset);
                rbuffer += offset;
        }
//Create the second pipe
//Obtencion de informaci n para la inicializaci n del proceso hijo
GetStartupInfo(&siChild);
//Creacion de la tuber a sin nombre
CreatePipe(&pipefd1[0], &pipefd1[1], &pipeSec, 0);

//Write the matrix1 into wbuffer
for(i = 0; i < SIZE; i++)
{
        for(j = 0; j < SIZE; j++)
                sprintf(wbuffer + strlen(wbuffer), "%d␣", matrix1[i][j]);
        sprintf(wbuffer + strlen(wbuffer), "\n");
}
//Write the matrix2 into wbuffer
for(i = 0; i < SIZE; i++)
{
        for(j = 0; j < SIZE; j++)
                sprintf(wbuffer + strlen(wbuffer), "%d␣", matrix2[i][j]);
        sprintf(wbuffer + strlen(wbuffer), "\n");
}
//Write matrix1 and matrix2 to the second pipe
WriteFile(pipefd1[1], wbuffer, strlen(wbuffer), &written, NULL);

//Hereda el proceso hijo los manejadores de la tuber a del proceso padre
siChild.hStdInput = pipefd1[0];
siChild.hStdError = GetStdHandle(STD_ERROR_HANDLE);
siChild.hStdOutput = pipefd[1];
siChild.dwFlags = STARTF_USESTDHANDLES;
CreateProcess(NULL, "MatrixPipeWGrandChild", NULL, NULL, TRUE, 0, NULL,
    NULL, &siChild, &piChild);

//Do multiplication
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;
        }
}
//Clean wbuffer
for(i = 0; i < BUF_SIZE; i++)
        wbuffer[i] = '\0';
//Write multiplication into wbuffer
for(i = 0; i < SIZE; i++)
```

```
	{
		for(j = 0; j < SIZE; j++)
			sprintf(wbuffer + strlen(wbuffer), "%d ", result_matrix[i][
			    j]);
		sprintf(wbuffer + strlen(wbuffer), "\n");
	}
	//Write to first pipe the result of multiplication
	WriteFile(pipefd[1], wbuffer, strlen(wbuffer), &written, NULL);

	CloseHandle(piChild.hThread);
	CloseHandle(piChild.hProcess);

	return 0;
}
```

2.2.8. Código Nieto

```
//Program that uses process inter communication to process two matrices
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdbool.h>
#include <windows.h>

#define SIZE 10
#define BUF_SIZE 1000

int main(void)
{
	int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE], result_matrix[SIZE][SIZE], i,
	    j, offset = 0;
	char *wbuffer, *rbuffer;
	DWORD written, read;
	//To be congurent I declared both arrays but they can be ommited since we
	    only use pipefd1[0] and pipefd[1]
	HANDLE pipefd[2], pipefd1[2];

	//Get descriptors
	pipefd1[0] = GetStdHandle(STD_INPUT_HANDLE);
	pipefd[1] = GetStdHandle(STD_OUTPUT_HANDLE);

	//Allocate memory for buffers
	wbuffer = malloc(BUF_SIZE);
	rbuffer = malloc(BUF_SIZE);

	//Read matrix1 and matrix2 from pipe
	ReadFile(pipefd1[0], rbuffer, BUF_SIZE, &read, NULL);

	//Read into matrix1 from rbuffer
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
		{
			sscanf(rbuffer, " %d%n", &matrix1[i][j], &offset);
			rbuffer += offset;
		}
	//Read into matrix2 from rbuffer
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
		{
			sscanf(rbuffer, " %d%n", &matrix2[i][j], &offset);
			rbuffer += offset;
		}
	//Do sum
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
			result_matrix[i][j] = matrix1[i][j] + matrix2[i][j];
```

```
        //Write the resultx of sum into wbuffer
        for(i = 0; i < SIZE; i++)
        {
                for(j = 0; j < SIZE; j++)
                        sprintf(wbuffer + strlen(wbuffer), "%d ", result_matrix[i][
                            j]);
                sprintf(wbuffer + strlen(wbuffer), "\n");
        }
        //Write to first pipe the result of sum
        WriteFile(pipefd[1], wbuffer, strlen(wbuffer), &written, NULL);

        return 0;
}
```

2.3. Punto 7

Programe la misma aplicación del punto cuatro utilizando en esta ocasión memoria compartida en lugar de tuberías (utilice tantas memorias compartidas como requiera). Programe esta aplicación tanto para Linux como para Windows utilizando la memoria compartida de cada sistema operativo.

2.3.1. Linux

2.3.2. Funcionamiento

Al correr el programa este nos indicará en pantalla cuando este enviando el resultado de cada maatriz respectiva a un archivo:

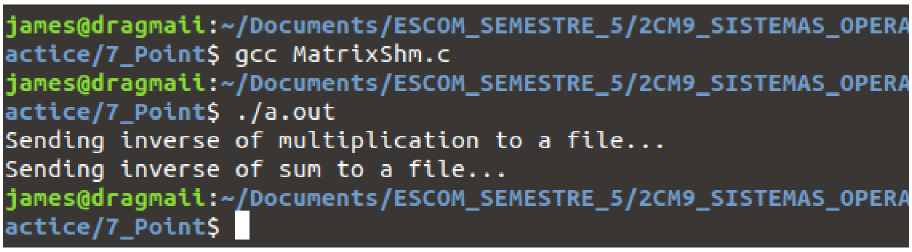


Figura 7:

Posteriormente podremos ver los archivos dentro del directorio de trabajo:

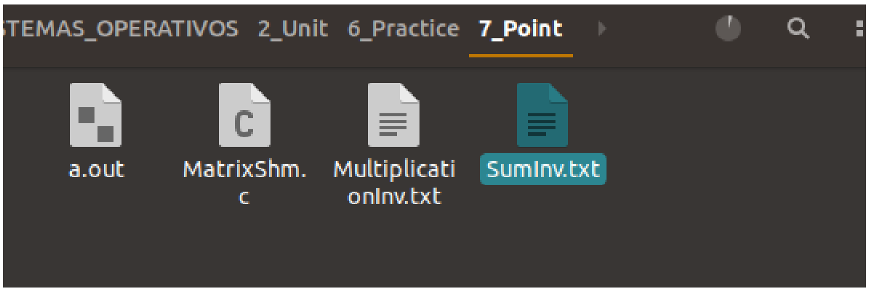
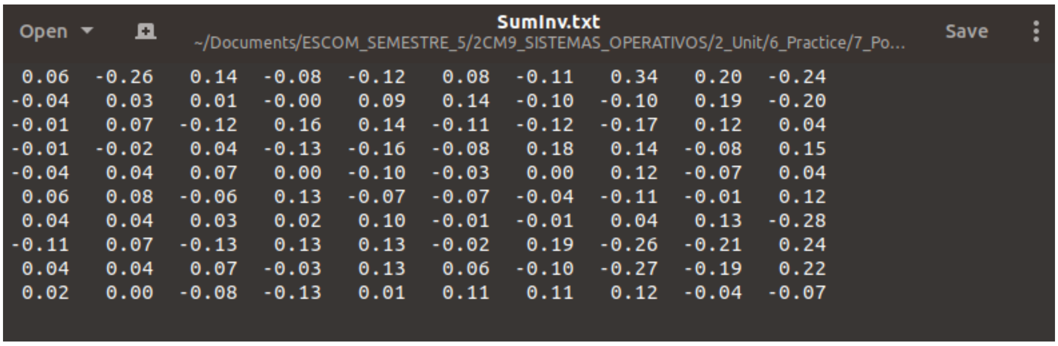


Figura 8:

Y su contenido:



0.06	-0.26	0.14	-0.08	-0.12	0.08	-0.11	0.34	0.20	-0.24
-0.04	0.03	0.01	-0.00	0.09	0.14	-0.10	-0.10	0.19	-0.20
-0.01	0.07	-0.12	0.16	0.14	-0.11	-0.12	-0.17	0.12	0.04
-0.01	-0.02	0.04	-0.13	-0.16	-0.08	0.18	0.14	-0.08	0.15
-0.04	0.04	0.07	0.00	-0.10	-0.03	0.00	0.12	-0.07	0.04
0.06	0.08	-0.06	0.13	-0.07	-0.07	-0.04	-0.11	-0.01	0.12
0.04	0.04	0.03	0.02	0.10	-0.01	-0.01	0.04	0.13	-0.28
-0.11	0.07	-0.13	0.13	0.13	-0.02	0.19	-0.26	-0.21	0.24
0.04	0.04	0.07	-0.03	0.13	0.06	-0.10	-0.27	-0.19	0.22
0.02	0.00	-0.08	-0.13	0.01	0.11	0.11	0.12	-0.04	-0.07

Figura 9:

2.3.3. Código

```
//Program that uses process inter communication to process two matrices
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <stdbool.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SIZE 10
#define MEM_SIZE 5000

bool inverse(int A[SIZE][SIZE], float inverse[SIZE][SIZE]);
void adjoint(int A[SIZE][SIZE], int adj[SIZE][SIZE]);
void getCofactor(int A[SIZE][SIZE], int temp[SIZE][SIZE], int p, int q, int n);
int determinant(int A[SIZE][SIZE], int n);

int main(void)
{
	int i, j, k;
	int *buffer, shmid, *shm;
	int **matrix1, **matrix2, result_matrix[SIZE][SIZE];
	float inv[SIZE][SIZE];
	FILE *write_fp;

	srand((unsigned) time(NULL));

	//Asign memory space for both matrices
	matrix1 = malloc(SIZE * sizeof(int *));
	matrix2 = malloc(SIZE * sizeof(int *));
	for(i = 0; i < SIZE; i++)
	{
		matrix1[i] = malloc(SIZE * sizeof(int));
		matrix2[i] = malloc(SIZE * sizeof(int));
	}

	//Fill the matrices with random values
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
		{
			matrix1[i][j] = rand() % 5;
			matrix2[i][j] = rand() % 5;
		}

	//ftok to generate an unique key
	key_t key = 2018;
	//shmget returns an identifier in shmid
	if((shmid = shmget(key, MEM_SIZE, 0666 | IPC_CREAT)) < 0)
	{
		perror("Failed to get shared memory: shmget");
		exit(EXIT_FAILURE);
	}
	//shmat to attach to shared memory
	if((shm = shmat(shmid, NULL, 0)) == (int *) -1)
	{
		perror("Failed to attach to shared memory: shmat");
		exit(EXIT_FAILURE);
	}
	//Make a pointer to the shared memory named buffer
	buffer = shm;

	//Write both matrices to the shared memory
	k = 0;
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
```

```
                    buffer[k++] = matrix1[i][j];

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

    //Put a marker to indicate end of matrices
    buffer[k] = -1000;
    //Create child process
    if(fork() == 0)
    {
            int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE], result_matrix[SIZE][
                SIZE], i, j, k, accumulator;

            //Create child process
            if(fork() == 0)
            {
                    int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE], result_matrix
                        [SIZE][SIZE], i, j, k;

                    //Copy into the matrices the values stored into shared
                        memory
                    k = 0;
                    for(i = 0; i < SIZE; i++)
                            for(j = 0; j < SIZE; j++)
                                    matrix1[i][j] = buffer[k++];

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

                    //Do sum
                    for(i = 0; i < SIZE; i++)
                            for(j = 0; j < SIZE; j++)
                                    result_matrix[i][j] = matrix1[i][j] +
                                        matrix2[i][j];

                    //Wait until multiplication is written into shared memory
                    k = 3 * SIZE * SIZE;
                    while(buffer[k] != -2000)
                            sleep(1);

                    //Write sum into shared memory starting from -2000 marker
                        at 3 * SIZE * SIZE
                    k = 3 * SIZE * SIZE;
                    for(i = 0; i < SIZE; i++)
                            for(j = 0; j < SIZE; j++)
                                    buffer[k++] = result_matrix[i][j];
                    //Put a marker to indicate end of sum
                    buffer[k] = -3000;
                    exit(EXIT_SUCCESS);
            }
            else
            {
                    //Copy into the matrices the values stored into shared
                        memory
                    k = 0;
                    for(i = 0; i < SIZE; i++)
                            for(j = 0; j < SIZE; j++)
                                    matrix1[i][j] = buffer[k++];

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

                    //Do multiplication
                    for(j = 0; j < SIZE; j++)
                    {
                            for(k = 0; k < SIZE; k++)
                            {
```

```c
                                        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;
                                }
                        }
                        //Write multiplication into shared memory starting from
                             -1000 marker at 2 * SIZE * SIZE
                        k = 2 * SIZE * SIZE;
                        for(i = 0; i < SIZE; i++)
                                for(j = 0; j < SIZE; j++)
                                        buffer[k++] = result_matrix[i][j];
                        buffer[k] = -2000;
                }
                exit(EXIT_SUCCESS);
        }
        //Father
        else
        {
                k = 4 * SIZE * SIZE;
                while(buffer[k] != -3000)
                        sleep(1);
                /*
                //Print all the content into shared memory
                for(i = 0; i < SIZE * SIZE; i++)
                {
                        if(i %10 == 0 && i != 0)
                                printf("\n");
                        printf("%d ", buffer[i]);
                }
                printf("\n");
                for(; i < 2 * SIZE * SIZE; i++)
                {
                        if(i %10 == 0 && i != 0)
                                printf("\n");
                        printf("%d ", buffer[i]);
                }
                printf("\n");

                for(; i < 3 * SIZE * SIZE; i++)
                {
                        if(i %10 == 0 && i != 0)
                                printf("\n");
                        printf("%d ", buffer[i]);
                }
                printf("\n");

                for(; i < 4 * SIZE * SIZE; i++)
                {
                        if(i %10 == 0 && i != 0)
                                printf("\n");
                        printf("%d ", buffer[i]);
                }
                printf("\n");
                */

                //Read into result_matrix the result of multiplication from shared
                     memory
                k = 2 * SIZE * SIZE;
                for(i = 0; i < SIZE; i++)
                        for(j = 0; j < SIZE; j++)
                                result_matrix[i][j] = buffer[k++];

                //Calculate inverse of multiplication and send it to a file
                write_fp = fopen("MultiplicationInv.txt", "w");
                printf("Sending inverse of multiplication to a file...\n");
                if(inverse(result_matrix, inv))
                {
```

```
                        for(i = 0; i < SIZE; i++)
                        {
                                for(j = 0; j < SIZE; j++)
                                        fprintf(write_fp, "%6.2f ", inv[i][j]);
                                fprintf(write_fp, "\n");
                        }
                }
                fclose(write_fp);

                //Read into result_matrix the result of sum from shared memory
                k = 3 * SIZE * SIZE;
                for(i = 0; i < SIZE; i++)
                        for(j = 0; j < SIZE; j++)
                                result_matrix[i][j] = buffer[k++];

                //Calculate inverse of sum and send it to a file
                write_fp = fopen("SumInv.txt", "w");
                printf("Sending inverse of sum to a file...\n");
                if(inverse(result_matrix, inv))
                {
                        for(i = 0; i < SIZE; i++)
                        {
                                for(j = 0; j < SIZE; j++)
                                        fprintf(write_fp, "%6.2f ", inv[i][j]);
                                fprintf(write_fp, "\n");
                        }
                }
                fclose(write_fp);
        }

        return 0;
}

// Function to calculate and store inverse, returns false if
// matrix is singular
bool inverse(int A[SIZE][SIZE], float inverse[SIZE][SIZE])
{
    // Find determinant of A[][]
    int det = determinant(A, SIZE);
    if (det == 0)
    {
        printf("Singular matrix, can't find its inverse\n");
        return false;
    }

    // Find adjoint
    int adj[SIZE][SIZE];
    adjoint(A, adj);

    // Find Inverse using formula "inverse(A) = adj(A)/det(A)"
    for (int i=0; i<SIZE; i++)
        for (int j=0; j<SIZE; j++)
            inverse[i][j] = adj[i][j]/(float) det;

    return true;
}

// Function to get adjoint of A[N][N] in adj[N][N].
void adjoint(int A[SIZE][SIZE], int adj[SIZE][SIZE])
{
    if (SIZE == 1)
    {
        adj[0][0] = 1;
        return;
    }

    // temp is used to store cofactors of A[][]
    int sign = 1, temp[SIZE][SIZE];

    for (int i=0; i<SIZE; i++)
```

```
    {
        for (int j=0; j<SIZE; j++)
        {
            // Get cofactor of A[i][j]
            getCofactor(A, temp, i, j, SIZE);

            // sign of adj[j][i] positive if sum of row
            // and column indexes is even.
            sign = ((i+j)%2==0)? 1: -1;

            // Interchanging rows and columns to get the
            // transpose of the cofactor matrix
            adj[j][i] = (sign)*(determinant(temp, SIZE-1));
        }
    }
}

// Function to get cofactor of A[p][q] in temp[][]. n is current
// dimension of A[][]
void getCofactor(int A[SIZE][SIZE], int temp[SIZE][SIZE], int p, int q, int n)
{
    int i = 0, j = 0;

    // Looping for each element of the matrix
    for (int row = 0; row < n; row++)
    {
        for (int col = 0; col < n; col++)
        {
            //  Copying into temporary matrix only those element
            //  which are not in given row and column
            if (row != p && col != q)
            {
                temp[i][j++] = A[row][col];

                // Row is filled, so increase row index and
                // reset col index
                if (j == n - 1)
                {
                    j = 0;
                    i++;
                }
            }
        }
    }
}

/* Recursive function for finding determinant of matrix.
   n is current dimension of A[][]. */
int determinant(int A[SIZE][SIZE], int n)
{
    int D = 0; // Initialize result

    //  Base case : if matrix contains single element
    if (n == 1)
        return A[0][0];

    int temp[SIZE][SIZE]; // To store cofactors

    int sign = 1;  // To store sign multiplier

     // Iterate for each element of first row
    for (int f = 0; f < n; f++)
    {
        // Getting Cofactor of A[0][f]
        getCofactor(A, temp, 0, f, n);
        D += sign * A[0][f] * determinant(temp, n - 1);

        // terms are to be added with alternate sign
        sign = -sign;
    }
```

```
    return D;
}
```

2.3.4. Windows

2.3.5. Funcionamiento

Al correr el programa este nos indicará en pantalla cuando este enviando el resultado de cada maatriz respectiva a un archivo:

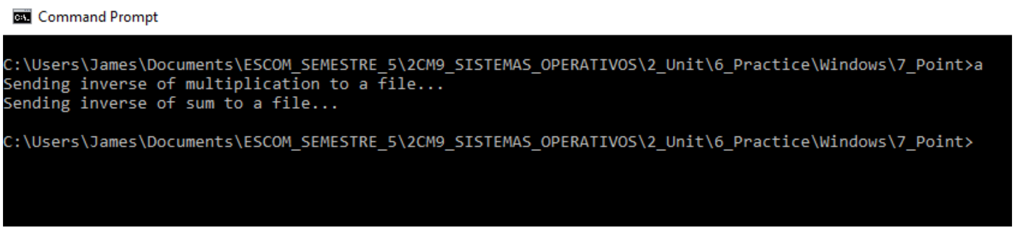


Figura 10:

Posteriormente podremos ver los archivos dentro del directorio de trabajo:

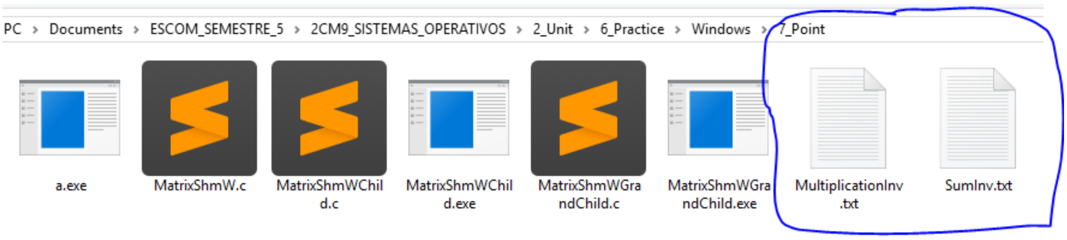
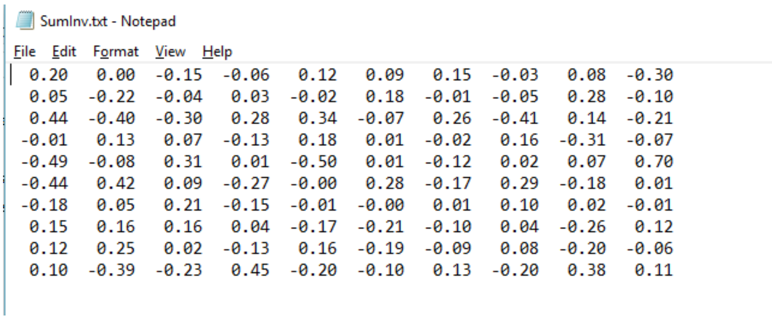


Figura 11:

Y su contenido:



0.20	0.00	-0.15	-0.06	0.12	0.09	0.15	-0.03	0.08	-0.30
0.05	-0.22	-0.04	0.03	-0.02	0.18	-0.01	-0.05	0.28	-0.10
0.44	-0.40	-0.30	0.28	0.34	-0.07	0.26	-0.41	0.14	-0.21
-0.01	0.13	0.07	-0.13	0.18	0.01	-0.02	0.16	-0.31	-0.07
-0.49	-0.08	0.31	0.01	-0.50	0.01	-0.12	0.02	0.07	0.70
-0.44	0.42	0.09	-0.27	-0.00	0.28	-0.17	0.29	-0.18	0.01
-0.18	0.05	0.21	-0.15	-0.01	-0.00	0.01	0.10	0.02	-0.01
0.15	0.16	0.16	0.04	-0.17	-0.21	-0.10	0.04	-0.26	0.12
0.12	0.25	0.02	-0.13	0.16	-0.19	-0.09	0.08	-0.20	-0.06
0.10	-0.39	-0.23	0.45	-0.20	-0.10	0.13	-0.20	0.38	0.11

Figura 12:

2.3.6. Código Padre

```c
//Program that uses process inter communication to process two matrices
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdbool.h>
#include <windows.h>

#define SIZE 10
#define BUF_SIZE 1000

bool inverse(int A[SIZE][SIZE], float inverse[SIZE][SIZE]);
void adjoint(int A[SIZE][SIZE], int adj[SIZE][SIZE]);
void getCofactor(int A[SIZE][SIZE], int temp[SIZE][SIZE], int p, int q, int n);
int determinant(int A[SIZE][SIZE], int n);

int main(void)
{
	int i, j, offset;
	char *wbuffer, *rbuffer;
	int **matrix1, **matrix2, result_matrix[SIZE][SIZE];
	float inv[SIZE][SIZE];
	FILE *write_fp;
	DWORD written, read;
	HANDLE pipefd[2];
	PROCESS_INFORMATION piChild;
	STARTUPINFO siChild;
	SECURITY_ATTRIBUTES pipeSec = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};

	srand((unsigned) time(NULL));

	//Asign memory space for both matrices and buffers
	matrix1 = malloc(SIZE * sizeof(int *));
	matrix2 = malloc(SIZE * sizeof(int *));
	wbuffer = malloc(BUF_SIZE);
	rbuffer = malloc(BUF_SIZE);
	for(i = 0; i < SIZE; i++)
	{
		matrix1[i] = malloc(SIZE * sizeof(int));
		matrix2[i] = malloc(SIZE * sizeof(int));
	}

	//Fill the matrices with random values
	for(i = 0; i < SIZE; i++)
		for(j = 0; j < SIZE; j++)
		{
			matrix1[i][j] = rand() % 5;
			matrix2[i][j] = rand() % 5;
		}

	//Create the first pipe
	//Obtencion de informaci n para la inicializaci n del proceso hijo
	GetStartupInfo(&siChild);
	//Creacion de la tuber a sin nombre
	CreatePipe(&pipefd[0], &pipefd[1], &pipeSec, 0);

	//Write the matrix1 into wbuffer
	for(i = 0; i < SIZE; i++)
	{
		for(j = 0; j < SIZE; j++)
			sprintf(wbuffer + strlen(wbuffer), "%d ", matrix1[i][j]);
		sprintf(wbuffer + strlen(wbuffer), "\n");
	}
	//Write the matrix2 into wbuffer
	for(i = 0; i < SIZE; i++)
	{
		for(j = 0; j < SIZE; j++)
			sprintf(wbuffer + strlen(wbuffer), "%d ", matrix2[i][j]);
```

```
            sprintf(wbuffer + strlen(wbuffer), "\n");
        }
        //Write matrix1 and matrix2 to the first pipe
        WriteFile(pipefd[1], wbuffer, strlen(wbuffer), &written, NULL);

        //Hereda el proceso hijo los manejadores de la tuber a del proceso padre
        siChild.hStdInput = pipefd[0];
        siChild.hStdError = GetStdHandle(STD_ERROR_HANDLE);
        siChild.hStdOutput = pipefd[1];
        siChild.dwFlags = STARTF_USESTDHANDLES;
        if(!CreateProcess(NULL, "MatrixPipeWChild", NULL, NULL, TRUE, 0, NULL, NULL
            , &siChild, &piChild))
            fprintf(stderr, "Error\n");

        //Read from first pipe result of multiplication and sum
        //Wait necessary this time
        WaitForSingleObject(piChild.hProcess, INFINITE);
        ReadFile(pipefd[0], rbuffer, BUF_SIZE, &read, NULL);

        //Result of sum always is written first on the pipe at least on Windows
        //Read into result_matrix the result of multiplication from rbuffer
        offset = 0;
        for(i = 0; i < SIZE; i++)
            for(j = 0; j < SIZE; j++)
            {
                sscanf(rbuffer, " %d%n", &result_matrix[i][j], &offset);
                rbuffer += offset;
            }
        //Calculate inverse of multiplication and send it to a file
        write_fp = fopen("SumInv.txt", "w");
        printf("Sending inverse of multiplication to a file...\n");
        if(inverse(result_matrix, inv))
        {
            for(i = 0; i < SIZE; i++)
            {
                for(j = 0; j < SIZE; j++)
                    fprintf(write_fp, "%6.2f ", inv[i][j]);
                fprintf(write_fp, "\n");
            }
        }
        fclose(write_fp);

        //Read into result_matrix the result of sum from rbuffer
        for(i = 0; i < SIZE; i++)
            for(j = 0; j < SIZE; j++)
            {
                sscanf(rbuffer, " %d%n", &result_matrix[i][j], &offset);
                rbuffer += offset;
            }
        //Calculate inverse of sum and send it to a file
        write_fp = fopen("MultiplicationInv.txt", "w");
        printf("Sending inverse of sum to a file...\n");
        if(inverse(result_matrix, inv))
        {
            for(i = 0; i < SIZE; i++)
            {
                for(j = 0; j < SIZE; j++)
                    fprintf(write_fp, "%6.2f ", inv[i][j]);
                fprintf(write_fp, "\n");
            }
        }
        fclose(write_fp);

        CloseHandle(pipefd[0]);
        CloseHandle(pipefd[1]);
        CloseHandle(piChild.hThread);
        CloseHandle(piChild.hProcess);

        return 0;
}
```

```
// Function to calculate and store inverse, returns false if
// matrix is singular
bool inverse(int A[SIZE][SIZE], float inverse[SIZE][SIZE])
{
    // Find determinant of A[][]
    int det = determinant(A, SIZE);
    if (det == 0)
    {
        printf("Singular matrix, can't find its inverse\n");
        return false;
    }

    // Find adjoint
    int adj[SIZE][SIZE];
    adjoint(A, adj);

    // Find Inverse using formula "inverse(A) = adj(A)/det(A)"
    for (int i=0; i<SIZE; i++)
        for (int j=0; j<SIZE; j++)
            inverse[i][j] = adj[i][j]/(float) det;

    return true;
}

// Function to get adjoint of A[N][N] in adj[N][N].
void adjoint(int A[SIZE][SIZE], int adj[SIZE][SIZE])
{
    if (SIZE == 1)
    {
        adj[0][0] = 1;
        return;
    }

    // temp is used to store cofactors of A[][]
    int sign = 1, temp[SIZE][SIZE];

    for (int i=0; i<SIZE; i++)
    {
        for (int j=0; j<SIZE; j++)
        {
            // Get cofactor of A[i][j]
            getCofactor(A, temp, i, j, SIZE);

            // sign of adj[j][i] positive if sum of row
            // and column indexes is even.
            sign = ((i+j)%2==0)? 1: -1;

            // Interchanging rows and columns to get the
            // transpose of the cofactor matrix
            adj[j][i] = (sign)*(determinant(temp, SIZE-1));
        }
    }
}

// Function to get cofactor of A[p][q] in temp[][]. n is current
// dimension of A[][]
void getCofactor(int A[SIZE][SIZE], int temp[SIZE][SIZE], int p, int q, int n)
{
    int i = 0, j = 0;

    // Looping for each element of the matrix
    for (int row = 0; row < n; row++)
    {
        for (int col = 0; col < n; col++)
        {
            //  Copying into temporary matrix only those element
            //  which are not in given row and column
            if (row != p && col != q)
            {
```

```
                    temp[i][j++] = A[row][col];

                    // Row is filled, so increase row index and
                    // reset col index
                    if (j == n - 1)
                    {
                        j = 0;
                        i++;
                    }
                }
            }
        }
}

/* Recursive function for finding determinant of matrix.
   n is current dimension of A[][]. */
int determinant(int A[SIZE][SIZE], int n)
{
    int D = 0; // Initialize result

    //  Base case : if matrix contains single element
    if (n == 1)
        return A[0][0];

    int temp[SIZE][SIZE]; // To store cofactors

    int sign = 1;  // To store sign multiplier

     // Iterate for each element of first row
    for (int f = 0; f < n; f++)
    {
        // Getting Cofactor of A[0][f]
        getCofactor(A, temp, 0, f, n);
        D += sign * A[0][f] * determinant(temp, n - 1);

        // terms are to be added with alternate sign
        sign = -sign;
    }

    return D;
}
```

2.3.7. Código Hijo

```
//Program that uses process inter communication to process two matrices
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdbool.h>
#include <windows.h>

#define SIZE 10
#define BUF_SIZE 1000

int main(void)
{
        int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE], result_matrix[SIZE][SIZE], i,
             j, offset = 0, accumulator;
        char *wbuffer, *rbuffer;
        DWORD written, read;
        HANDLE pipefd1[2], pipefd[2];
        PROCESS_INFORMATION piChild;
        STARTUPINFO siChild;
        SECURITY_ATTRIBUTES pipeSec = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};

        //Get descriptors of first pipe
        pipefd[0] = GetStdHandle(STD_INPUT_HANDLE);
```

```
pipefd[1] = GetStdHandle(STD_OUTPUT_HANDLE);

//Allocate memory for buffers
wbuffer = malloc(BUF_SIZE);
rbuffer = malloc(BUF_SIZE);

//Read matrix1 and matrix2 from first pipe
ReadFile(pipefd[0], rbuffer, BUF_SIZE, &read, NULL);

//Read into matrix1 from rbuffer
for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
        {
                sscanf(rbuffer, " %d%n", &matrix1[i][j], &offset);
                rbuffer += offset;
        }
//Read into matrix2 from rbuffer
for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
        {
                sscanf(rbuffer, " %d%n", &matrix2[i][j], &offset);
                rbuffer += offset;
        }
//Create the second pipe
//Obtencion de informaci n para la inicializaci n del proceso hijo
GetStartupInfo(&siChild);
//Creacion de la tuber a sin nombre
CreatePipe(&pipefd1[0], &pipefd1[1], &pipeSec, 0);

//Write the matrix1 into wbuffer
for(i = 0; i < SIZE; i++)
{
        for(j = 0; j < SIZE; j++)
                sprintf(wbuffer + strlen(wbuffer), "%d ", matrix1[i][j]);
        sprintf(wbuffer + strlen(wbuffer), "\n");
}
//Write the matrix2 into wbuffer
for(i = 0; i < SIZE; i++)
{
        for(j = 0; j < SIZE; j++)
                sprintf(wbuffer + strlen(wbuffer), "%d ", matrix2[i][j]);
        sprintf(wbuffer + strlen(wbuffer), "\n");
}
//Write matrix1 and matrix2 to the second pipe
WriteFile(pipefd1[1], wbuffer, strlen(wbuffer), &written, NULL);

//Hereda el proceso hijo los manejadores de la tuber a del proceso padre
siChild.hStdInput = pipefd1[0];
siChild.hStdError = GetStdHandle(STD_ERROR_HANDLE);
siChild.hStdOutput = pipefd[1];
siChild.dwFlags = STARTF_USESTDHANDLES;
CreateProcess(NULL, "MatrixPipeWGrandChild", NULL, NULL, TRUE, 0, NULL,
    NULL, &siChild, &piChild);

//Do multiplication
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;
        }
}
//Clean wbuffer
for(i = 0; i < BUF_SIZE; i++)
        wbuffer[i] = '\0';
//Write multiplication into wbuffer
for(i = 0; i < SIZE; i++)
```

```
        {
                for(j = 0; j < SIZE; j++)
                        sprintf(wbuffer + strlen(wbuffer), "%d ", result_matrix[i][
                            j]);
                sprintf(wbuffer + strlen(wbuffer), "\n");
        }
        //Write to first pipe the result of multiplication
        WriteFile(pipefd[1], wbuffer, strlen(wbuffer), &written, NULL);

        CloseHandle(piChild.hThread);
        CloseHandle(piChild.hProcess);

        return 0;
}
```

2.3.8. Código Nieto

```
//Program that uses process inter communication to process two matrices
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdbool.h>
#include <windows.h>

#define SIZE 10
#define BUF_SIZE 1000

int main(void)
{
        int matrix1[SIZE][SIZE], matrix2[SIZE][SIZE], result_matrix[SIZE][SIZE], i,
            j, offset = 0;
        char *wbuffer, *rbuffer;
        DWORD written, read;
        //To be congurent I declared both arrays but they can be ommited since we
            only use pipefd1[0] and pipefd[1]
        HANDLE pipefd[2], pipefd1[2];

        //Get descriptors
        pipefd1[0] = GetStdHandle(STD_INPUT_HANDLE);
        pipefd[1] = GetStdHandle(STD_OUTPUT_HANDLE);

        //Allocate memory for buffers
        wbuffer = malloc(BUF_SIZE);
        rbuffer = malloc(BUF_SIZE);

        //Read matrix1 and matrix2 from pipe
        ReadFile(pipefd1[0], rbuffer, BUF_SIZE, &read, NULL);

        //Read into matrix1 from rbuffer
        for(i = 0; i < SIZE; i++)
                for(j = 0; j < SIZE; j++)
                {
                        sscanf(rbuffer, " %d%n", &matrix1[i][j], &offset);
                        rbuffer += offset;
                }
        //Read into matrix2 from rbuffer
        for(i = 0; i < SIZE; i++)
                for(j = 0; j < SIZE; j++)
                {
                        sscanf(rbuffer, " %d%n", &matrix2[i][j], &offset);
                        rbuffer += offset;
                }
        //Do sum
        for(i = 0; i < SIZE; i++)
                for(j = 0; j < SIZE; j++)
                        result_matrix[i][j] = matrix1[i][j] + matrix2[i][j];
```

```
        //Write the resultx of sum into wbuffer
        for(i = 0; i < SIZE; i++)
        {
                for(j = 0; j < SIZE; j++)
                        sprintf(wbuffer + strlen(wbuffer), "%d ", result_matrix[i][
                            j]);
                sprintf(wbuffer + strlen(wbuffer), "\n");
        }
        //Write to first pipe the result of sum
        WriteFile(pipefd[1], wbuffer, strlen(wbuffer), &written, NULL);

        return 0;
}
```

3. Análisis Crítico

En está ocasión me pareció una práctica muy buena y el hecho de que hubo el tiempo suficiente para realizarla con tranquilidad fue mejor aún.

4. Observaciones

Como siempre las implementaciones en Windows acarrean diversos problemas, pero también en este caso fue interesante observar como se comportan ambos sistemas operativos pues por ejemplo, en Linux el nieto termina siempre después del hijo de escribir en la tubería, mientras que en Windows el nieto siempre escribe primero en la tubería.

5. Conclusión

La comunicación entre procesos resulta de muchísima ayuda a la hora de programar diferentes procesos, estos ayudan a una programación más acercada a la concurrencia aunque la implementación de esta comunicación entre procesos no es trivial pues debemos tener mucho cuidado a la hora de controlar la escritura y lectura de información a través de los distintos mecanismos.