

# Practica 8

Mecanismos de sincronización de procesos en Linux y Windows  
(semáforos)

## Integrantes

- Franco Rayas Ángel
- Damián
- - Gómez López
- Miguel Ángel
- - Vasco Giraldo Juan
- Esteban
- - López Coria Axel
- Yahir



## Profesor

Jorge Cortés Galicia

## Fecha de entrega

05/01/2025

# INTRODUCCIÓN

En los sistemas operativos modernos, la concurrencia es un aspecto fundamental que permite a múltiples procesos o hilos ejecutarse de manera simultánea. Sin embargo, este enfoque puede generar problemas cuando varios procesos intentan acceder al mismo recurso compartido, como un archivo, memoria o dispositivo de hardware. Para resolver este desafío, se han desarrollado mecanismos de sincronización que aseguran un acceso controlado a dichos recursos, garantizando la integridad de los datos y el correcto funcionamiento del sistema. Uno de los mecanismos más destacados para este propósito son los semáforos, que han demostrado ser esenciales en la programación concurrente.

En esencia, los semáforos funcionan como señales que los procesos utilizan para coordinar su acceso a recursos críticos, resolviendo problemas comunes de concurrencia y exclusión mutua. En los sistemas operativos Linux, los semáforos se gestionan con la función `semget()`. Estos pueden ser binarios (representando un estado de ocupado o libre) o contadores (indicando el número de recursos disponibles o procesos en espera). Las operaciones realizadas sobre los semáforos, como `wait()` y `signal()`, permiten sincronizar procesos en contextos como exclusión mutua y espera activa. Por otro lado, en Windows, los semáforos se crean con la función `CreateSemaphore()` y operan de manera similar, utilizando las funciones `WaitForSingleObject()` y `ReleaseSemaphore()` para gestionar su estado. Además, los semáforos en Windows son particularmente útiles para la sincronización de hilos dentro de un mismo proceso.

La implementación de semáforos permite a los desarrolladores evitar condiciones de carrera y garantizar un acceso ordenado a los recursos compartidos, especialmente en aplicaciones críticas como bases de datos, sistemas distribuidos y sistemas de tiempo real. Gracias a su flexibilidad, los semáforos son aplicables tanto en entornos simples como complejos, adaptándose a las necesidades de sincronización de cada sistema. Este mecanismo no solo mejora la eficiencia del sistema operativo, sino que también proporciona un entorno más seguro y predecible para la ejecución de tareas concurrentes.

## Desarrollo

**1. A través de la ayuda en línea que proporciona Linux, investigue el funcionamiento de las funciones: semget(), semop(). Explique los argumentos, retorno de las funciones y las estructuras relacionadas con dichas funciones.**

### Función

#### semget()

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

#### Argumentos

##### **key\_t key**

Identificador único para el conjunto de semáforos.

Puede ser una clave generada por ftok o IPC\_PRIVATE.

Sirve para identificar un conjunto de semáforos existente o para crear uno nuevo.

##### **int nsems**

Número de semáforos en el conjunto.

Solo se utiliza cuando se crea un nuevo conjunto. Debe ser mayor que 0 y menor o igual a SEMMSL (máximo permitido por el sistema).

##### **int semflg**

Especifica opciones de creación y permisos para el conjunto de semáforos:

IPC\_CREAT: Crea un conjunto si no existe uno asociado con la clave.

IPC\_EXCL: Falla si el conjunto ya existe (debe combinarse con IPC\_CREAT).

Permisos de acceso (9 bits menos significativos, como en open(2)).

### Retorno

En éxito

Retorna el identificador del conjunto de semáforos (un entero no negativo).

En error

Retorna -1 y establece errno con un valor que indica el error.

### Estructuras relacionadas

#### **sem\_perm**

uid: ID de usuario del propietario.

gid: ID de grupo del propietario.

cuid: ID de usuario del creador.

cgid: ID de grupo del creador.

mode: Permisos (9 bits menos significativos de semflg).

#### **sem\_nsems**

Número de semáforos en el conjunto.

#### **sem\_otime**

Última operación en cualquier semáforo del conjunto (inicializado a 0).

#### **sem\_ctime**

Tiempo de creación o último cambio en el conjunto.

## Función

### **semop()**

```
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

```
int semtimedop(int semid, struct sembuf *sops, size_t nsops,  
const struct timespec *_Nullable timeout);
```

## Argumentos

- int semid

Identificador del conjunto de semáforos, obtenido previamente mediante semget.

- struct sembuf \*sops:

Puntero a un array de estructuras sembuf que especifican las operaciones a realizar en el conjunto de semáforos.

Elementos para los array

- unsigned short sem\_num: Número del semáforo dentro del conjunto (empezando desde 0).
- short sem\_op: Operación a realizar:

Valor positivo: Incrementa el valor del semáforo (semval).

Cero (0): Espera hasta que el valor del semáforo sea cero.

Valor negativo: Decrementa el valor del semáforo, esperando si no es posible realizar la operación inmediatamente.

- short sem\_flg: Flags que afectan el comportamiento:
- IPC\_NOWAIT: No espera; falla inmediatamente si no puede completar la operación.
- SEM\_UNDO: La operación se deshace automáticamente al finalizar el proceso.
- size\_t nsops

Número de operaciones (tamaño del array sops).

- const struct timespec \*timeout (solo en semtimedop):

Tiempo máximo de espera cuando no se puede completar una operación de inmediato.

Si es NULL, semtimedop se comporta igual que semop.

## Retorno

En éxito

Ambas funciones retornan 0.

En error

Retornan -1 y establecen errno para indicar el error.

### Estructuras relacionadas

struct sembuf

Especifica las operaciones a realizar en semáforos individuales dentro del conjunto

semval

Valor actual del semáforo.

semzcent

Número de procesos esperando que el valor del semáforo sea 0.

semncnt

Número de procesos esperando que el valor del semáforo aumente.

sempid

ID del proceso que realizó la última operación sobre el semáforo.

**2. Capture, compile y ejecute el siguiente programa. Observe su funcionamiento y explique.**

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(void){
    int i,j;
```

```
int pid;
int semid;
key_t llave = 1234;
int semban = IPC_CREAT | 0666;
int nsems = 1;
int nsops;
struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));
printf("Iniciando semaforo...\n");

if ((semid = semget(llave, nsems, semban)) == -1) {
    perror("semget: error al iniciar semaforo");
    exit(1);
}

else{
    printf("Semaforo iniciado...\n");
}

if ((pid = fork()) < 0) {
    perror("fork: error al crear proceso\n");
    exit(1);
}

if (pid == 0) {
    i = 0;
    while (i < 3) {
        nsops = 2;
        sops[0].sem_num = 0;
        sops[0].sem_op = 0;
        sops[0].sem_flg = SEM_UNDO;
        sops[1].sem_num = 0;
```

```

sops[1].sem_op = 1;

sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT;

printf("semop: hijo llamando a semop(%d, &sops, %d) con:", semid, nsops);

for (j = 0; j < nsops; j++) {

    printf("\n\sops[%d].sem_num = %d, ", j, sops[j].sem_num);

    printf("sem_op = %d, ", sops[j].sem_op);

    printf("sem_flg = %#o\n", sops[j].sem_flg);

}

if ((j = semop(semid, sops, nsops)) == -1) {

    perror("semop: error en operacion del semaforo\n");

}

else {

    printf("\tsemop: regreso de semop() %d\n", j);

    printf("\n\nProceso hijo toma el control del semaforo: %d/3 veces\n", i+1);

    sleep(5);

    nsops = 1;

    sops[0].sem_num = 0;

    sops[0].sem_op = -1;

    sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT;

    if ((j = semop(semid, sops, nsops)) == -1) {

        perror("semop: error en operacion del semaforo\n");

    }

    else {

        printf("Proceso hijo regresa el control del semaforo: %d/3 veces\n", i+1);

        sleep(5);

    }

}

```

```

++i;

}

}

else {

    i = 0;

    while (i < 3) {

        nsops = 2;

        sops[0].sem_num = 0;

        sops[0].sem_op = 0;

        sops[0].sem_flg = SEM_UNDO;

        sops[1].sem_num = 0;

        sops[1].sem_op = 1;

        sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT;

        printf("\nsemop: Padre llamando semop(%d, &sops, %d) con:", semid, nsops);

        for (j = 0; j < nsops; j++) {

            printf("\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);

            printf("sem_op = %d, ", sops[j].sem_op);

            printf("sem_flg = %#o\n", sops[j].sem_flg);

        }

        if ((j = semop(semid, sops, nsops)) == -1) {

            perror("semop: error en operacion del semaforo\n");

        }

        else {

            printf("semop: regreso de semop() %d\n", j);

            printf("Proceso padre toma el control del semaforo: %d/3 veces\n", i+1);

            sleep(5);

            nsops = 1;

```

```
sops[0].sem_num = 0;

sops[0].sem_op = -1;

sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT;

if ((j = semop(semid, sops, nsops)) == -1) {

    perror("semop: error en semop()\n");

}

else {

    printf("Proceso padre regresa el control del semaforo: %d/3 veces\n", i+1);

    sleep(5);

}

}

++i;

}

}

}
```

## Salida

```
Iniciando semáforo...
Semáforo iniciado...

semop: Padre llamando semop(0, &sops, 2) con:
      sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000

      sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
semop: regreso de semop() 0
Proceso padre toma el control del semáforo: 1/3 veces
semop: hijo llamando a semop(0, &sops, 2) con:
      sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000

      sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
Proceso padre regresa el control del semáforo: 1/3 veces
semop: regreso de semop() 0

Proceso hijo toma el control del semáforo: 1/3 veces

semop: Padre llamando semop(0, &sops, 2) con:
      sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000

      sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
Proceso hijo regresa el control del semáforo: 1/3 veces
semop: regreso de semop() 0
Proceso padre toma el control del semáforo: 2/3 veces
Proceso padre regresa el control del semáforo: 2/3 veces
semop: hijo llamando a semop(0, &sops, 2) con:
      sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000

      sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
semop: regreso de semop() 0

Proceso hijo toma el control del semáforo: 2/3 veces

Proceso hijo regresa el control del semáforo: 2/3 veces
semop: Padre llamando semop(0, &sops, 2) con:
      sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000

      sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
semop: regreso de semop() 0
Proceso padre toma el control del semáforo: 3/3 veces
semop: hijo llamando a semop(0, &sops, 2) con:
Proceso padre regresa el control del semáforo: 3/3 veces
      sops[0].sem_num = 0, sem_op = 0, sem_flg = 010000

      sops[1].sem_num = 0, sem_op = 1, sem_flg = 014000
semop: regreso de semop() 0

Proceso hijo toma el control del semáforo: 3/3 veces
Proceso hijo regresa el control del semáforo: 3/3 veces
```

## Observaciones

Este programa en C utiliza semáforos para coordinar la ejecución de un proceso padre y su proceso hijo, asegurando que ambos accedan de manera controlada a un recurso compartido simulado. Primero, el programa crea un semáforo utilizando semget() con una llave única y permisos

específicos, verificando que pueda inicializarse correctamente. Luego, se utiliza fork() para dividir el flujo de ejecución en dos procesos: padre e hijo. Ambos procesos ejecutan un ciclo en el que realizan operaciones sobre el semáforo (semop()) para tomar y liberar su control. Estas operaciones aseguran que solo un proceso a la vez pueda "bloquear" el semáforo y simular la realización de una tarea crítica mientras impide que el otro proceso lo use.

Cada proceso, padre e hijo, intenta tomar el control del semáforo mediante una operación de espera (P) y, si tiene éxito, realiza una simulación de tarea durante cinco segundos (sleep(5)) antes de liberar el semáforo con una operación de señalización (V). Este comportamiento alternado se repite tres veces para cada proceso, lo que demuestra la sincronización efectiva utilizando semáforos. Si ocurre algún error durante las operaciones con el semáforo, se imprime un mensaje de error, y al final del programa, el semáforo permanece activo, aunque idealmente debería eliminarse con semctl() para liberar los recursos del sistema.

**3. A través de la ayuda del sitio MSDN, investigue el funcionamiento de las funciones: CreateSemaphore(), OpenSemaphore(), y ReleaseSemaphore. Explique los argumentos, retorno de las funciones y las estructuras relacionadas con dichas funciones.**

### **CreateSemaphore()**

Crea un nuevo semáforo con un valor inicial especificado.

#### **Argumentos:**

- lpSemaphoreAttributes: Puntero a una estructura SECURITY\_ATTRIBUTES que controla los atributos de seguridad del semáforo.
- bInitialCount: Valor inicial del semáforo. Indica el número de hilos que pueden acceder al recurso protegido inicialmente.
- lMaximumCount: Valor máximo que puede alcanzar el semáforo. Define el número máximo de hilos que pueden acceder al recurso simultáneamente.
- lpName: Nombre del semáforo. Si se proporciona un nombre, otros procesos pueden obtener un manejador al semáforo existente usando OpenSemaphore().

**Retorno:** Devuelve un manejador al semáforo recién creado si la operación es exitosa, o NULL en caso de error.

### **OpenSemaphore()**

Obtiene un manejador a un semáforo existente, identificado por su nombre.

#### **Argumentos:**

- dwDesiredAccess: Tipo de acceso que se solicita al semáforo (por ejemplo, SEMAPHORE\_MODIFY\_STATE para modificar el valor del semáforo).
- bInheritHandle: Indica si los procesos hijo heredarán el manejador.

- lpName: Nombre del semáforo.

**Retorno:** Devuelve un manejador al semáforo si se encuentra, o NULL en caso de error.

### **ReleaseSemaphore()**

Función: Incrementa el valor de un semáforo, permitiendo que un hilo adicional pueda acceder al recurso protegido.

#### **Argumentos:**

- hSemaphore: Manejador al semáforo.
- lReleaseCount: Número de unidades en que se incrementará el valor del semáforo.
- lpPreviousCount: Puntero a una variable DWORD donde se almacenará el valor del semáforo antes del incremento.

**Retorno:** Devuelve un valor booleano que indica si la operación se realizó correctamente.

### **SECURITY\_ATTRIBUTES**

Esta estructura se utiliza para controlar los atributos de seguridad de los objetos del sistema, como semáforos, mutexes y eventos.

**4. Capture, compile y ejecute los siguientes programas. Observe su funcionamiento. Ejecute de la siguiente forma: C:\>nombre\_programa\_padre nombre\_programa\_hijo.**

## PROGRAMA PADRE

```
#include <windows.h> /*Programa padre*/
#include <stdio.h>

int main(int argc, char *argv[]){
    STARTUPINFO si; /* Estructura de información inicial para Windows */
    PROCESS_INFORMATION pi; /* Estructura de información del adm. de procesos */
    HANDLE hSemaforo;
    int i=1;

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

    if(argc!=2) {
        printf("Usar: %s Nombre_programa_hijo\n", argv[0]);
        return;
    }

    // Creación del semáforo
    if((hSemaforo = CreateSemaphore(NULL, 1, 1, "Semaforo")) == NULL){
        printf("Falla al invocar CreateSemaphore: %d\n", GetLastError());
        return -1;
    }

    // Creación proceso hijo
    if(!CreateProcess(NULL, argv[1], NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)){
        printf("Falla al invocar CreateProcess: %d\n", GetLastError());
        return -1;
    }
```

```

while(i<4)

{
    // Prueba del semáforo

    WaitForSingleObject(hSemaphore, INFINITE);

    //Sección crítica

    printf("Soy el padre entrando %i de 3 veces al semaforo\n",i);

    Sleep(5000);

    //Liberación el semáforo

    if (!ReleaseSemaphore(hSemaphore, 1, NULL) ){

        printf("Falla al invocar ReleaseSemaphore: %d\n", GetLastError());

    }

    printf("Soy el padre liberando %i de 3 veces al semaforo\n",i);

    Sleep(5000);

    i++;

}

// Terminación controlada del proceso e hilo asociado de ejecución

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

}

```

## PROGRAMA HIJO

```

#include <windows.h> /*Programa hijo*/

#include <stdio.h>

int main()

```

```

{
HANDLE hSemaforo;
int i=1;

// Apertura del semáforo

if((hSemaforo = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "Semaforo")) ==NULL){
    printf("Falla al invocar OpenSemaphore: %d\n", GetLastError());
    return -1;
}

while(i<4){

    // Prueba del semáforo

    WaitForSingleObject(hSemaforo, INFINITE);

    //Sección crítica

    printf("Soy el hijo entrando %i de 3 veces al semaforo\n",i);
    Sleep(5000);

    //Liberación el semáforo

    if (!ReleaseSemaphore(hSemaforo, 1, NULL) ){
        printf("Falla al invocar ReleaseSemaphore: %d\n", GetLastError());
    }

    printf("Soy el hijo liberando %i de 3 veces al semaforo\n",i);
    Sleep(5000);

    i++;
}

}

```

## Salida

```
C:\Users\axelc\Documents\ESCOM\4TO SEMESTRE\SISTEMAS OPERATIVOS\PRACTICA8>padre.exe hijo.exe
Soy el padre entrando 1 de 3 veces al semáforo
Soy el padre liberando 1 de 3 veces al semáforo
Soy el hijo entrando 1 de 3 veces al semáforo
Soy el hijo liberando 1 de 3 veces al semáforo
Soy el padre entrando 2 de 3 veces al semáforo
Soy el padre liberando 2 de 3 veces al semáforo
Soy el hijo entrando 2 de 3 veces al semáforo
Soy el padre entrando 3 de 3 veces al semáforo
Soy el hijo liberando 2 de 3 veces al semáforo
Soy el padre liberando 3 de 3 veces al semáforo
Soy el hijo entrando 3 de 3 veces al semáforo
Soy el hijo liberando 3 de 3 veces al semáforo
```

## Observaciones

El programa padre inicializa los parámetros para crear un nuevo proceso e invoca al programa hijo, verifica si se ha proporcionado el programa hijo como argumento y luego crea un semáforo con un valor inicial y máximo de 1, luego el padre crea el proceso hijo utilizando la función CreateProcess(), una vez que el proceso hijo está en ejecución, el programa padre entra en un bucle donde intenta acceder a una sección crítica tres veces, también utiliza WaitForSingleObject() para esperar la señal del semáforo, para que pueda entrar en la sección crítica, realiza una operación simulada con Sleep() durante 5 segundos y luego libera el semáforo con ReleaseSemaphore() antes de dormir por otros 5 segundos y repetir el proceso.

Por su parte el programa hijo, abre el semáforo creado por el padre usando OpenSemaphore(), para después entrar en su propio bucle, similar al del padre, donde también intenta acceder a la sección crítica tres veces, utilizando WaitForSingleObject() para esperar la señal del semáforo antes de entrar en la sección crítica, donde permanece durante 5 segundos, luego libera el semáforo y duerme por otros 5 segundos antes de repetirlo. Ambos programas están diseñados para garantizar que solo uno de ellos pueda entrar en la sección crítica a la vez, sincronizando su acceso mediante el uso del semáforo.

**5. Programe la misma aplicación del punto 8 de la práctica 7, utilizando como máximo tres regiones de memoria compartida para almacenar todas las matrices requeridas por la aplicación. Utilice como mecanismo de sincronización los semáforos revisados en esta práctica tanto para la escritura y como para la lectura de las memorias compartidas. Úselos en los lugares donde haya necesidad de sincronizar el acceso a memoria compartida.**

## Linux

### Código

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <math.h>

// Función para calcular la matriz inversa
void calcularInversa(int A[10][10], double inverse[10][10]) {
    int n = 10;
    double aug[10][20]; // Matriz aumentada
    // Inicializar la matriz aumentada con la matriz A y la matriz identidad
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            aug[i][j] = A[i][j];
            aug[i][j + n] = (i == j) ? 1.0 : 0.0;
        }
    }
    // Aplicar eliminación de Gauss-Jordan
    for (int i = 0; i < n; i++) {
        // Verificar que el pivote no sea cero
        if (fabs(aug[i][i]) < 1e-9) {
            printf("La matriz no es invertible.\n");
            return; // No se puede invertir
        }
        // Hacer que el pivote sea 1
        double pivot = aug[i][i];
        for (int j = 0; j < 2 * n; j++) {
            aug[i][j] /= pivot;
        }
        // Hacer que todos los elementos de la columna i, excepto el pivote,
        sean cero
        for (int k = 0; k < n; k++) {
            if (k != i) {
                double factor = aug[k][i];
                for (int j = 0; j < 2 * n; j++) {
                    aug[k][j] -= factor * aug[i][j];
                }
            }
        }
    }
}

```

```

// Extraer la matriz inversa
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        inverse[i][j] = aug[i][j + n];
    }
}
}

// Función para guardar la matriz inversa en un archivo
void guardarMatrizEnArchivo(double matrix[10][10], const char *filename) {
FILE *file = fopen(filename, "w");
if (file == NULL) {
perror("Error abriendo el archivo");
exit(1);
}
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        fprintf(file, "%.2f ", matrix[i][j]);
    }
    fprintf(file, "\n");
}
fclose(file);
}

int main(void) {
int M1[10][10], M2[10][10];
int i, j;
int ResultadoHijo[10][10], ResultadoNieto[10][10];
key_t key = 1234, semkey = 5678;
int shmid, semid;
int (*shm)[10][10]; // Puntero a una matriz 10x10
size_t size = sizeof(int) * 200; // Tamaño suficiente para dos matrices
10x10
// Crear el segmento de memoria compartida
if ((shmid = shmget(key, size, IPC_CREAT | 0666)) < 0) {
perror("shmget");
exit(1);
}
// Adjuntar el segmento de memoria al espacio de direcciones del proceso
if ((shm = (int (*)[10][10])shmat(shmid, NULL, 0)) == (int (*)[10][10])-1) {
perror("shmat");
exit(1);
}
// Crear el conjunto de semáforos
if ((semid = semget(semkey, 1, IPC_CREAT | 0666)) == -1) {
perror("semget");
exit(1);
}
// Inicializar el semáforo
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *_buf;
} arg;
arg.val = 1;
if (semctl(semid, 0, SETVAL, arg) == -1) {
perror("semctl");
exit(1);
}
int pid1, pid2;
if ((pid1 = fork()) == 0) {
// Primer proceso hijo (multiplicación)
int M1RH[10][10];
int M2RH[10][10];
struct sembuf p1 = {0, -1, SEM_UNDO};
if (semop(semid, &p1, 1) == -1) {
perror("semop P1");
exit(1);
}
}

```

```

}
for (int i = 0; i < 10; i++) {
for (int j = 0; j < 10; j++) {
M1RH[i][j] = (*shm)[i][j];
}
}
struct sembuf v1 = {0, 1, SEM_UNDO};
if (semop(semid, &v1, 1) == -1) {
perror("semop V1");
exit(1);
}
struct sembuf p11 = {0, -1, SEM_UNDO};
if (semop(semid, &p11, 1) == -1) {
perror("semop P11");
exit(1);
}
for (int i = 0; i < 10; i++) {
for (int j = 0; j < 10; j++) {
M1RH[i][j] = (*shm)[i][j];
}
}
struct sembuf v11 = {0, 1, SEM_UNDO};
if (semop(semid, &v11, 1) == -1) {
perror("semop V11");
exit(1);
}
// Multiplicación de matrices M1 y M2
for (int i = 0; i < 10; i++) {
for (int j = 0; j < 10; j++) {
ResultadoHijo[i][j] = 0;
for (int k = 0; k < 10; k++) {
ResultadoHijo[i][j] += M1RH[i][k] * (*shm)[10 + k][j];
}
}
}
// Guardar el resultado en memoria compartida
struct sembuf p2 = {0, -1, SEM_UNDO};
if (semop(semid, &p2, 1) == -1) {
perror("semop P2");
exit(1);
}
for (int i = 0; i < 10; i++) {
for (int j = 0; j < 10; j++) {
(*shm)[i][j] = ResultadoHijo[i][j];
}
}
struct sembuf v2 = {0, 1, SEM_UNDO};
if (semop(semid, &v2, 1) == -1) {
perror("semop V2");
exit(1);
}
if ((pid2 = fork()) == 0) {
// Proceso nieto (suma)
int M2RN[10][10];
struct sembuf p3 = {0, -1, SEM_UNDO};
if (semop(semid, &p3, 1) == -1) {
perror("semop P3");
exit(1);
}
for (int i = 0; i < 10; i++) {
for (int j = 0; j < 10; j++) {
M2RN[i][j] = (*shm)[10 + i][j];
}
}
}

```

```
}

}

struct sembuf v3 = {0, 1, SEM_UNDO};
if (semop(semid, &v3, 1) == -1) {
    perror("semop V3");
exit(1);
}

// Suma de matrices M1 y M2
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        ResultadoNieto[i][j] = M2RN[i][j] + M1[i][j];
    }
}

// Guardar el resultado en memoria compartida
struct sembuf p4 = {0, -1, SEM_UNDO};
if (semop(semid, &p4, 1) == -1) {
    perror("semop P4");
exit(1);
}

for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        (*shm)[10 + i][j] = ResultadoNieto[i][j];
    }
}

struct sembuf v4 = {0, 1, SEM_UNDO};
if (semop(semid, &v4, 1) == -1) {
    perror("semop V4");
exit(1);
}

exit(0); // Terminar el proceso nieto
} else if (pid2 < 0) {
    perror("fork");
exit(1);
} else {
    wait(NULL); // Esperar a que el proceso nieto termine
    exit(0); // Terminar el primer proceso hijo
}
} else if (pid1 < 0) {
    perror("fork");
exit(1);
} else {
    // Se crean las matrices
```

```

// Se crean las matrices
for (i = 0; i < 10; ++i) {
    for (j = 0; j < 10; ++j) {
        M1[i][j] = 1 + (i * j);
    }
}
for (i = 0; i < 10; ++i) {
    for (j = 0; j < 10; ++j) {
        M2[i][j] = 1 + (i + j);
    }
}
struct sembuf p1 = {0, -1, SEM_UNDO};
if (semop(semid, &p1, 1) == -1) {
    perror("semop P1");
    exit(1);
}
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        (*shm)[i][j] = M1[i][j];
    }
}
struct sembuf v1 = {0, 1, SEM_UNDO};
if (semop(semid, &v1, 1) == -1) {
    perror("semop V1");
    exit(1);
}
struct sembuf p11 = {0, -1, SEM_UNDO};
if (semop(semid, &p11, 1) == -1) {
    perror("semop P11");
    exit(1);
}
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        (*shm)[i][j] = M2[i][j];
    }
}
struct sembuf v11 = {0, 1, SEM_UNDO};
if (semop(semid, &v11, 1) == -1) {
    perror("semop V11");
    exit(1);
}
wait(NULL); // Esperar a que el primer proceso hijo termine
// Imprimir la matriz resultado
printf("Matriz resultado:\n");
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        printf("%d ", (*shm)[i][j]);
    }
    printf("\n");
}
// Calcular la inversa de la matriz resultado
double inverse[10][10];
calcularInversa(*shm, inverse);
// Guardar la matriz inversa en un archivo
guardarMatrizEnArchivo(inverse, "matriz_inversa.txt");
// Desvincular y eliminar el segmento de memoria compartida
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
    perror("shmctl");
    exit(1);
}
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
    perror("shmctl");
    exit(1);
}
// Eliminar el conjunto de semáforos
if (semctl(semid, 0, IPC_RMID) == -1) {
    perror("semctl IPC_RMID");
    exit(1);
}
Salida
return 0;
}

```

MatrizInversa1.txt						MatrizInversa2.txt				
-0.27	0.59	0.08	-0.49	0.16	-0.03	0.17	0.53	-0.29	-0.40	
0.18	-0.45	-0.36	0.35	0.14	-0.20	-0.22	-0.23	0.56	0.20	
-0.05	0.16	-0.12	-0.10	0.09	0.01	-0.07	0.14	0.03	-0.09	
-0.02	0.00	0.04	0.04	-0.00	0.08	0.12	0.03	-0.13	-0.15	
0.10	-0.26	-0.09	0.29	0.03	-0.11	-0.17	-0.13	0.25	0.07	
-0.02	-0.05	0.01	-0.01	-0.08	0.04	0.00	0.03	0.02	0.08	
-0.00	0.00	0.14	-0.03	0.00	0.10	0.02	-0.11	-0.12	-0.00	
0.13	-0.16	0.02	0.16	-0.15	0.00	-0.06	-0.28	0.13	0.20	
-0.17	0.38	0.36	-0.33	-0.15	0.15	0.26	0.15	-0.58	-0.04	
0.12	-0.19	-0.06	0.10	-0.02	-0.03	-0.04	-0.11	0.11	0.12	
MatrizInversa1.txt						MatrizInversa2.txt				
0.24	-0.23	-0.18	-0.06	0.05	-0.14	-0.01	0.06	0.17	0.11	
-0.16	0.09	-0.18	0.07	0.21	-0.12	-0.10	0.09	0.25	-0.14	
0.04	0.02	-0.17	-0.02	0.07	-0.01	-0.10	0.06	0.11	-0.01	
0.02	-0.07	0.02	0.08	-0.01	0.07	0.10	-0.01	-0.09	-0.11	
-0.09	0.05	0.01	0.13	0.07	-0.07	-0.10	0.04	0.08	-0.12	
0.01	-0.10	-0.01	0.02	-0.09	0.03	-0.01	-0.00	0.04	0.11	
0.00	-0.02	0.14	-0.02	0.00	0.10	0.02	-0.12	-0.11	0.01	
-0.12	0.23	0.14	-0.05	-0.10	0.05	0.02	-0.05	-0.09	-0.04	
0.15	-0.15	0.19	-0.05	-0.22	0.07	0.15	-0.16	-0.28	0.29	
-0.11	0.17	0.05	-0.09	0.02	0.02	0.03	0.10	-0.10	-0.11	

## Observaciones

En este punto, el programa comienza creando una función para calcular la inversa de una matriz, que corresponde a la obtenida por los procesos hijo y nieto. Posteriormente, se desarrolla otra función encargada de guardar el resultado en un archivo, que servirá como salida del programa. Luego se inicia el flujo principal, donde se generan dos matrices que serán enviadas al proceso hijo y al proceso nieto. Estos procesos realizan operaciones específicas con las matrices y devuelven los resultados al proceso padre, que se encarga de calcular las inversas. Para facilitar la comunicación, se utiliza memoria compartida, permitiendo a los procesos almacenar y acceder a las matrices según lo definido por el programa. Además, se crea e inicializa un conjunto de semáforos para coordinar el acceso concurrente.

El primer proceso hijo recibe las matrices desde el padre y las envía al proceso nieto mediante la memoria compartida. Después, el hijo calcula la multiplicación de las dos matrices y guarda el resultado en la memoria compartida, para que el padre lo pueda leer. El proceso nieto, por su parte, recibe las matrices y calcula su suma, almacenando también el resultado en la memoria compartida. El proceso padre genera las matrices iniciales, envía los datos, recibe los resultados del hijo y del nieto, y calcula las inversas finales. Finalmente, se libera el segmento de memoria compartida y se eliminan los semáforos, los cuales son clave para garantizar que los procesos accedan de forma ordenada a los recursos compartidos.

**5. Programe la misma aplicación del punto 8 de la práctica 7, utilizando como máximo tres regiones de memoria compartida para almacenar todas las matrices requeridas por la aplicación. Utilice como mecanismo de sincronización los semáforos revisados en esta práctica tanto para la escritura y como para la lectura de las memorias compartidas. Úselos en los lugares donde haya necesidad de sincronizar el acceso a memoria compartida.**

### **Windows**

El programa ahora utiliza solo tres matrices en la memoria compartida: la primera y la segunda matriz original, y una tercera matriz que se usa tanto para los resultados de la multiplicación como de la suma. Se utilizan semáforos para sincronizar el acceso a la memoria compartida entre los procesos padre, hijo y nieto. El proceso padre inicia y genera las matrices originales, luego espera a que el hijo complete la multiplicación y el nieto la suma. Cada proceso usa semáforos para asegurar que las operaciones se realicen en el orden correcto y que no haya conflictos en el acceso a la memoria compartida.

### **PADRE**

El programa padre, crea matrices aleatorias, utiliza memoria compartida para coordinar operaciones entre procesos, y calcula las matrices inversas de resultados almacenados. Inicialmente, define constantes y funciones auxiliares para imprimir y calcular la inversa de matrices. Luego, establece una sección de memoria compartida para almacenar matrices y emplea semáforos para sincronizar la ejecución con un proceso hijo.

El proceso padre genera dos matrices aleatorias y lanza un proceso hijo (hijo.exe) que realiza operaciones (presumiblemente suma y multiplicación) sobre estas matrices. Después de que el hijo termina su tarea, el padre lee los resultados desde la memoria compartida, calcula las matrices inversas de estos resultados, y las imprime tanto en archivos como en pantalla.

La sincronización entre el padre y el hijo asegura que las operaciones se realicen en el orden correcto, utilizando semáforos para controlar el acceso a la memoria compartida y coordinar el flujo de trabajo entre los procesos. Al finalizar, el programa limpia los recursos utilizados, incluyendo la memoria compartida y los semáforos.

```
C ejercicio5_padre.c > ...
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <windows.h>
5
6 #define SIZE 10
7
8 void matrizInversa(double mat[SIZE][SIZE], double inv[SIZE][SIZE]) {
9     double mataumentada[SIZE][2 * SIZE];
10
11    for (int i = 0; i < SIZE; i++) {
12        for (int j = 0; j < SIZE; j++) {
13            mataumentada[i][j] = mat[i][j];
14            mataumentada[i][j + SIZE] = (i == j) ? 1 : 0;
15        }
16    }
17
18    for (int i = 0; i < SIZE; i++) {
19        double pivot = mataumentada[i][i];
20        if (pivot == 0) {
21            printf("La matriz no es invertible.\n");
22            exit(1);
23        }
24        for (int j = 0; j < 2 * SIZE; j++) {
25            mataumentada[i][j] /= pivot;
26        }
27        for (int k = 0; k < SIZE; k++) {
28            if (k != i) {
29                double factor = mataumentada[k][i];
30                for (int j = 0; j < 2 * SIZE; j++) {
31                    mataumentada[k][j] -= factor * mataumentada[i][j];
32                }
33            }
34        }
35    }
36
37    for (int i = 0; i < SIZE; i++) {
38        for (int j = 0; j < SIZE; j++) {
39            inv[i][j] = mataumentada[i][j + SIZE];
40        }
41    }
42 }
```

```
C ejercicio5_padre.c > ⌂ main()
8 void matrizInversa(double mat[SIZE][SIZE], double inv[SIZE][SIZE]) {
9
10    void imprimirMatriz(const char* nombre, double mat[SIZE][SIZE]) {
11        printf("\n%s:\n\n", nombre);
12        for (int i = 0; i < SIZE; i++) {
13            for (int j = 0; j < SIZE; j++) {
14                printf("%0.2f ", mat[i][j]);
15            }
16            printf("\n");
17        }
18        printf("\n");
19    }
20
21    int main() {
22        HANDLE hMapFile;
23        int (*matrices)[SIZE][SIZE];
24        const char *name = "MemoriaCompartida";
25        hMapFile = CreateFileMapping(
26            INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
27            3 * SIZE * SIZE * sizeof(int), name
28        );
29
30        if (hMapFile == NULL) {
31            printf(
32                "No se pudo crear la asignación de archivos (%d).\n",
33                GetLastError()
34            );
35            return 1;
36        }
37
38        matrices = (int (*)[SIZE][SIZE]) MapViewOfFile(
39            hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 3 * SIZE * SIZE * sizeof(int));
40
41        if (matrices == NULL) {
42            printf("No se pudo mapear el archivo (%d).\n", GetLastError());
43            CloseHandle(hMapFile);
44            return 1;
45        }
46
47        HANDLE semPadre = CreateSemaphore(NULL, 1, 1, "SemPadre");
48        HANDLE semHijo = CreateSemaphore(NULL, 0, 1, "SemHijo");
49        HANDLE semNieto = CreateSemaphore(NULL, 0, 1, "SemNieto");
50
51        if (semPadre == NULL || semHijo == NULL || semNieto == NULL) {
52            printf("No se pudieron crear los semáforos (%d).\n", GetLastError());
53            return 1;
54        }
55    }
56}
```

```
C ejercicio5_padre.c > main()
55  int main() {
85      if (semPadre == NULL || semHijo == NULL || semNieto == NULL) {
89
90          srand(time(NULL));
91          for (int i = 0; i < SIZE; i++) {
92              for (int j = 0; j < SIZE; j++) {
93                  matrices[0][i][j] = rand() % 10;
94                  matrices[1][i][j] = rand() % 10;
95              }
96          }
97
98          STARTUPINFO si;
99          PROCESS_INFORMATION pi;
100         ZeroMemory(&si, sizeof(si));
101         si.cb = sizeof(si);
102         ZeroMemory(&pi, sizeof(pi));
103
104         if (!CreateProcess(
105             NULL, "hijo.exe", NULL, NULL, FALSE, 0, NULL,
106             NULL, &si, &pi)
107         ) {
108             printf(
109                 "Fallo en la creación de proceso (%d).\n",
110                 GetLastError());
111             return 1;
112         }
113
114         // Liberar semáforo del hijo para que pueda iniciar
115         ReleaseSemaphore(semHijo, 1, NULL);
116
117         WaitForSingleObject(pi.hProcess, INFINITE);
118         CloseHandle(pi.hProcess);
119         CloseHandle(pi.hThread);
120
121         double inv_mult[SIZE][SIZE];
122         double inv_sum[SIZE][SIZE];
123         double mult[SIZE][SIZE];
124         double sum[SIZE][SIZE];
125
126         WaitForSingleObject(semPadre, INFINITE);
127
128         for (int i = 0; i < SIZE; i++) {
129             for (int j = 0; j < SIZE; j++) {
130                 mult[i][j] = matrices[2][i][j];
131                 sum[i][j] = matrices[2][i][j];
132             }
133         }
```

```

135     ReleaseSemaphore(semHijo, 1, NULL);
136
137     matrizInversa(mult, inv_mult);
138     matrizInversa(sum, inv_sum);
139
140     FILE *file_mult = fopen("Inversa_multiplicacion.txt", "w");
141     FILE *file_sum = fopen("Inversa_suma.txt", "w");
142
143     for (int i = 0; i < SIZE; i++) {
144         for (int j = 0; j < SIZE; j++) {
145             fprintf(file_mult, "%0.2f ", inv_mult[i][j]);
146             fprintf(file_sum, "%0.2f ", inv_sum[i][j]);
147         }
148         fprintf(file_mult, "\n");
149         fprintf(file_sum, "\n");
150     }
151
152     fclose(file_mult);
153     fclose(file_sum);
154
155     imprimirMatriz("Inversa de la multiplicacion", inv_mult);
156     imprimirMatriz("Inversa de la suma", inv_sum);
157
158     UnmapViewOfFile(matrices);
159     CloseHandle(hMapFile);
160     CloseHandle(semPadre);
161     CloseHandle(semHijo);
162     CloseHandle(semNieto);
163
164     return 0;
165 }
```

## Hijo

El programa abre la memoria compartida creada por el proceso padre para acceder a las matrices almacenadas. Primero, define constantes y funciones necesarias, incluyendo multiplicarMatrices, que realiza la multiplicación de dos matrices de tamaño fijo (10x10). Luego, el programa abre la memoria compartida con OpenFileMapping y mapea su contenido en su espacio de direcciones usando MapViewOfFile.

Después de establecer la conexión a la memoria compartida, el programa abre tres semáforos (semPadre, semHijo, y semNieto) para sincronizar su ejecución con el proceso padre y el proceso nieto. Utiliza WaitForSingleObject para esperar la señal del semáforo del hijo (semHijo) antes de proceder a multiplicar las dos matrices y almacenar el resultado en la tercera matriz dentro de la memoria compartida.

Posteriormente, el programa crea un proceso nieto (nieto.exe) para continuar con las operaciones necesarias. Se asegura de liberar el semáforo del nieto (semNieto) para permitir que este inicie su ejecución.

Finalmente, el programa hijo espera a que el proceso nieto termine, desmapea la vista de la memoria compartida y cierra todos los manejadores de archivos y semáforos abiertos, asegurando la limpieza adecuada de los recursos utilizados.

```
ejercicio5_hijo.c > ⌂ main()
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <windows.h>
4  #define SIZE 10
5
6  void multiplicarMatrices(int mat1[SIZE][SIZE], int mat2[SIZE][SIZE], int
7  res[SIZE][SIZE]) {
8      for (int i = 0; i < SIZE; i++) {
9          for (int j = 0; j < SIZE; j++) {
10              res[i][j] = 0;
11              for (int k = 0; k < SIZE; k++) {
12                  res[i][j] += mat1[i][k] * mat2[k][j];
13              }
14          }
15      }
16  }
17
18 int main() {
19     HANDLE hMapFile;
20     int (*matrices)[SIZE][SIZE];
21     const char *name = "MemoriaCompartida";
22     hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, name);
23
24     if (hMapFile == NULL) {
25         printf(
26             "No se pudo abrir la asignación de archivos (%d).\n",
27             GetLastError());
28         return 1;
29     }
30
31     matrices = (int (*)[SIZE][SIZE]) MapViewOfFile(hMapFile,
32             FILE_MAP_ALL_ACCESS, 0, 0, 3 * SIZE * SIZE * sizeof(int));
33
34     if (matrices == NULL) {
35         printf("No se pudo mapear el archivo (%d).\n", GetLastError());
36         CloseHandle(hMapFile);
37         return 1;
38     }
39
40     HANDLE semPadre = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE,
41             "SemPadre");
42     HANDLE semHijo = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE,
43             "SemHijo");
44     HANDLE semNieto = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE,
45             "SemNieto");
46 }
```

```

if (semPadre == NULL || semHijo == NULL || semNieto == NULL) {
    printf("No se pudieron abrir los semáforos (%d).\n", GetLastError());
    return 1;
}

WaitForSingleObject(semHijo, INFINITE);

multiplicarMatrices(matrices[0], matrices[1], matrices[2]);

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

if (
    !CreateProcess(
        NULL, "nieto.exe", NULL, NULL, FALSE, 0, NULL,
        NULL, &si, &pi)
) {
    printf("Fallo en la creacion de Proceso (%d).\n", GetLastError());
    return 1;
}

ReleaseSemaphore(semNieto, 1, NULL);

WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);

UnmapViewOfFile(matrices);
CloseHandle(hMapFile);
CloseHandle(semPadre);
CloseHandle(semHijo);
CloseHandle(semNieto);

return 0;
}

```

## NIETO

El programa nieto abre la memoria compartida previamente creada por el proceso padre para acceder a las matrices. Primero, define constantes y funciones necesarias, incluyendo sumarMatrices, que realiza la suma de dos matrices de tamaño fijo (10x10). Luego, el programa abre la memoria compartida con OpenFileMapping y mapea su contenido en su espacio de direcciones usando MapViewOfFile.

Después de establecer la conexión a la memoria compartida, el programa abre tres semáforos (semPadre, semHijo, y semNieto) para sincronizar su ejecución con los procesos padre e hijo. Utiliza WaitForSingleObject para esperar la señal del semáforo del nieto (semNieto) antes de

proceder a sumar las dos matrices y almacenar el resultado en la tercera matriz dentro de la memoria compartida.

Una vez realizada la suma de matrices, el programa libera el semáforo del padre (semPadre) para notificar que la operación ha sido completada.

```
C ejercicio5_nieto.c > main()
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <windows.h>
4
5  #define SIZE 10
6
7  void sumarMatrices(int mat1[SIZE][SIZE], int mat2[SIZE][SIZE], int
8  res[SIZE][SIZE]) {
9      for (int i = 0; i < SIZE; i++) {
10         for (int j = 0; j < SIZE; j++) {
11             res[i][j] = mat1[i][j] + mat2[i][j];
12         }
13     }
14 }
15
16 int main() {
17     HANDLE hMapFile;
18     int (*matrices)[SIZE][SIZE];
19     const char *name = "MemoriaCompartida";
20     hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, name);
21
22     if (hMapFile == NULL) {
23         printf("No se pudo abrir la asignación de archivos (%d).\n", GetLastError());
24         return 1;
25     }
26
27     matrices = (int (*)[SIZE][SIZE]) MapViewOfFile(hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 3 * SIZE * SIZE * sizeof(int));
28
29     if (matrices == NULL) {
30         printf("No se pudo mapear el archivo (%d).\n", GetLastError());
31         CloseHandle(hMapFile);
32         return 1;
33     }
34
35     HANDLE semPadre = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemPadre");
36     HANDLE semHijo = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemHijo");
37     HANDLE semNieto = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemNieto");
38
39     if (semPadre == NULL || semHijo == NULL || semNieto == NULL) {
40         printf("No se pudieron abrir los semáforos (%d).\n", GetLastError());
41         return 1;
42     }
43
44     WaitForSingleObject(semNieto, INFINITE);
45
46     sumarMatrices(matrices[0], matrices[1], matrices[2]);
47
48
49     sumarMatrices(matrices[0], matrices[1], matrices[2]);
50
51     ReleaseSemaphore(semPadre, 1, NULL);
52
53     UnmapViewOfFile(matrices);
54     CloseHandle(hMapFile);
55     CloseHandle(semPadre);
56     CloseHandle(semHijo);
57     CloseHandle(semNieto);
58     return 0;
59 }
```

Capturas de Pantalla.

**Inversa de la multiplicacion:**

```
-0.04 0.01 -0.05 -0.03 0.05 -0.03 -0.04 0.16 -0.06 -0.01  
-0.08 0.06 0.06 -0.08 0.04 0.00 0.00 0.09 -0.01 -0.08  
-0.02 -0.06 0.05 -0.04 0.03 0.01 0.18 -0.08 0.00 -0.01  
-0.04 -0.04 -0.03 -0.04 0.07 -0.00 -0.01 0.12 0.03 -0.06  
0.03 0.04 0.14 -0.06 -0.16 0.06 -0.13 -0.12 0.07 0.14  
-0.01 -0.04 -0.28 0.20 0.19 -0.15 0.03 0.18 -0.09 -0.09  
-0.05 0.01 0.10 -0.10 -0.11 0.09 0.17 -0.09 0.05 0.01  
0.07 0.00 0.01 0.02 -0.04 -0.04 -0.14 0.03 0.02 0.04  
0.08 0.06 -0.08 -0.02 0.06 -0.08 -0.01 0.00 -0.01 -0.02  
0.07 -0.05 0.06 0.17 -0.11 0.12 -0.05 -0.26 0.02 0.09
```

**Inversa de la suma:**

```
-0.04 0.01 -0.05 -0.03 0.05 -0.03 -0.04 0.16 -0.06 -0.01  
-0.08 0.06 0.06 -0.08 0.04 0.00 0.00 0.09 -0.01 -0.08  
-0.02 -0.06 0.05 -0.04 0.03 0.01 0.18 -0.08 0.00 -0.01  
-0.04 -0.04 -0.03 -0.04 0.07 -0.00 -0.01 0.12 0.03 -0.06  
0.03 0.04 0.14 -0.06 -0.16 0.06 -0.13 -0.12 0.07 0.14  
-0.01 -0.04 -0.28 0.20 0.19 -0.15 0.03 0.18 -0.09 -0.09  
-0.05 0.01 0.10 -0.10 -0.11 0.09 0.17 -0.09 0.05 0.01  
0.07 0.00 0.01 0.02 -0.04 -0.04 -0.14 0.03 0.02 0.04  
0.08 0.06 -0.08 -0.02 0.06 -0.08 -0.01 0.00 -0.01 -0.02  
0.07 -0.05 0.06 0.17 -0.11 0.12 -0.05 -0.26 0.02 0.09
```

6. Programe la misma aplicación del punto 9 de la práctica 7 en la versión de memoria compartida, pero en esta ocasión sólo se usarán tres regiones de memoria compartidas para almacenar todas las matrices requeridas por la aplicación. Utilice como mecanismo de sincronización los semáforos revisados en esta práctica tanto para la escritura y como para la lectura de las memorias compartidas. Úselos en los lugares donde haya necesidad de sincronizar el acceso a memoria compartida.

## Linux

En la versión de Linux, se reinterpretó el punto 9 correspondiente a la práctica 7 para implementar el uso de semáforos. La modificación no fue extensa, ya que el programa utiliza una sola región de memoria compartida. La nueva lógica establece que el cliente encargado de leer los resultados debe esperar hasta que los otros cinco clientes hayan completado su ejecución.

Anteriormente, con el uso exclusivo de memoria compartida, los clientes podían ejecutarse en cualquier orden, lo que permitía que el cliente de lectura se activara en cualquier momento. Sin embargo, esto no era ideal. Por ello, se añadieron dos semáforos: uno para controlar las ejecuciones de los cinco clientes que realizan operaciones con matrices y otro para liberar al cliente de lectura una vez que los cinco clientes han terminado su ejecución. Con este mecanismo,

se garantiza que el cliente de lectura solo se active después de que los resultados estén listos para ser mostrados.

```
// SERVIDOR DE LA MEMORIA COMPARTIDA
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#include <sys/sem.h>

// Declaración de la función
void CrearMemoriaCompartida(int a[10][10], int b[10][10], size_t tam);

#define TAM_MEM 806 // Tamaño de la memoria compartida

int main()
{
    int a[10][10], b[10][10];
    int i, j;
    key_t llavesem = 4321;

    srand(time(NULL));

    // Llenar matrices A y B con valores aleatorios
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 10; j++)
        {
            a[i][j] = rand() % 101; // valores entre 0 y 100
            b[i][j] = rand() % 101;
        }
    }

    // Imprimir la matriz A
    printf("\nLa matriz A es:\n");
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 10; j++)
        {
            printf("%d\t", a[i][j]);
        }
        printf("\n");
    }

    // Imprimir la matriz B
    printf("\nLa matriz B es:\n");
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 10; j++)
        {
            printf("%d\t", b[i][j]);
        }
        printf("\n");
    }
}
```

```
// Crear memoria compartida y almacenar matrices
CrearMemoriaCompartida(a, b, sizeof(a));

// Obtener semid después de CrearMemoriaCompartida
int semid = semget(llavesem, 2, 0);
if (semid == -1) {
    perror("Error al obtener semáforo");
    exit(1);
}

printf("\nEsperando a que los clientes terminen sus operaciones...\n");
fflush(stdout);

// Ya no necesitamos esperar aquí, el lector se encargará de eso
return 0;
}

/**
 * CrearMemoriaCompartida:
 * Crea la memoria compartida, la vincula al proceso, copia las dos
 * matrices (A, B) y crea e inicializa los semáforos.
 */
void CrearMemoriaCompartida(int a[10][10], int b[10][10], size_t tam)
{
    int shmid;
    key_t llave = 5678;
    void *shm;

    // Variables del semáforo
    int semid;
    key_t llavesem = 4321;

    // Crear la memoria compartida
    if ((shmid = shmget(llave, TAM_MEM, IPC_CREAT | 0666)) < 0)
    {
        perror("Error al obtener memoria compartida: shmget");
        exit(1);
    }

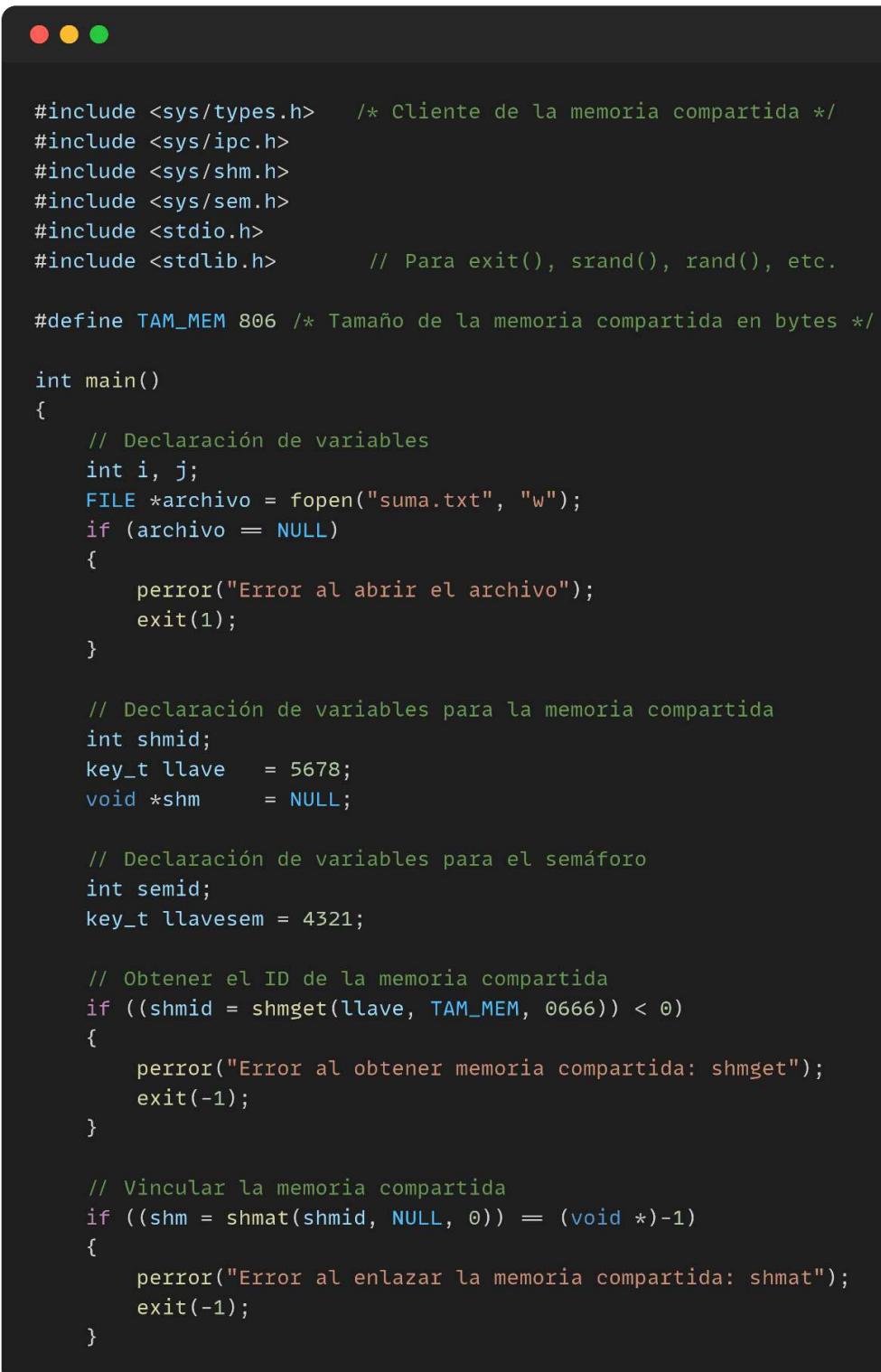
    // Vincular la memoria compartida
    if ((shm = shmat(shmid, NULL, 0)) == (void *)-1)
    {
        perror("Error al enlazar la memoria compartida: shmat");
        exit(1);
    }

    // Copiar la matriz A y luego B a la zona de memoria
    memcpy(shm, a, tam);
    memcpy((char *)shm + tam, b, tam);

    // Crear/obtener el conjunto de semáforos (dos en total)
    if ((semid = semget(llavesem, 2, IPC_CREAT | 0666)) == -1)
    {
        perror("\nError al crear los semáforos");
        exit(1);
    }

    // Inicializar semáforos
    // semáforo[0] = 0 → para los 5 clientes de operación
    if (semctl(semid, 0, SETVAL, 0) == -1)
    {
        perror("\nError en la inicialización del semáforo de operaciones: semctl");
    }
}
```

## SUMA



```
#include <sys/types.h>      /* Cliente de la memoria compartida */
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>          // Para exit(), srand(), rand(), etc.

#define TAM_MEM 806 /* Tamaño de la memoria compartida en bytes */

int main()
{
    // Declaración de variables
    int i, j;
    FILE *archivo = fopen("suma.txt", "w");
    if (archivo == NULL)
    {
        perror("Error al abrir el archivo");
        exit(1);
    }

    // Declaración de variables para la memoria compartida
    int shmid;
    key_t llave    = 5678;
    void *shm      = NULL;

    // Declaración de variables para el semáforo
    int semid;
    key_t llavesem = 4321;

    // Obtener el ID de la memoria compartida
    if ((shmid = shmget(llave, TAM_MEM, 0666)) < 0)
    {
        perror("Error al obtener memoria compartida: shmget");
        exit(-1);
    }

    // Vincular la memoria compartida
    if ((shm = shmat(shmid, NULL, 0)) == (void *)-1)
    {
        perror("Error al enlazar la memoria compartida: shmat");
        exit(-1);
    }
}
```

```
// Obtener acceso a los semáforos
if ((semid = semget(llavesem, 2, 0666)) == -1)
{
    perror("Error al obtener el semáforo de operaciones: semget");
    exit(1);
}

// Apuntadores a las matrices A y B dentro de la memoria compartida
// Aquí asumimos que la primera parte de la memoria corresponde a A
// y que inmediatamente después está la memoria para B.
int (*a)[10] = (int (*)[10])(shm);
int (*b)[10] = (int (*)[10])((char *)shm + sizeof(int [10][10]));

// Escribir la suma de A y B en el archivo
fprintf(archivo, "\nEl resultado de la suma A+B es:\n");
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        fprintf(archivo, "%d\t", a[i][j] + b[i][j]);
    }
    fprintf(archivo, "\n");
}

fclose(archivo);

// Incrementar el semáforo para indicar que este cliente ha terminado
struct sembuf sb;
sb.sem_num = 0;
sb.sem_op = 1; // Incrementar (liberar)
sb.sem_flg = 0;

if (semop(semid, &sb, 1) == -1)
{
    perror("Error al incrementar el semáforo: semop");
    exit(1);
}

return 0;
}
```

## RESTA

```
// CLIENTE QUE HACE LA RESTA DE MATRICES
#include <sys/types.h> /* Cliente de la memoria compartida */
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h> // Para exit() y perror()

#define TAM_MEM 806 /* Tamaño de la memoria compartida en bytes */

int main()
{
    // Declaración de variables
    int i, j;
    FILE *archivo = fopen("resta.txt", "w");
    if (archivo == NULL)
    {
        perror("Error al abrir el archivo");
        exit(1);
    }

    // Declaración de variables para la memoria compartida
    int shmid;
    key_t llave = 5678;
    void *shm = NULL;

    // Declaración de variables para el semáforo
    int semid;
    key_t llavesem = 4321;

    // Obtener el ID de la memoria compartida
    if ((shmid = shmget(llave, TAM_MEM, 0666)) < 0)
    {
        perror("Error al obtener memoria compartida: shmget");
        exit(-1);
    }

    // Vincular la memoria compartida
    if ((shm = shmat(shmid, NULL, 0)) == (void *)-1)
    {
        perror("Error al enlazar la memoria compartida: shmat");
        exit(-1);
    }
```

```
// Obtener acceso a los semáforos
if ((semid = semget(llavesem, 2, 0666)) == -1)
{
    perror("Error al obtener el semáforo de operaciones: semget");
    exit(1);
}

// Apuntadores a las matrices A y B dentro de la memoria compartida
int (*a)[10] = (int (*)[10])(shm);
int (*b)[10] = (int (*)[10])((char *)shm + sizeof(int [10][10]));

// Escribir la resta de A y B en el archivo
fprintf(archivo, "\nEl resultado de la resta A-B es:\n");
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        fprintf(archivo, "%d\t", a[i][j] - b[i][j]);
    }
    fprintf(archivo, "\n");
}

fclose(archivo);

// Incrementar el semáforo para indicar que este cliente ha terminado
struct sembuf sb;
sb.sem_num = 0; // Asumiendo que el semáforo[0] gestiona operaciones
sb.sem_op = 1; // Incrementar (liberar)
sb.sem_flg = 0;

if (semop(semid, &sb, 1) == -1)
{
    perror("Error al incrementar el semáforo: semop");
    exit(1);
}

return 0;
}
```

## PRODUCTO

```
// CLIENTE QUE HACE LA MULTIPLICACIÓN DE MATRICES
#include <sys/types.h>    /* Cliente de la memoria compartida */
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>          // Para exit() y perror()

#define TAM_MEM 806 /* Tamaño de la memoria compartida en bytes */

int main()
{
    // Declaración de variables
    int i, j, k;
    int mr[10][10];

    FILE *archivo = fopen("multiplicacion.txt", "w");
    if (archivo == NULL)
    {
        perror("Error al abrir el archivo");
        exit(1);
    }

    // Declaración de variables para la memoria compartida
    int shmid;
    key_t llave = 5678;
    void *shm = NULL;

    // Declaración de variables para el semáforo
    int semid;
    key_t llavesem = 4321;

    // Obtener el ID de la memoria compartida
    if ((shmid = shmget(llave, TAM_MEM, 0666)) < 0)
    {
        perror("Error al obtener memoria compartida: shmget");
        exit(-1);
    }

    // Vincular la memoria compartida
    if ((shm = shmat(shmid, NULL, 0)) == (void *)-1)
    {
        perror("Error al enlazar la memoria compartida: shmat");
        exit(-1);
    }
```

```
// Obtener el ID de los semáforos
if ((semid = semget(llavesem, 2, 0666)) == -1)
{
    perror("Error al obtener el semáforo de operaciones: semget");
    exit(1);
}

// Apuntadores a las matrices A y B dentro de la memoria compartida
int (*a)[10] = (int (*)[10])(shm);
int (*b)[10] = (int (*)[10])((char *)shm + sizeof(int[10][10]));

// Escribir la multiplicación de A y B en el archivo
fprintf(archivo, "\nEl resultado de la multiplicación A*B es:\n");
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        mr[i][j] = 0; // Inicializar en 0 para acumular la suma
        for (k = 0; k < 10; k++)
        {
            mr[i][j] += a[i][k] * b[k][j]; // Realiza la multiplicación
        }
        fprintf(archivo, "%d\t", mr[i][j]);
    }
    fprintf(archivo, "\n");
}

fclose(archivo);

// Incrementar el semáforo para indicar que este cliente ha terminado
struct sembuf sb;
sb.sem_num = 0; // Asumiendo que semáforo[0] controla las operaciones
sb.sem_op = 1; // Incrementar (liberar)
sb.sem_flg = 0;

if (semop(semid, &sb, 1) == -1)
{
    perror("Error al incrementar el semáforo: semop");
    exit(1);
}

return 0;
}
```

## TRANSPUESTA

```
// CLIENTE QUE HACE LA TRANSPUESTA DE MATRICES
#include <sys/types.h> /* Cliente de la memoria compartida */
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h> // Para exit() y perror()

#define TAM_MEM 806 /* Tamaño de la memoria compartida en bytes */

int main()
{
    // Declaración de variables
    int i, j;
    FILE *archivo = fopen("transpuesta.txt", "w");
    if (archivo == NULL)
    {
        perror("Error al abrir el archivo");
        exit(1);
    }

    // Declaración de variables para la memoria compartida
    int shmid;
    key_t llave = 5678;
    void *shm = NULL;

    // Declaración de variables para el semáforo
    int semid;
    key_t llavesem = 4321;

    // Obtener el ID de la memoria compartida
    if ((shmid = shmget(llave, TAM_MEM, 0666)) < 0)
    {
        perror("Error al obtener memoria compartida: shmget");
        exit(-1);
    }

    // Vincular la memoria compartida
    if ((shm = shmat(shmid, NULL, 0)) == (void *)-1)
    {
        perror("Error al enlazar la memoria compartida: shmat");
        exit(-1);
    }
```

```
// Obtener acceso a los semáforos
if ((semid = semget(llavesem, 2, 0666)) == -1)
{
    perror("Error al obtener el semáforo de operaciones: semget");
    exit(1);
}

// Apuntadores a las matrices A y B dentro de la memoria compartida
int (*a)[10] = (int (*)[10])shm;
int (*b)[10] = (int (*)[10])((char *)shm + sizeof(int[10][10]));

// Imprimir la transpuesta de la matriz A en el archivo
fprintf(archivo, "\nLa transpuesta de la matriz A es:\n");
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        fprintf(archivo, "%d\t", a[j][i]);
    }
    fprintf(archivo, "\n");
}

// Imprimir la transpuesta de la matriz B en el archivo
fprintf(archivo, "\nLa transpuesta de la matriz B es:\n");
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        fprintf(archivo, "%d\t", b[j][i]);
    }
    fprintf(archivo, "\n");
}

fclose(archivo);

// Incrementar el semáforo para indicar que este cliente ha terminado
struct sembuf sb;
sb.sem_num = 0;
sb.sem_op = 1; // Incrementar
sb.sem_flg = 0;

if (semop(semid, &sb, 1) == -1)
{
    perror("Error al incrementar el semáforo: semop");
    exit(1);
}

return 0;
}
```

## INVERSA

```
// CLIENTE QUE HACE LA INVERSA DE MATRICES
#include <sys/types.h> /* Cliente de la memoria compartida */
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>      // Para exit() y perror()

#define TAM_MEM 806 /* Tamaño de la memoria compartida en bytes */

int main()
{
    // Declaración de variables
    int i, j, k;
    FILE *archivo = fopen("inversa.txt", "w");
    if (archivo == NULL)
    {
        perror("Error al abrir el archivo");
        exit(1);
    }

    // Matrices para el cálculo
    double m1[10][10], m2[10][10];
    double id[10][10], id2[10][10];
    double pivote, pivote2, aux, aux2;

    // Declaración de variables para la memoria compartida
    int shmid;
    key_t llave = 5678;
    void *shm = NULL;

    // Declaración de variables para el semáforo
    int semid;
    key_t llavesem = 4321;

    // Obtener el ID de la memoria compartida
    if ((shmid = shmget(llave, TAM_MEM, 0666)) < 0)
    {
        perror("Error al obtener memoria compartida: shmget");
        exit(-1);
    }

    // Vincular la memoria compartida
    if ((shm = shmat(shmid, NULL, 0)) == (void *)-1)
    {
        perror("Error al enlazar la memoria compartida: shmat");
        exit(-1);
    }
```

```
// Obtener acceso a los semáforos
if ((semid = semget(llavesem, 2, 0666)) == -1)
{
    perror("Error al obtener el semáforo de operaciones: semget");
    exit(1);
}

// Apuntadores a las matrices A y B dentro de la memoria compartida
int (*a)[10] = (int (*)[10])(shm);
int (*b)[10] = (int (*)[10])((char *)shm + sizeof(int[10][10]));

// Convertir las matrices a tipo double y llenar las matrices identidad
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        m1[i][j] = a[i][j];
        m2[i][j] = b[i][j];
        id[i][j] = 0.0;
        id2[i][j] = 0.0;

        if (i == j)
        {
            id[i][j] = 1.0;
            id2[i][j] = 1.0;
        }
    }
}

// Obtener la inversa de las dos matrices (m1 e id) y (m2 e id2) simultáneamente
for (i = 0; i < 10; i++)
{
    pivot = m1[i][i];
    pivot2 = m2[i][i];

    // Hacer que el pivote sea 1 en ambas matrices
    for (j = 0; j < 10; j++)
    {
        m1[i][j] /= pivot;
        id[i][j] /= pivot;

        m2[i][j] /= pivot2;
        id2[i][j] /= pivot2;
    }

    // Hacer ceros en la columna i para todas las filas k ≠ i
    for (k = 0; k < 10; k++)
    {
        if (k != i)
        {
            aux = m1[k][i];
            aux2 = m2[k][i];
            m1[k][i] = 0.0;
            m2[k][i] = 0.0;
            id[k][i] = 0.0;
            id2[k][i] = 0.0;
        }
    }
}
```

```
    for (j = 0; j < 10; j++)
    {
        m1[k][j] = m1[k][j] - aux * m1[i][j];
        id[k][j] = id[k][j] - aux * id[i][j];

        m2[k][j] = m2[k][j] - aux2 * m2[i][j];
        id2[k][j] = id2[k][j] - aux2 * id2[i][j];
    }
}

// Imprimir la inversa de la matriz A en el archivo
fprintf(archivo, "\nLa inversa de la matriz A es:\n");
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        fprintf(archivo, "% .2f\t", id[i][j]);
    }
    fprintf(archivo, "\n");
}

// Imprimir la inversa de la matriz B en el archivo
fprintf(archivo, "\nLa inversa de la matriz B es:\n");
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        fprintf(archivo, "% .2f\t", id2[i][j]);
    }
    fprintf(archivo, "\n");
}

fclose(archivo);

// Incrementar el semáforo para indicar que este cliente ha terminado
struct sembuf sb;
sb.sem_num = 0;
sb.sem_op = 1; // Incrementar
sb.sem_flg = 0;

if (semop(semid, &sb, 1) == -1)
{
    perror("Error al incrementar el semáforo: semop");
    exit(1);
}

return 0;
}
```

## LEER AL CLIENTE

```
// CLIENTE QUE LEE LOS RESULTADOS DE LAS MATRICES
#include <sys/types.h>    /* Cliente de la memoria compartida */
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/sem.h>
#include <string.h>

#define TAM_MEM 806 /* Tamaño de la memoria compartida en bytes */

int main()
{
    setbuf(stdout, NULL); // Deshabilitar buffering de stdout

    // Declaración de variables para la lectura de los archivos
    char *ruta1, *ruta2, *ruta3, *ruta4, *ruta5, buffer[500];
    int archivo;

    // Declaración de variables para la memoria compartida
    int shmid;
    key_t llave = 5678;
    void *shm = NULL;

    // Declaración de variables para el semáforo
    int semid;
    key_t llavesem = 4321;

    // Obtener el ID de la memoria compartida
    if ((shmid = shmget(llave, TAM_MEM, 0666)) < 0)
    {
        perror("Error al obtener memoria compartida: shmget");
        exit(-1);
    }

    // Vincular la memoria compartida
    if ((shm = shmat(shmid, NULL, 0)) == (void *)-1)
    {
        perror("Error al enlazar la memoria compartida: shmat");
        exit(-1);
    }
```

```
// Obtener el semáforo
if ((semid = semget(llavesem, 2, 0666)) == -1)
{
    perror("Error al obtener el semáforo: semget");
    exit(1);
}

printf("\n--- Estado de los semáforos ---\n");
printf("Semáforo 0 (operaciones): %d\n", semctl(semid, 0, GETVAL));
printf("Semáforo 1 (lectura): %d\n", semctl(semid, 1, GETVAL));
printf("Esperando a que los clientes terminen...\n");
fflush(stdout);

int valor_actual;
while ((valor_actual = semctl(semid, 0, GETVAL)) < 5) {
    printf("Valor actual del semáforo: %d/5\n", valor_actual);
    sleep(1);
}

printf("Todos los clientes han terminado. Comenzando lectura...\n");
fflush(stdout);

// Rutas de los 5 archivos generados por los otros clientes
ruta1 = "suma.txt";
ruta2 = "resta.txt";
ruta3 = "multiplicacion.txt";
ruta4 = "transpuesta.txt";
ruta5 = "inversa.txt";

// Antes de comenzar las lecturas
printf("Verificando archivos:\n");
printf("suma.txt: %s\n", access("suma.txt", F_OK) != -1 ? "existe" : "no existe");
printf("resta.txt: %s\n", access("resta.txt", F_OK) != -1 ? "existe" : "no existe");
printf("multiplicacion.txt: %s\n", access("multiplicacion.txt", F_OK) != -1 ? "existe" : "no existe");
printf("transpuesta.txt: %s\n", access("transpuesta.txt", F_OK) != -1 ? "existe" : "no existe");
printf("inversa.txt: %s\n", access("inversa.txt", F_OK) != -1 ? "existe" : "no existe");
fflush(stdout);

// --- Lectura del archivo de SUMA ---
archivo = open(ruta1, O_RDONLY);
if (archivo != -1)
{
    printf("\nArchivo '%s' abierto exitosamente.\n", ruta1);
    ssize_t bytesleidos;
    while ((bytesleidos = read(archivo, buffer, sizeof(buffer))) > 0)
    {
        printf("%..*s", (int)bytesleidos, buffer);
        fflush(stdout);
    }
    if (bytesleidos == -1)
        perror("\nError al leer el archivo");
    close(archivo);
}
else
{
    perror("\nNo se pudo abrir el archivo de suma");
}
```

```
// --- Lectura del archivo de RESTA ---
archivo = open(ruta2, O_RDONLY);
if (archivo != -1)
{
    printf("\nArchivo '%s' abierto exitosamente.\n", ruta2);
    ssize_t bytesleidos;
    while ((bytesleidos = read(archivo, buffer, sizeof(buffer))) > 0)
    {
        printf("%.*s", (int)bytesleidos, buffer);
        fflush(stdout);
    }
    if (bytesleidos == -1)
        perror("\nError al leer el archivo");
    close(archivo);
}
else
{
    perror("\nNo se pudo abrir el archivo de resta");
}

// --- Lectura del archivo de MULTIPLICACIÓN ---
archivo = open(ruta3, O_RDONLY);
if (archivo != -1)
{
    printf("\nArchivo '%s' abierto exitosamente.\n", ruta3);
    ssize_t bytesleidos;
    while ((bytesleidos = read(archivo, buffer, sizeof(buffer))) > 0)
    {
        printf("%.*s", (int)bytesleidos, buffer);
        fflush(stdout);
    }
    if (bytesleidos == -1)
        perror("\nError al leer el archivo");
    close(archivo);
}
else
{
    perror("\nNo se pudo abrir el archivo de multiplicación");
}
```

```
// --- Lectura del archivo de TRANSPUESTA ---
archivo = open(ruta4, O_RDONLY);
if (archivo != -1)
{
    printf("\nArchivo '%s' abierto exitosamente.\n", ruta4);
    ssize_t bytesleidos;
    while ((bytesleidos = read(archivo, buffer, sizeof(buffer))) > 0)
    {
        printf("%..*s", (int)bytesleidos, buffer);
        fflush(stdout);
    }
    if (bytesleidos == -1)
        perror("\nError al leer el archivo");
    close(archivo);
}
else
{
    perror("\nNo se pudo abrir el archivo de transpuesta");
}

// --- Lectura del archivo de INVERSA ---
archivo = open(ruta5, O_RDONLY);
if (archivo != -1)
{
    printf("\nArchivo '%s' abierto exitosamente.\n", ruta5);
    ssize_t bytesleidos;
    while ((bytesleidos = read(archivo, buffer, sizeof(buffer))) > 0)
    {
        printf("%..*s", (int)bytesleidos, buffer);
        fflush(stdout);
    }
    if (bytesleidos == -1)
        perror("\nError al leer el archivo");
    close(archivo);
}
else
{
    perror("\nNo se pudo abrir el archivo de inversa");
}

// Incrementar el semáforo para notificar que el cliente de lectura terminó
struct sembuf sbl;
sbl.sem_num = 1;
sbl.sem_op  = 1;
sbl.sem_flg = 0;
if (semop(semid, &sbl, 1) == -1)
{
    perror("\nError en la liberación del semáforo");
    exit(1);
}

return 0;
}
```

## Salida

```
> ls
@ inv.c @ main.c @ minus.c @ plus.c @ prod.c @ read.c @ trans.c
> gcc main.c -o servidor
gcc plus.c -o suma
gcc minus.c -o resta
gcc prod.c -o multiplicacion
gcc trans.c -o transpuesta
gcc inv.c -o inversa
gcc read.c -o lector
> ls
@ inv.c lector @ minus.c      @ plus.c @ read.c @ servidor @ trans.c
@ inversa @ main.c @multiplicacion @ prod.c @resta @suma @transpuesta
> ./servidor &
[1] 331999

La matriz A es:
17   12   30   61   17   70   11   59   43   30
91   45   24    8   63   91   95   78   79    7
80   78   95   21   65   82   73   75   33   42
68    8   33   53   23    7   3   68   71   68
98   89   76   99    5   37   4   34   64   16
13   89   33   80   96   38   63   29   90   80
8   33   95    9   87   10   81   48   81   84
41    1   47    8   92   64   71   63   98   98
29   16   32   90   58   20   68   35   86   77
98   34   78   100   68   17   22   74   29   32

La matriz B es:
69   32   44   17    9    7   18   64   37   12
22   54    6   80   26   46   69   24    4   96
77   58   28    3   51   21   27   63   76   62
99   62   52   12   68    2   43   98   81    1
49   78   64   76   48   73   11   38   53   87
98   16   15   70   47   94   26   38   95   95
17    5   34   58   25   66   66   13   28   42
21    3   98    3   88   92   98   66   38   50
8   10   64   79   73   66   89   74   43   77
6   34   64   67   56   98   100   54   63   88

Esperando a que los clientes terminen sus operaciones...
[1] + done      /servidor
> ./suma &
./resta &
./multiplicacion &
./transpuesta &
./inversa &
[1] 332044
[2] 332045
[3] 332046
[4] 332047
[5] 332048
[1] done      ./suma
[2] done      ./resta
[3] done      ./multiplicacion
[4] - done     ./transpuesta
[5] + done     ./inversa
~/SO > |
```

```

> ls
❷ inv.c      ❷ main.c          ❷ plus.c   ❶ resta.txt ❷ trans.c
❸ inversa    ❷ minus.c        ❷ prod.c    ❸ servidor ❸ transpuesta
❶ inversa.txt ❸ multiplicacion ❷ read.c    ❷ suma      ❶ transpuesta.txt
❸ lector     ❶ multiplicacion.txt ❸ resta     ❶ suma.txt

> ./lector

--- Estado de los semáforos ---
Semáforo 0 (operaciones): 5
Semáforo 1 (lectura): 0
Esperando a que los clientes terminen...
Todos los clientes han terminado. Comenzando lectura...
Verificando archivos:
suma.txt: existe
resta.txt: existe
multiplicacion.txt: existe
transpuesta.txt: existe
inversa.txt: existe

Archivo 'suma.txt' abierto exitosamente.

El resultado de la suma A+B es:
 86   44   74   78   26   77   29   123   80   42
 113   99   30   80   89   137   164   102   83   103
 157   128   123   24   116   103   100   138   109   104
 167   70   85   65   83   9   46   158   152   69
 147   159   140   175   45   110   15   72   117   103
 103   105   48   150   143   132   89   67   185   175
 17   38   129   67   112   76   147   61   109   126
 62   4   145   11   180   156   169   129   136   140
 29   26   96   169   131   86   149   109   129   154
 104   68   134   167   124   115   122   128   92   120

Archivo 'resta.txt' abierto exitosamente.

El resultado de la resta A-B es:
 -52   -20   -14   44   8   63   -7   -5   6   18
 69   -9   18   -80   37   45   26   54   75   -89
 3   12   67   18   14   61   46   12   -43   -20
 -31   -54   -19   41   -37   5   -40   -38   -10   67
 49   19   12   23   -35   -36   -7   -4   11   -71
 -77   73   18   10   49   -56   37   -9   -5   -15
 -17   28   61   -49   62   -56   15   35   53   42
 20   -2   -51   5   4   -28   -27   -3   60   40
 29   6   -32   11   -15   -46   -29   -39   43   0
 92   0   6   33   12   -81   -78   20   -34   -56

Archivo 'multiplicacion.txt' abierto exitosamente.

```

El resultado de la multiplicación A\*B es:

18525	10706	17798	14485	19911	21176	19909	21389	21738	20769
23689	14337	26721	28831	25408	35710	30095	26875	26817	35615
30087	21362	27714	28057	28285	36282	31890	31038	32355	40089
16132	12747	23175	15288	21278	21048	24578	25882	20934	20926
28826	20736	21585	19634	23611	19601	26123	32390	26721	26685
23600	23446	28768	34567	29717	36224	35760	31692	30299	40854
16984	18315	27062	27214	27059	34162	32105	26096	26509	37017
20600	16636	31010	30565	29473	39838	33678	29514	30963	39192
20586	17491	26862	24295	26139	28307	30262	29652	27392	28572
29782	21974	28187	18087	25826	23792	25975	33572	29092	26084

Archivo 'transpuesta.txt' abierto exitosamente.

La transpuesta de la matriz A es:

17	91	80	68	98	13	0	41	29	98
12	45	70	8	89	89	33	1	16	34
30	24	95	33	76	33	95	47	32	70
61	0	21	53	99	80	9	8	90	100
17	63	65	23	5	96	87	92	58	68
70	91	82	7	37	38	10	64	20	17
11	95	73	3	4	63	81	71	60	22
59	78	75	60	34	29	48	63	35	74
43	79	33	71	64	90	81	98	86	29
30	7	42	68	16	80	84	90	77	32

La transpuesta de la matriz B es:

69	22	77	99	49	90	17	21	0	6
32	54	58	62	70	16	5	3	10	34
44	6	28	52	64	15	34	98	64	64
17	80	3	12	76	70	58	3	79	67
9	26	51	60	40	47	25	88	73	56
7	46	21	2	73	94	66	92	66	98
18	69	27	43	11	26	66	98	89	100
64	24	63	98	38	38	13	66	74	54
37	4	76	81	53	95	28	38	43	63
12	96	62	1	87	95	42	50	77	88

Archivo 'inversa.txt' abierto exitosamente.

La inversa de la matriz A es:

-0.01	-0.00	0.00	0.00	0.00	-0.00	-0.01	0.01	0.00	0.00
-0.00	0.00	0.00	0.01	-0.00	0.01	0.00	-0.01	-0.01	-0.00
0.00	-0.00	-0.00	-0.01	0.01	-0.01	0.01	0.01	-0.00	0.00
0.00	-0.00	-0.00	-0.00	0.00	-0.00	-0.00	-0.00	0.01	0.00
-0.00	-0.00	-0.01	-0.01	0.00	0.01	-0.00	0.01	-0.01	0.01
0.01	-0.00	0.00	-0.01	0.00	-0.00	-0.01	0.01	0.00	-0.00
-0.00	0.01	0.01	-0.00	-0.01	-0.01	0.00	-0.01	0.02	-0.01
0.01	0.01	-0.00	0.02	-0.01	0.00	0.01	-0.02	-0.01	0.00

## **Observaciones**

Como puede verse, todos los clientes fueron modificados con el fin de que pudiera controlarse el contador de cada de los semáforos, este contador es el que sirve para que el cliente de lectura sepa en qué momento puede ejecutarse, realmente las modificaciones son mínimas, pues solamente se obtiene el semáforo y al final de cada uno de estos se modifica.

Sin embargo, para que pueda haber una correcta ejecución, en el programa servidor tienen que obtenerse los semáforos y esperar a que terminen de ejecutarse.

6. Programe la misma aplicación del punto 9 de la práctica 7 en la versión de memoria compartida, pero en esta ocasión sólo se usarán tres regiones de memoria compartidas para almacenar todas las matrices requeridas por la aplicación. Utilice como mecanismo de sincronización los semáforos revisados en esta práctica tanto para la escritura y como para la lectura de las memorias compartidas. Úselos en los lugares donde haya necesidad de sincronizar el acceso a memoria compartida.

## Windows

### SERVIDOR

Genera dos matrices 10x10 con números aleatorios del 0 al 100, las imprime y luego las almacena en la memoria compartida utilizando la API de Windows. Primero, se inicializan las matrices y se llenan con valores aleatorios mediante un ciclo for. Despues de llenar las matrices, se imprimen en la consola.

Seguido, se define una función CrearMemoriaCompartida que se encarga de crear un segmento de memoria compartida y copiar los datos de una matriz en esa memoria. Esta función utiliza CreateFileMapping para crear un bloque de memoria compartida y MapViewOfFile para mapearlo en el espacio de direcciones del proceso. Luego, copia los datos de la matriz a la memoria compartida usando memcpy y finalmente desmapea la memoria compartida con UnmapViewOfFile y cierra el manejador con CloseHandle.

En el main, se llaman a CrearMemoriaCompartida para las matrices a y b con identificadores únicos. Además, se crean dos semáforos: uno para controlar la escritura (hSemaforoEscritura) y otro para la lectura (hSemaforoLectura). El semáforo de escritura se libera para permitir a otros procesos escribir y luego se espera indefinidamente a que todos los clientes terminen usando WaitForSingleObject en el semáforo de lectura. Finalmente, se cierran los manejadores de los semáforos y el programa termina.

```
C ejercicio6_servidor.c > ...
1 // SERVIDOR DE LA MEMORIA COMPARTIDA
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <windows.h>
6 #include <time.h>
7 #define TAM_MEM 806      // Tamaño de la memoria compartida
8
9 void CrearMemoriaCompartida(char *idMemCompartida, int a[10][10], int
10 b[10][10], size_t tam);
11
12 int main()
13 {
14     // Declaración de variables
15     int a[10][10], b[10][10], i, j;
16     srand(time(NULL));
17
18     // Ciclo que se encarga de llenar las matrices
19     for(i=0; i<10; i++)
20     {
21         for(j=0; j<10; j++)
22         {
23             // La matriz se llena con números aleatorios del 0 al 100
24             a[i][j] = rand() % 101;
25             b[i][j] = rand() % 101;
26         }
27     }
28
29     printf("\nLa matriz A es:\n");
30     for(i=0; i<10; i++)
31     {
32         for(j=0; j<10; j++)
33             printf("%d\t", a[i][j]);
34         printf("\n");
35     }
36
37     printf("\nLa matriz B es:\n");
38     for(i=0; i<10; i++)
39     {
40         for(j=0; j<10; j++)
41             printf("%d\t", b[i][j]);
42         printf("\n");
43     }
44
45     CrearMemoriaCompartida("MemoriaCompartidaA", a, NULL, sizeof(a));
46     CrearMemoriaCompartida("MemoriaCompartidaB", b, NULL, sizeof(b));
47
```

```

C ejercicio6_servidor.c > ...
12 int main()
47 // Inicialización del semáforo
48 HANDLE hSemaforoEscritura = CreateSemaphore(NULL, 1, 1, "SemaforoEscritura");
49 HANDLE hSemaforoLectura = CreateSemaphore(NULL, 0, 1, "SemaforoLectura");
50
51 // Liberar el semáforo de escritura y esperar que todos los clientes terminen
52 ReleaseSemaphore(hSemaforoEscritura, 1, NULL);
53 WaitForSingleObject(hSemaforoLectura, INFINITE);
54
55 CloseHandle(hSemaforoEscritura);
56 CloseHandle(hSemaforoLectura);
57
58 exit(0);
59 }
60
61
62 void CrearMemoriaCompartida(char *idMemCompartida, int a[10][10], int
63 b[10][10], size_t tam)
64 {
65 // Declaraciones necesarias para usar memoria compartida
66 HANDLE hMemComp;
67 void *apDatos;
68
69 // Intenta crear un bloque de memoria compartida
70 if(hMemComp = CreateFileMapping(
71     INVALID_HANDLE_VALUE, // Usa la memoria compartida
72     NULL, // Seguridad por defecto
73     PAGE_READWRITE, // Acceso de lectura/escritura a la memoria compartida
74     0,
75     TAM_MEM, // Tamaño máximo parte baja de un DWORD
76     idMemCompartida) // Identificador de memoria compartida
77     ) == NULL)
78 {
79     printf("\nNo se creo la memoria compartida: (%i)", GetLastError());
80     exit(-1);
81 }
82
83 if((apDatos = MapViewOfFile(hMemComp, // Manejador del mapeo
84     FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
85     0,
86     0,
87     TAM_MEM)) == NULL)
88 {
89     printf("\nNo se enlazo la memoria compartida (%i)", GetLastError());
90     CloseHandle(hMemComp);
91     exit(-1);
92 }
93
94 // Copiamos las matrices en la región de memoria compartida
95 if (a != NULL) {
96     memcpy(apDatos, a, tam);
97 } else if (b != NULL) {
98     memcpy(apDatos, b, tam);
99 }
100
101 UnmapViewOfFile(apDatos);
102 CloseHandle(hMemComp);
103
104 exit(0);
105 }

```

```

C ejercicio6_servidor.c > ...
63     b[10][10], size_t tam)
87     TAM_MEM)) == NULL)
92
93
94 // Copiamos las matrices en la región de memoria compartida
95 if (a != NULL) {
96     memcpy(apDatos, a, tam);
97 } else if (b != NULL) {
98     memcpy(apDatos, b, tam);
99 }
100
101 UnmapViewOfFile(apDatos);
102 CloseHandle(hMemComp);
103
104 exit(0);
105 }

```

SUMA

Implementa un cliente que accede a matrices almacenadas en memoria compartida por un servidor, las suma y guarda el resultado en un archivo. Primero, abre y mapea los segmentos de memoria compartida para las matrices a, b y c. Utiliza semáforos para sincronizar el acceso, esperando a que el servidor termine de escribir antes de proceder.

Accede a las matrices a y b desde la memoria compartida, realiza la suma elemento a elemento, y almacena el resultado en la matriz c. Luego, escribe los resultados de la suma en el archivo "suma.txt". Así, libera los semáforos, desmapea las vistas de la memoria y cierra los manejadores de los segmentos de memoria compartida antes de terminar el programa.

```
C ejercicio6_suma.c > ⊖ main()
1 // CLIENTE DE LA MEMORIA COMPARTIDA - SUMA DE MATRICES
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <windows.h>
6 #define TAM_MEM 806 // Tamaño de la memoria compartida
7
8 int main()
9 {
10     // Declaraciones para el uso de memoria compartida
11     HANDLE hMemCompA, hMemCompB, hMemCompC;
12     char *idMemCompartidaA = "MemoriaCompartidaA";
13     char *idMemCompartidaB = "MemoriaCompartidaB";
14     char *idMemCompartidaC = "MemoriaCompartidaC";
15     void *apDatosA, *apDatosB, *apDatosC;
16     FILE *archivo = fopen("suma.txt", "w");
17     int i, j;
18
19     // Inicializar semáforos
20     HANDLE hSemaphoreEscritura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoEscritura");
21     HANDLE hSemaphoreLectura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoLectura");
22
23     // Esperar a que el servidor termine de escribir en la memoria compartida
24     WaitForSingleObject(hSemaphoreEscritura, INFINITE);
25
26     if((hMemCompA = OpenFileMapping(
27         FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
28         FALSE, // No se hereda el nombre
29         idMemCompartidaA)
30     ) == NULL)
31     {
32         printf("\nNo se accedio a la memoria compartida A: (%i)", GetLastError());
33         exit(-1);
34     }
35
36     if((apDatosA = MapViewOfFile(hMemCompA, // Manejador del mapeo
37         FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
38         0,
39         0,
40         TAM_MEM)) == NULL)
41     {
42         printf("\nNo se enlazo la memoria compartida A: (%i)", GetLastError());
43         CloseHandle(hMemCompA);
44         exit(-1);
45     }
```

```
C ejercicio6_suma.c > main()
8 int main()
47 if((hMemCompB = OpenFileMapping(
48     FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
49     FALSE, // No se hereda el nombre
50     idMemCompartidaB)
51     ) == NULL)
52 {
53     printf("\nNo se accedio a la memoria compartida B: (%i)", GetLastError());
54     exit(-1);
55 }
56
57 if((apDatosB = MapViewOfFile(hMemCompB, // Manejador del mapeo
58     FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
59     0,
60     0,
61     TAM_MEM)) == NULL)
62 {
63     printf("\nNo se enlazo la memoria compartida B: (%i)", GetLastError());
64     CloseHandle(hMemCompB);
65     exit(-1);
66 }
67
68 if((hMemCompC = OpenFileMapping(
69     FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
70     FALSE, // No se hereda el nombre
71     idMemCompartidaC)
72     ) == NULL)
73 {
74     printf("\nNo se accedio a la memoria compartida C: (%i)", GetLastError());
75     exit(-1);
76 }
77
78 if((apDatosC = MapViewOfFile(hMemCompC, // Manejador del mapeo
79     FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
80     0,
81     0,
82     TAM_MEM)) == NULL)
83 {
84     printf("\nNo se enlazo la memoria compartida C: (%i)", GetLastError());
85     CloseHandle(hMemCompC);
86     exit(-1);
87 }
88
89 // Lee las matrices directamente desde la memoria
90 int (*a)[10] = (int (*)[10])(apDatosA);
91 int (*b)[10] = (int (*)[10])(apDatosB);
92 int (*c)[10] = (int (*)[10])(apDatosC);
93
```

```
C ejercicio6_suma.c > main()
8 int main()
93
94     fprintf(archivo, "\nEl resultado de la suma A+B es:\n");
95     // Escribe el resultado de las matrices en un archivo
96     for(i=0; i<10; i++)
97     {
98         for(j=0; j<10; j++)
99         {
100             c[i][j] = a[i][j] + b[i][j];
101             fprintf(archivo, "%d\t", c[i][j]);
102         }
103         fprintf(archivo, "\n");
104     }
105
106     fclose(archivo);
107
108     // Liberar el semáforo de lectura
109     ReleaseSemaphore(hSemaforoLectura, 1, NULL);
110
111     UnmapViewOfFile(apDatosA);
112     UnmapViewOfFile(apDatosB);
113     UnmapViewOfFile(apDatosC);
114     CloseHandle(hMemCompA);
115     CloseHandle(hMemCompB);
116     CloseHandle(hMemCompC);
117     exit(0);
118 }
```

## RESTA

Primero, el programa declara variables para manejar la memoria compartida y abre un archivo llamado "resta.txt" para escribir los resultados de la resta de las matrices. Luego, inicializa los semáforos necesarios para sincronizar el acceso a la memoria compartida.

El programa espera a que el servidor termine de escribir en la memoria compartida utilizando el semáforo hSemaforoEscritura. Después, intenta abrir y mapear los segmentos de memoria compartida correspondientes a las matrices a y b (MemoriaCompartidaA y MemoriaCompartidaB). Si no puede acceder a la memoria compartida, muestra un mensaje de error y finaliza el programa.

Una vez que las matrices a y b están mapeadas en el espacio de direcciones del proceso, el programa intenta abrir y mapear un segmento de memoria compartida para almacenar la matriz resultante de la resta (MemoriaCompartidaC). Si no puede acceder a esta memoria compartida, muestra un mensaje de error y finaliza el programa.

El programa accede a las matrices a y b directamente desde la memoria compartida, realiza la resta elemento a elemento y almacena el resultado en la matriz c. Luego, escribe los resultados de la resta en el archivo "resta.txt".

Finalmente, libera el semáforo de lectura hSemaforoLectura, desmapea las vistas de la memoria compartida y cierra los manejadores de los segmentos de memoria compartida antes de terminar el programa.

```
C ejercicio6_resta.c > main()
1 // CLIENTE DE LA MEMORIA COMPARTIDA - RESTA DE MATRICES
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <windows.h>
5 #define TAM_MEM 806 // Tamaño de la memoria compartida
6 int main()
7 {
8 // Declaraciones para el uso de memoria compartida
9 HANDLE hMemCompA, hMemCompB, hMemCompC;
10 char *idMemCompartidaA = "MemoriaCompartidaA";
11     char *idMemCompartidaB = "MemoriaCompartidaB";
12     char *idMemCompartidaC = "MemoriaCompartidaC";
13     void *apDatosA, *apDatosB, *apDatosC;
14     FILE *archivo = fopen("resta.txt", "w");
15     int i, j;
16
17 // Inicializar semáforos
18 HANDLE hSemaforoEscritura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoEscritura");
19 HANDLE hSemaforoLectura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoLectura");
20
21 // Esperar a que el servidor termine de escribir en la memoria compartida
22 WaitForSingleObject(hSemaforoEscritura, INFINITE);
23
24 if((hMemCompA = OpenFileMapping(
25     FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
26     FALSE, // No se hereda el nombre
27     idMemCompartidaA)
28 ) == NULL)
29 {
30     printf("\nNo se accedio a la memoria compartida A: (%i)",
31 GetLastError());
32     exit(-1);
33 }
34
35 if((apDatosA = MapViewOfFile(hMemCompA, // Manejador del mapeo
36     FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
37     0,
38     0,
39     TAM_MEM)) == NULL)
40 {
41     printf("\nNo se enlazo la memoria compartida A: (%i)", GetLastError());
42     CloseHandle(hMemCompA);
43     exit(-1);
44 }
```

```
C ejercicio6_resta.c > main()
6 int main()
46     if((hMemCompB = OpenFileMapping(
47             FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
48             FALSE, // No se hereda el nombre
49             idMemCompartidaB)
50             ) == NULL)
51     {
52         printf("\nNo se accedio a la memoria compartida B: (%i)", GetLastError());
53         exit(-1);
54     }
55
56     if((apDatosB = MapViewOfFile(hMemCompB, // Manejador del mapeo
57             FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
58             0,
59             0,
60             TAM_MEM)) == NULL)
61     {
62         printf("\nNo se enlazo la memoria compartida B: (%i)", GetLastError());
63         CloseHandle(hMemCompB);
64         exit(-1);
65     }
66
67     if((hMemCompC = OpenFileMapping(
68             FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
69             FALSE, // No se hereda el nombre
70             idMemCompartidaC)
71             ) == NULL)
72     {
73         printf("\nNo se accedio a la memoria compartida C: (%i)", GetLastError());
74         exit(-1);
75     }
76
77     if((apDatosC = MapViewOfFile(hMemCompC, // Manejador del mapeo
78             FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
79             0,
80             0,
81             TAM_MEM)) == NULL)
82     {
83         printf("\nNo se enlazo la memoria compartida C: (%i)", GetLastError());
84         CloseHandle(hMemCompC);
85         exit(-1);
86     }
```

```

87
88 // Lee las matrices directamente desde la memoria
89 int (*a)[10] = (int (*)[10])(apDatosA);
90 int (*b)[10] = (int (*)[10])(apDatosB);
91 int (*c)[10] = (int (*)[10])(apDatosC);
92
93 fprintf(archivo, "\nEl resultado de la resta A-B es:\n");
94 // Escribe el resultado de las matrices en un archivo
95 for(i=0; i<10; i++)
96 {
97     for(j=0; j<10; j++)
98     {
99         c[i][j] = a[i][j] - b[i][j];
100        fprintf(archivo, "%d\t", c[i][j]);
101    }
102    fprintf(archivo, "\n");
103 }
104
105 fclose(archivo);
106
107 // Liberar el semáforo de lectura
108 ReleaseSemaphore(hSemaforoLectura, 1, NULL);
109
110 UnmapViewOfFile(apDatosA);
111 UnmapViewOfFile(apDatosB);
112 UnmapViewOfFile(apDatosC);
113 CloseHandle(hMemCompA);
114 CloseHandle(hMemCompB);
115 CloseHandle(hMemCompC);
116
117 exit(0);
118 }
```

## MULTIPLICACION

Se implementa un cliente que accede a matrices almacenadas en memoria compartida por un servidor, realiza la multiplicación de estas matrices y guarda el resultado en un archivo llamado "multiplicacion.txt" para escribir los resultados de la multiplicación de las matrices. Luego, inicializa los semáforos necesarios para sincronizar el acceso a la memoria compartida.

El programa espera a que el servidor termine de escribir en la memoria compartida utilizando el semáforo hSemaforoEscritura. Después, intenta abrir y mapear los segmentos de memoria compartida correspondientes a las matrices a y b (MemoriaCompartidaA y MemoriaCompartidaB). Si no puede acceder a la memoria compartida, muestra un mensaje de error y finaliza el programa.

Una vez que las matrices a y b están mapeadas en el espacio de direcciones del proceso, el programa intenta abrir y mapear un segmento de memoria compartida para almacenar la matriz resultante de la multiplicación (MemoriaCompartidaC). Si no puede acceder a esta memoria compartida, muestra un mensaje de error y finaliza el programa.

El programa accede a las matrices a y b directamente desde la memoria compartida, realiza la multiplicación de matrices, y almacena el resultado en la matriz c. Luego, escribe los resultados de la multiplicación en el archivo "multiplicacion.txt".

```
C ejercicio6_multiplicacion.c > main()
1 // CLIENTE DE LA MEMORIA COMPARTIDA - MULTIPLICACIÓN DE MATRICES
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <windows.h>
6 #define TAM_MEM 806 // Tamaño de la memoria compartida
7
8 int main()
9 {
10    // Declaraciones para el uso de memoria compartida
11    HANDLE hMemCompA, hMemCompB, hMemCompC;
12    char *idMemCompartidaA = "MemoriaCompartidaA";
13    char *idMemCompartidaB = "MemoriaCompartidaB";
14    char *idMemCompartidaC = "MemoriaCompartidaC";
15    void *apDatosA, *apDatosB, *apDatosC;
16    FILE *archivo = fopen("multiplicacion.txt", "w");
17    int i, j, k;
18
19    // Inicializar semáforos
20    HANDLE hSemaforoEscritura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoEscritura");
21    HANDLE hSemaforoLectura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoLectura");
22
23    // Esperar a que el servidor termine de escribir en la memoria compartida
24    WaitForSingleObject(hSemaforoEscritura, INFINITE);
25
26    if((hMemCompA = OpenFileMapping(
27        FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
28        FALSE, // No se hereda el nombre
29        idMemCompartidaA)
30        ) == NULL)
31    {
32        printf("\nNo se accedio a la memoria compartida A: (%i)", GetLastError());
33        exit(-1);
34    }
35
36    if((apDatosA = MapViewOfFile(hMemCompA, // Manejador del mapeo
37        FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
38        0,
39        0,
40        TAM_MEM)) == NULL)
41    {
42        printf("\nNo se enlazo la memoria compartida A: (%i)", GetLastError());
43        CloseHandle(hMemCompA);
44        exit(-1);
45    }
```

```
C ejercicio6_multiplicacion.c > main()
123    }
40        TAM_MEM)) == NULL)
45    }
46
47    if((hMemCompB = OpenFileMapping(
48        FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
49        FALSE, // No se hereda el nombre
50        idMemCompartidaB)
51        ) == NULL)
52    {
53        printf("\nNo se accedio a la memoria compartida B: (%i)", GetLastError());
54        exit(-1);
55    }
56
57    if((apDatosB = MapViewOfFile(hMemCompB, // Manejador del mapeo
58        FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
59        0,
60        0,
61        TAM_MEM)) == NULL)
62    {
63        printf("\nNo se enlazo la memoria compartida B: (%i)", GetLastError());
64        CloseHandle(hMemCompB);
65        exit(-1);
66    }
67
68    if((hMemCompC = OpenFileMapping(
69        FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
70        FALSE, // No se hereda el nombre
71        idMemCompartidaC)
72        ) == NULL)
73    {
74        printf("\nNo se accedio a la memoria compartida C: (%i)", GetLastError());
75        exit(-1);
76    }
77
78    if((apDatosC = MapViewOfFile(hMemCompC, // Manejador del mapeo
79        FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
80        0,
81        0,
82        TAM_MEM)) == NULL)
83    {
84        printf("\nNo se enlazo la memoria compartida C: (%i)", GetLastError());
85        CloseHandle(hMemCompC);
86        exit(-1);
87    }
```

```

ejercicio6_multiplicacion.c > main()
8 int main()
9 // Lee las matrices directamente desde la memoria
10 int (*a)[10] = (int (*)[10])(apDatosA);
11 int (*b)[10] = (int (*)[10])(apDatosB);
12 int (*c)[10] = (int (*)[10])(apDatosC);
13
14
15 // Escribe el resultado de las matrices en un archivo
16 for(i=0; i<10; i++)
17 {
18     for(j=0; j<10; j++)
19     {
20         c[i][j] = 0;
21         for (k=0; k<10; k++)
22         {
23             c[i][j] += a[i][k] * b[k][j];
24         }
25         fprintf(archivo, "%d\t", c[i][j]);
26     }
27     fprintf(archivo, "\n");
28 }
29
30 fclose(archivo);
31
32 // Liberar el semáforo de lectura
33 ReleaseSemaphore(hSemaforoLectura, 1, NULL);
34
35 UnmapViewOfFile(apDatosA);
36 UnmapViewOfFile(apDatosB);
37 UnmapViewOfFile(apDatosC);
38 CloseHandle(hMemCompA);
39 CloseHandle(hMemCompB);
40 CloseHandle(hMemCompC);
41 exit(0);
42 }
```

## TRANSPUESTA

Se accede a matrices almacenadas en memoria compartida por un servidor, calcula sus transpuestas y guarda los resultados en un archivo. Utiliza semáforos y funciones de la API de Windows para manejar la memoria compartida y sincronizar el acceso.

Primero, se declaran las variables necesarias para manejar la memoria compartida (hMemCompA, hMemCompB, hMemCompC) y los identificadores correspondientes (idMemCompartidaA, idMemCompartidaB, idMemCompartidaC). Además, se abre un archivo llamado "transpuesta.txt" para escribir los resultados.

Los semáforos de escritura y lectura (hSemaforoEscritura, hSemaforoLectura) se inicializan con OpenSemaphore para coordinar el acceso a la memoria compartida. El programa espera a que el

servidor termine de escribir en la memoria compartida mediante WaitForSingleObject en el semáforo de escritura.

Luego, se abre el mapeo de la memoria compartida usando OpenFileMapping y se enlaza con MapViewOfFile para cada una de las matrices A, B y C. Si alguna de estas operaciones falla, el programa muestra un mensaje de error y termina.

Se leen las matrices directamente desde la memoria compartida y se calculan sus transpuestas. Los resultados se escriben en el archivo "transpuesta.txt". Una vez finalizado, se libera el semáforo de lectura con ReleaseSemaphore, se desvinculan las vistas de la memoria compartida con UnMapViewOfFile y se cierran los manejadores de la memoria compartida con CloseHandle.

```
C ejercicio6_transpuesta.c > main()
1 // CLIENTE DE LA MEMORIA COMPARTIDA - TRANSPUESTA DE UNA MATRIZ
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <windows.h>
5 #define TAM_MEM 806 // Tamaño de la memoria compartida
6 int main()
7 {
8 // Declaraciones para el uso de memoria compartida
9 HANDLE hMemCompA, hMemCompB, hMemCompC;
10 char *idMemCompartidaA = "MemoriaCompartidaA";
11 char *idMemCompartidaB = "MemoriaCompartidaB";
12 char *idMemCompartidaC = "MemoriaCompartidaC";
13 void *apDatosA, *apDatosB, *apDatosC;
14 FILE *archivo = fopen("transpuesta.txt", "w");
15 int i, j;
16
17 // Inicializar semáforos
18 HANDLE hSemaforoEscritura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoEscritura");
19 HANDLE hSemaforoLectura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoLectura");
20
21 // Esperar a que el servidor termine de escribir en la memoria compartida
22 WaitForSingleObject(hSemaforoEscritura, INFINITE);
23
24 if((hMemCompA = OpenFileMapping(
25     FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
26     FALSE, // No se hereda el nombre
27     idMemCompartidaA
28 ) == NULL)
29 {
30     printf("\nNo se accedio a la memoria compartida A: (%i)", GetLastError());
31     exit(-1);
32 }
33
34 if((apDatosA = MapViewOfFile(hMemCompA, // Manejador del mapeo
35     FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
36     0,
37     0,
38     TAM_MEM)) == NULL)
39 {
40     printf("\nNo se enlazo la memoria compartida A: (%i)", GetLastError());
41     CloseHandle(hMemCompA);
42     exit(-1);
43 }
```

```
C ejercicio6_transpuesta.c > main()
6 int main()
44
45     if((hMemCompB = OpenFileMapping(
46         FILE_MAP_ALL_ACCESS,      // Acceso a la memoria compartida
47         FALSE, // No se hereda el nombre
48         idMemCompartidaB)
49     ) == NULL)
50     {
51         printf("\nNo se accedio a la memoria compartida B: (%i)", GetLastError());
52         exit(-1);
53     }
54
55     if((apDatosB = MapViewOfFile(hMemCompB, // Manejador del mapeo
56         FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
57         0,
58         0,
59         TAM_MEM)) == NULL)
60     {
61         printf("\nNo se enlazo la memoria compartida B: (%i)", GetLastError());
62         CloseHandle(hMemCompB);
63         exit(-1);
64     }
65
66     if((hMemCompC = OpenFileMapping(
67         FILE_MAP_ALL_ACCESS,      // Acceso a la memoria compartida
68         FALSE, // No se hereda el nombre
69         idMemCompartidaC)
70     ) == NULL)
71     {
72         printf("\nNo se accedio a la memoria compartida C: (%i)", GetLastError());
73         exit(-1);
74     }
75
76     if((apDatosC = MapViewOfFile(hMemCompC, // Manejador del mapeo
77         FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
78         0,
79         0,
80         TAM_MEM)) == NULL)
81     {
82         printf("\nNo se enlazo la memoria compartida C: (%i)", GetLastError());
83         CloseHandle(hMemCompC);
84         exit(-1);
85     }
```

```

C ejercicio6_transpuesta.c > main()
6   int main()
80     TAM_MEM)) == NULL)
85   }
86
87   // Lee las matrices directamente desde la memoria
88   int (*a)[10] = (int (*)[10])(apDatosA);
89   int (*b)[10] = (int (*)[10])(apDatosB);
90   int (*c)[10] = (int (*)[10])(apDatosC);
91
92   fprintf(archivo, "\nLa transpuesta de la matriz A es:\n");
93   for(i=0; i<10; i++)
94   {
95     for(j=0; j<10; j++)
96       fprintf(archivo, "%d\t", a[j][i]);
97     fprintf(archivo, "\n");
98   }
99
100  fprintf(archivo, "\nLa transpuesta de la matriz B es:\n");
101  for(i=0; i<10; i++)
102  {
103    for(j=0; j<10; j++)
104      fprintf(archivo, "%d\t", b[j][i]);
105    fprintf(archivo, "\n");
106  }
107
108  fclose(archivo);
109
110  // Liberar el semáforo de lectura
111  ReleaseSemaphore(hSemaforoLectura, 1, NULL);
112
113  UnmapViewOfFile(apDatosA);
114  UnmapViewOfFile(apDatosB);
115  UnmapViewOfFile(apDatosC);
116  CloseHandle(hMemCompA);
117  CloseHandle(hMemCompB);
118  CloseHandle(hMemCompC);
119
120  exit(0);
121 }
```

## INVERSA

Este programa utiliza memoria compartida y semáforos en un entorno Windows para realizar cálculos de inversión de matrices. Comienza estableciendo acceso a la memoria compartida y sincronizando la lectura y escritura entre procesos mediante semáforos. Luego, lee las matrices desde la memoria compartida, las convierte a matrices de tipo double y calcula sus inversas utilizando el método de eliminación gaussiana con pivoteo parcial. Finalmente, escribe las matrices inversas en un archivo de texto y libera los recursos correctamente.

```
C ejercicio6_inversa.c > ...
1 // CLIENTE DE LA MEMORIA COMPARTIDA - INVERSA DE LAS MATRICES
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <windows.h>
6 #define TAM_MEM 806 // Tamaño de la memoria compartida
7
8 int main()
9 {
10    // Declaraciones para el uso de memoria compartida
11    HANDLE hMemCompA, hMemCompB, hMemCompC;
12    char *idMemCompartidaA = "MemoriaCompartidaA";
13    char *idMemCompartidaB = "MemoriaCompartidaB";
14    char *idMemCompartidaC = "MemoriaCompartidaC";
15    void *apDatosA, *apDatosB, *apDatosC;
16    FILE *archivo = fopen("inversa.txt", "w");
17    int i, j, k;
18    double id[10][10], id2[10][10], m1[10][10], m2[10][10], pivote, pivote2, aux, aux2;
19
20    // Inicializar semáforos
21    HANDLE hSemaforoEscritura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoEscritura");
22    HANDLE hSemaforoLectura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoLectura");
23
24    // Esperar a que el servidor termine de escribir en la memoria compartida
25    WaitForSingleObject(hSemaforoEscritura, INFINITE);
26
27    if((hMemCompA = OpenFileMapping(
28        FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
29        FALSE, // No se hereda el nombre
30        idMemCompartidaA
31        ) == NULL)
32    {
33        printf("\nNo se accedio a la memoria compartida A: (%i)", GetLastError());
34        exit(-1);
35    }
36
37    if((apDatosA = MapViewOfFile(hMemCompA, // Manejador del mapeo
38        FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
39        0,
40        0,
41        TAM_MEM)) == NULL)
42    {
43        printf("\nNo se enlazo la memoria compartida A: (%i)", GetLastError());
44        CloseHandle(hMemCompA);
45        exit(-1);
46    }
```

```
C ejercicio6_inversa.c > ...
8 int main()
47
48     if((hMemCompB = OpenFileMapping(
49         FILE_MAP_ALL_ACCESS,    // Acceso a la memoria compartida
50         FALSE, // No se hereda el nombre
51         idMemCompartidaB)
52         ) == NULL)
53     {
54         printf("\nNo se accedio a la memoria compartida B: (%i)", GetLastError());
55         exit(-1);
56     }
57
58     if((apDatosB = MapViewOfFile(hMemCompB, // Manejador del mapeo
59         FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
60         0,
61         0,
62         TAM_MEM)) == NULL)
63     {
64         printf("\nNo se enlazo la memoria compartida B: (%i)", GetLastError());
65         CloseHandle(hMemCompB);
66         exit(-1);
67     }
68
69     if((hMemCompC = OpenFileMapping(
70         FILE_MAP_ALL_ACCESS,    // Acceso a la memoria compartida
71         FALSE, // No se hereda el nombre
72         idMemCompartidaC)
73         ) == NULL)
74     {
75         printf("\nNo se accedio a la memoria compartida C: (%i)", GetLastError());
76         exit(-1);
77     }
78
79     if((apDatosC = MapViewOfFile(hMemCompC, // Manejador del mapeo
80         FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
81         0,
82         0,
83         TAM_MEM)) == NULL)
84     {
85         printf("\nNo se enlazo la memoria compartida C: (%i)", GetLastError());
86         CloseHandle(hMemCompC);
87         exit(-1);
88     }
89
```

```

C ejercicio6_inversa.c > ...
8 int main()
9 {
10     // Lee las matrices directamente desde la memoria
11     int (*a)[10] = (int (*)[10])(apDatosA);
12     int (*b)[10] = (int (*)[10])(apDatosB);
13     int (*c)[10] = (int (*)[10])(apDatosC);
14
15     // Convierte las matrices a matrices de tipo double
16     for(i=0; i<10; i++)
17         for(j=0; j<10; j++)
18         {
19             m1[i][j] = a[i][j];
20             m2[i][j] = b[i][j];
21             id[i][j] = 0;
22             id2[i][j] = 0;
23             if(i == j)
24             {
25                 id[i][j] = 1;
26                 id2[i][j] = 1;
27             }
28         }
29
30     // Obtiene la inversa de la matriz 1
31     // Reducción por renglones
32     for(i=0; i<10; i++)
33     {
34         pivot = m1[i][i]; // Pivote de la matriz 1
35         pivot2 = m2[i][i]; // Pivote de la matriz 2
36         // Operación que debemos realizar para cada columna
37         for(j=0; j<10; j++)
38         {
39             // Matriz 1
40             m1[i][j] = m1[i][j]/pivot; // La realiza sobre la matriz
41             id[i][j] = id[i][j]/pivot; // La realiza en la matriz extendida
42             // Matriz 2
43             m2[i][j] = m2[i][j]/pivot2; // Realiza la operación sobre la matriz
44             id2[i][j] = id2[i][j]/pivot2; // Realiza la operación sobre la matriz
45         }
46         for(k=0; k<10; k++) // Vuelve 0 a todos los elementos debajo y encima del pivote
47         {
48             if(i!=k) // No estoy en la diagonal
49             {
50                 aux = m1[k][i]; // Agarra el valor que quiere convertir a 0
51                 aux2 = m2[k][i]; // Auxiliar de la matriz 2
52                 for(j=0; j<10; j++)
53                 {
54                     // Matriz 1
55                     m1[k][j] = m1[k][j]-aux*m1[i][j]; // Realiza la operación en la matriz
56                     m2[k][j] = m2[k][j]-aux2*m2[i][j]; // Realiza la operación en la matriz
57                 }
58             }
59         }
60     }
61 }

```

```

C ejercicio6_inversa.c > ...
8  int main()
112 <   for(i=0; i<10; i++)
126 <     for(k=0; k<10; k++) // Vuelve 0 a todos los elementos debajo y encima del pivote
128 <       if(i!=k) // No estoy en la diagonal
132 <         for(j=0; j<10; j++)
135   m1[k][j] = m1[k][j]-aux*m1[i][j]; // Realiza la operación en la matriz
136   id[k][j] = id[k][j]-aux*id[i][j]; // Realiza la operación en la matriz extendida
137   // Matriz 2
138   m2[k][j] = m2[k][j]-aux2*m2[i][j]; // Operación en la matriz
139   id2[k][j] = id2[k][j]-aux2*id2[i][j]; // Operación en la matriz extendida 1
140 }
141 }
142 }
143 }

// Imprime la inversa de la matriz 1
146 fprintf(archivo, "La inversa de la matriz A es:\n");
147 for(i=0; i<10; i++)
148 {
149   for(j=0; j<10; j++)
150     fprintf(archivo, "%2f\t", id[i][j]);
151   fprintf(archivo, "\n");
152 }
153 fprintf(archivo, "\n");

// Imprime la inversa de la matriz 2
156 fprintf(archivo, "\nLa inversa de la matriz B es:\n");
157 for(i=0; i<10; i++)
158 {
159   for(j=0; j<10; j++)
160     fprintf(archivo, "%2f\t", id2[i][j]);
161   fprintf(archivo, "\n");
162 }

fclose(archivo);
// Liberar el semáforo de lectura
166 ReleaseSemaphore(hSemaphoreLectura, 1, NULL);
167 UnmapViewOfFile(apDatosA);
168 UnmapViewOfFile(apDatosB);
169 UnmapViewOfFile(apDatosC);
170 CloseHandle(hMemCompA);
171 CloseHandle(hMemCompB);
172 CloseHandle(hMemCompC);
173 exit(0);
174 }

```

## LECTURA

Este programa en C actúa como cliente para leer y mostrar los resultados almacenados en archivos de texto generados previamente por un servidor. Utiliza memoria compartida y semáforos para garantizar la sincronización adecuada con el servidor.

Primero, el programa accede y mapea las tres áreas de memoria compartida correspondientes a las matrices A, B y C. Luego, utiliza funciones de manejo de archivos para abrir y leer los contenidos de cinco archivos de texto específicos ("suma.txt", "resta.txt", "multiplicacion.txt", "transpuesta.txt", "inversa.txt").

Para cada archivo:

1. Utiliza CreateFileA para abrir el archivo en modo lectura.
2. Lee el contenido del archivo utilizando ReadFile en un bucle hasta que se lea todo el archivo.
3. Muestra el contenido leído en la consola.

Después de leer y mostrar todos los archivos, libera el semáforo de lectura para indicar al servidor que los datos han sido leídos correctamente. Finalmente, limpia y libera todos los recursos utilizados, incluyendo el cierre de manejadores de memoria y archivos.

Este enfoque permite al cliente obtener y visualizar resultados complejos de operaciones matriciales realizadas por el servidor, facilitando una comunicación eficiente y segura entre procesos utilizando técnicas avanzadas de manejo de memoria compartida y sincronización con semáforos.

```
C ejercicio6_lectura.c > main()
1 // CLIENTE DE LA MEMORIA COMPARTIDA - LECTURA DE LOS RESULTADOS
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <windows.h>
6 #define TAM_MEM 806 // Tamaño de la memoria compartida
7
8 int main()
9 {
10     // Declaraciones para el uso de memoria compartida
11     HANDLE hMemCompA, hMemCompB, hMemCompC;
12     char *idMemCompartidaA = "MemoriaCompartidaA";
13     char *idMemCompartidaB = "MemoriaCompartidaB";
14     char *idMemCompartidaC = "MemoriaCompartidaC";
15     void *apDatosA, *apDatosB, *apDatosC;
16     int i, j, k;
17
18     // Declaración de variables para la lectura de archivos
19     char *ruta1, *ruta2, *ruta3, *ruta4, *ruta5, buffer[1024];
20     ruta1 = (char *)malloc(sizeof(char)*MAX_PATH);
21     ruta2 = (char *)malloc(sizeof(char)*MAX_PATH);
22     ruta3 = (char *)malloc(sizeof(char)*MAX_PATH);
23     ruta4 = (char *)malloc(sizeof(char)*MAX_PATH);
24     ruta5 = (char *)malloc(sizeof(char)*MAX_PATH);
25     // Se obtiene la ruta donde se encuentra el archivo
26     ruta1 = "suma.txt";
27     ruta2 = "resta.txt";
28     ruta3 = "multiplicacion.txt";
29     ruta4 = "transpuesta.txt";
30     ruta5 = "inversa.txt";
31
32     // Inicializar semáforos
33     HANDLE hSemaforoEscritura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoEscritura");
34     HANDLE hSemaforoLectura = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SemaforoLectura");
35
36     // Esperar a que el servidor termine de escribir en la memoria compartida
37     WaitForSingleObject(hSemaforoEscritura, INFINITE);
38
39     if(hMemCompA = OpenFileMapping(
40         FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
41         FALSE, // No se hereda el nombre
42         idMemCompartidaA
43         ) == NULL)
44     {
45         printf("\nNo se accedio a la memoria compartida A: (%i)", GetLastError());
46         exit(-1);
47     }
```

```
C ejercicio6_lectura.c > main()
8 int main()
43     ) == NULL)
48
49     if((apDatosA = MapViewOfFile(hMemCompA, // Manejador del mapeo
50         FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
51         0,
52         0,
53         TAM_MEM)) == NULL)
54     {
55         printf("\nNo se enlazo la memoria compartida A: (%i)", GetLastError());
56         CloseHandle(hMemCompA);
57         exit(-1);
58     }
59
60     if((hMemCompB = OpenFileMapping(
61         FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
62         FALSE, // No se hereda el nombre
63         idMemCompartidaB)
64         ) == NULL)
65     {
66         printf("\nNo se accedio a la memoria compartida B: (%i)", GetLastError());
67         exit(-1);
68     }
69
70     if((apDatosB = MapViewOfFile(hMemCompB, // Manejador del mapeo
71         FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
72         0,
73         0,
74         TAM_MEM)) == NULL)
75     {
76         printf("\nNo se enlazo la memoria compartida B: (%i)", GetLastError());
77         CloseHandle(hMemCompB);
78         exit(-1);
79     }
80
81     if((hMemCompC = OpenFileMapping(
82         FILE_MAP_ALL_ACCESS, // Acceso a la memoria compartida
83         FALSE, // No se hereda el nombre
84         idMemCompartidaC)
85         ) == NULL)
86     {
87         printf("\nNo se accedio a la memoria compartida C: (%i)", GetLastError());
88         exit(-1);
89     }
90
```

```
C ejercicio6_lectura.c > ⌂ main()
8     int main()
85         ) == NULL)
90
91     if((apDatosC = MapViewOfFile(hMemCompC, // Manejador del mapeo
92         FILE_MAP_ALL_ACCESS, // Permiso de lectura/escritura
93         0,
94         0,
95         TAM_MEM)) == NULL)
96     {
97         printf("\nNo se enlazo la memoria compartida C: (%i)", GetLastError());
98         CloseHandle(hMemCompC);
99         exit(-1);
100    }
101
102 // Verifica que el archivo exista y lee SUMA DE MATRICES
103 HANDLE archivo = CreateFileA(ruta1, GENERIC_READ, FILE_SHARE_READ,
104     NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
105 if(archivo != INVALID_HANDLE_VALUE)
106 {
107     // Muestra en pantalla el contenido del archivo
108     DWORD bytesleidos;
109     while(ReadFile(archivo, buffer, sizeof(buffer), &bytesleidos,
110         NULL) && bytesleidos > 0)
111         printf("%.*s", bytesleidos, buffer);
112     CloseHandle(archivo); // Cierra el archivo
113     printf("\n");
114 }
115 else
116     printf("\n\nNo se pudo abrir el archivo: %lu\n", GetLastError());
117
118 // LEE EL ARCHIVO DE RESTA
119 archivo = CreateFileA(ruta2, GENERIC_READ, FILE_SHARE_READ, NULL,
120     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
121 if(archivo != INVALID_HANDLE_VALUE)
122 {
123     // Muestra en pantalla el contenido del archivo
124     DWORD bytesleidos;
125     while(ReadFile(archivo, buffer, sizeof(buffer), &bytesleidos,
126         NULL) && bytesleidos > 0)
127         printf("%.*s", bytesleidos, buffer);
128     CloseHandle(archivo); // Cierra el archivo
129     printf("\n");
130 }
131 else
132     printf("\n\nNo se pudo abrir el archivo: %lu\n", GetLastError());
133
134 // LEE EL ARCHIVO DE MULTIPLICACIÓN
135 archivo = CreateFileA(ruta3, GENERIC_READ, FILE_SHARE_READ, NULL,
```

```

C ejercicio6_lectura.c > ⌂ main()
8 int main()
134 // LEE EL ARCHIVO DE MULTIPLICACIÓN
135 archivo = CreateFileA(ruta3, GENERIC_READ, FILE_SHARE_READ, NULL,
136     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
137 if(archivo != INVALID_HANDLE_VALUE)
138 {
139     // Muestra en pantalla el contenido del archivo
140     DWORD bytesleidos;
141     while(ReadFile(archivo, buffer, sizeof(buffer), &bytesleidos,
142         NULL) && bytesleidos > 0)
143         printf("%.*s", bytesleidos, buffer);
144     CloseHandle(archivo); // Cierra el archivo
145     printf("\n");
146 }
147 else
148     printf("\n\nNo se pudo abrir el archivo: %lu\n", GetLastError());
149
150 // LEE EL ARCHIVO DE TRANSPUESTA
151 archivo = CreateFileA(ruta4, GENERIC_READ, FILE_SHARE_READ, NULL,
152     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
153 if(archivo != INVALID_HANDLE_VALUE)
154 {
155     // Muestra en pantalla el contenido del archivo
156     DWORD bytesleidos;
157     while(ReadFile(archivo, buffer, sizeof(buffer), &bytesleidos, NULL) && bytesleidos > 0)
158         printf("%.*s", bytesleidos, buffer);
159     CloseHandle(archivo); // Cierra el archivo
160     printf("\n");
161 }
162 else
163     printf("\n\nNo se pudo abrir el archivo: %lu\n", GetLastError());
164
165 // LEE EL ARCHIVO DE LA INVERSA
166 archivo = CreateFileA(ruta5, GENERIC_READ, FILE_SHARE_READ, NULL,
167     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
168 if(archivo != INVALID_HANDLE_VALUE)
169 {
170     // Muestra en pantalla el contenido del archivo
171     DWORD bytesleidos;
172     while(ReadFile(archivo, buffer, sizeof(buffer), &bytesleidos,
173         NULL) && bytesleidos > 0)
174         printf("%.*s", bytesleidos, buffer);
175     CloseHandle(archivo); // Cierra el archivo
176     printf("\n");
177 }
178 else
179     printf("\n\nNo se pudo abrir el archivo: %lu\n", GetLastError());
180 // Liberar el semáforo de lectura
181 ReleaseSemaphore(hSemáforoLectura, 1, NULL);
182 UnmapViewOfFile(apDatosA);
183 UnmapViewOfFile(apDatosB);
184 UnmapViewOfFile(apDatosC);
185 CloseHandle(hMemCompA);
186 CloseHandle(hMemCompB);
187 CloseHandle(hMemCompC);
188 exit(0);
189 }

```

## EJECUCION

Podemos observar que el servidor es el encargado de generar las matrices A y B, las cuales serán compartidas mediante memoria compartida para que los programas de suma, resta, multiplicación, inversa y transpuesta puedan hacer uso de ellas y llevar a cabo sus tareas.

Similar a la práctica anterior, únicamente que en este caso fue necesario reducir el tamaño de la memoria a compartir y haciendo uso del método de sincronización de semáforos para coordinar el acceso a la memoria por cada uno de los programas.

### Captura de pantalla

La matriz A es:									
26	81	100	53	21	48	0	99	14	34
72	73	33	10	69	12	59	20	66	99
15	61	53	46	99	98	67	88	45	7
58	26	3	56	56	77	53	88	45	41
45	29	8	4	45	9	32	22	46	53
88	11	5	53	4	20	32	98	62	13
3	5	15	97	32	4	86	77	0	37
68	78	71	93	86	68	51	80	86	10
20	72	87	47	25	1	25	54	0	2
6	54	88	16	73	86	16	73	18	78
La matriz B es:									
38	17	69	63	57	42	78	73	52	42
43	35	26	31	83	54	78	45	65	86
43	77	81	13	17	97	16	66	37	1
73	26	10	81	68	77	42	54	0	70
54	32	24	85	42	26	27	10	19	80
49	16	10	42	5	98	77	80	38	43
82	81	65	24	38	1	99	54	15	53
95	77	25	47	50	78	77	48	37	55
12	87	37	42	65	67	40	3	7	5
25	25	6	31	68	49	26	53	1	1

### Suma

El resultado de la suma A+B es:									
64	98	169	116	78	90	78	172	66	76
115	108	59	41	152	66	137	65	131	185
58	138	134	59	116	195	83	154	82	8
131	52	13	137	124	154	95	142	45	111
99	61	32	89	87	35	59	32	65	133
137	27	15	95	9	118	109	178	100	56
85	86	80	121	70	5	185	131	15	90
163	147	96	140	136	146	128	128	123	65
32	159	124	89	90	68	65	57	7	7
31	79	94	47	141	135	42	126	19	79

### Resta

El resultado de la resta A-B es:

-12	64	31	-10	-36	6	-78	26	-38	-8
29	38	7	-21	-14	-42	-19	-25	1	13
-28	-16	-28	33	82	1	51	22	8	6
-15	0	-7	-25	-12	0	11	34	45	-29
-9	-3	-16	-81	3	-17	5	12	27	-27
39	-5	-5	11	-1	-78	-45	18	24	-30
-79	-76	-50	73	-6	3	-13	23	-15	-16
-27	-7	46	46	36	-10	-26	32	49	-45
8	-15	50	5	-40	-66	-15	51	-7	-3
-19	29	82	-15	5	37	-10	20	17	77

### Multiplicación

El resultado de la multiplicacion A\*B es:

26549	23486	16711	19838	22803	34823	25502	25651	16335	21161
22343	23516	18786	22604	28626	24799	27640	22500	13663	20727
33547	28696	18992	27633	24942	35074	34318	27000	16894	29906
28607	22616	15151	25598	24187	29386	31282	24740	13214	24301
13055	13697	10367	13742	16225	14019	16056	12235	7387	11848
22100	19949	14491	19804	20916	23387	25208	19058	10865	16946
25271	18811	11067	18114	18435	18449	21515	17613	5972	18923
36476	33354	24494	33982	33377	42127	38273	31504	19268	32804
19657	17830	14366	13797	16632	21869	17979	17358	11827	16749
26071	23329	15717	21298	21636	34000	25306	25706	14878	20673

### Transpuesta

La transpuesta de la matriz A es:

26	72	15	58	45	88	3	68	20	6
81	73	61	26	29	11	5	70	72	54
100	33	53	3	8	5	15	71	87	88
53	10	46	56	4	53	97	93	47	16
21	69	99	56	45	4	32	86	25	73
48	12	98	77	9	20	4	68	1	86
0	59	67	53	32	32	86	51	25	16
99	20	88	88	22	98	77	80	54	73
14	66	45	45	46	62	0	86	0	18
34	99	7	41	53	13	37	10	2	78

La transpuesta de la matriz B es:

38	43	43	73	54	49	82	95	12	25
17	35	77	26	32	16	81	77	87	25
69	26	81	10	24	10	65	25	37	6
63	31	13	81	85	42	24	47	42	31
57	83	17	68	42	5	38	50	65	68
42	54	97	77	26	98	1	78	67	49
78	78	16	42	27	77	99	77	40	26
73	45	66	54	10	80	54	48	3	53
52	65	37	0	19	38	15	37	7	1
42	86	1	70	80	43	53	55	5	1

### Inversa

La inversa de la matriz A es:

-0.02	-0.00	-0.02	0.02	-0.01	0.00	-0.01	0.00	0.03	0.01
0.03	0.00	0.01	0.01	0.01	-0.02	-0.00	-0.01	-0.01	-0.02
-0.02	0.02	-0.00	-0.02	-0.04	0.02	-0.00	0.00	0.01	0.02
0.01	-0.01	-0.01	0.01	0.01	-0.01	0.01	0.01	-0.00	-0.01
-0.01	-0.04	-0.01	0.02	0.08	-0.02	-0.00	0.00	0.02	-0.00
-0.01	0.02	0.00	0.00	-0.05	0.01	-0.00	0.00	-0.01	0.01
-0.02	0.03	0.01	-0.02	-0.06	0.02	0.00	-0.01	-0.00	0.01
0.01	-0.02	0.01	-0.00	0.04	0.00	0.00	-0.01	-0.00	-0.01
0.02	0.01	0.01	-0.03	-0.01	0.01	0.01	0.01	-0.03	-0.00
0.01	0.00	-0.01	-0.00	0.00	-0.00	0.00	-0.00	-0.01	0.00

La inversa de la matriz B es:

0.01	-0.01	-0.00	0.01	-0.01	-0.02	-0.01	0.02	-0.01	-0.01
-0.02	0.00	0.00	-0.03	0.02	0.01	0.01	-0.01	0.00	0.02
0.01	-0.00	0.01	0.02	-0.02	-0.01	0.00	-0.00	0.00	-0.02
0.00	-0.01	-0.00	-0.01	0.02	0.01	-0.00	0.00	0.00	0.01
0.00	0.01	-0.00	0.01	-0.01	-0.01	-0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.02	-0.02	-0.00	-0.01	0.00	0.01	-0.02
0.01	-0.00	-0.01	0.01	-0.01	0.00	0.00	0.00	0.01	-0.01
-0.01	0.00	0.01	-0.02	0.02	0.01	0.01	-0.02	-0.01	0.03
0.00	0.00	0.00	-0.03	0.01	0.00	-0.01	0.01	-0.01	0.01
-0.01	0.01	0.00	0.01	0.01	0.01	0.01	-0.02	-0.00	-0.00

## CONCLUSION

En conclusión, hemos logrado un nivel avanzado de comprensión sobre los mecanismos de sincronización de procesos en los sistemas operativos Linux y Windows, conocimientos esenciales para el desarrollo de aplicaciones multiproceso en entornos donde múltiples tareas compiten por recursos compartidos.

A lo largo de esta experiencia, profundizamos en el estudio detallado de los semáforos, un componente clave en la sincronización de procesos. En el caso de Linux, exploramos funciones como `semget()` y `semop()`, responsables de la creación y gestión de semáforos. Por otro lado, en Windows, examinamos funciones como `CreateSemaphore()`, `OpenSemaphore()` y `ReleaseSemaphore()`, las cuales cumplen roles equivalentes adaptados a este entorno operativo.

La ejecución práctica de programas nos permitió observar de manera tangible cómo los semáforos actúan como mediadores, controlando el acceso a secciones críticas de código. Esta experiencia reforzó los conceptos teóricos, ayudándonos a comprender la relevancia de estos mecanismos en la prevención de problemas como las condiciones de carrera y los bloqueos mutuos (deadlocks).

Además, aplicamos estos conocimientos en el desarrollo de aplicaciones que hacen uso de memoria compartida, destacando el papel fundamental de los semáforos para garantizar la integridad de los datos y la sincronización eficiente entre procesos de lectura y escritura. Esta práctica nos permitió experimentar de primera mano cómo los semáforos se utilizan estratégicamente para construir aplicaciones multiproceso cooperativas, estables y eficaces.

En resumen, este aprendizaje no solo nos ha proporcionado una base sólida en la sincronización de procesos, sino que también nos ha preparado para enfrentar los desafíos del desarrollo de software en entornos complejos y altamente competitivos.