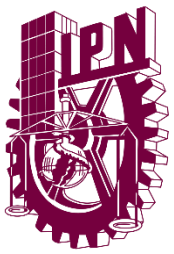

Instituto Politécnico Nacional
Escuela Superior de Cómputo
"ESCOM"

Unidad de Aprendizaje:
Cómputo Paralelo

Práctica No. 4
Sincronización de programas de cómputo paralelo usando memoria compartida

Integrantes:
Sánchez De Los Ríos Flavio Josué - 6BV1
Tinoco Videgaray Sergio Ernesto - 6BV1
Ibarra González Emilio Francisco- 6BV1

Maestro:
Jiménez Benítez José Alfredo

Fecha de Entrega: 07/05/23

Resumen

En el cómputo paralelo existen múltiples herramientas que facilitan el uso de los recursos utilizados en la paralelización de procesos como es el caso de los semáforos que nos van a permitir aplicar cierta sincronicidad entre los diferentes procesos que intentan acceder a la misma región de memoria, o también denominado “memoria crítica”.

El lenguaje de programación C, nos permite trabajar directamente con la memoria principal del sistema o memoria RAM, esto por medio del uso de variables como lo son los apuntadores que nos van a permitir justamente generar un espacio en memoria que almacena una dirección de la misma memoria donde se encuentra almacenada otra variable a la que estamos haciendo referencia con el apuntador. Además de esto, C nos permite el uso de los semáforos para mantener una sincronización entre los diferentes procesos del sistema que intenten acceder a la región crítica de memoria.

En el desarrollo de esta práctica se va a codificar un programa que genere sus propias regiones de memoria compartida con el fin de almacenar una matriz de números los cuales van a sumarse por filas y se van a almacenar en un arreglo de resultados de cada fila. En este caso se va a trabajar con una matriz de 3 filas por lo que con ayuda de 3 procesos independientes (A, B y C) se va a realizar la suma de cada fila dentro de la matriz y haciendo uso de los semáforos, se va a mantener la sincronicidad entre los distintos procesos que intenten acceder a un elemento dentro de la matriz almacenada en la región crítica de la memoria compartida.

Índice

Introducción..... 1
Marco teórico 2
Desarrollo..... 7
Conclusiones..... 19
Referencias..... 22
Anexo. Códigos completos. 23

Introducción

En el cómputo paralelo existe una herramienta que nos ayuda a acceder a cierta información en cada proceso que se está ejecutando, dicha herramienta es denominada “memoria compartida” y se denomina de tal forma debido a que está siendo utilizada por otros procesos que acceden a ella con cierta concurrencia, debido a esto surgen algunas problemáticas a la hora de intentar acceder a este espacio de memoria compartida como una posible incoherencia en los datos almacenados en la misma, o la denominada “exclusión mutua” que ocurre cuando dos o más procesos intentan acceder a esta región de la memoria al mismo tiempo, causando que dicha región de memoria entre en un estado de “bloqueo de segmento”, es por ello que es necesario aplicar cierta sincronicidad a la hora de acceder a estos espacios de memoria y a causa de esto han surgido distintos algoritmos y métodos que nos permiten organizar la manera en que distintos procesos acceden a una misma región de memoria compartida, uno de estos métodos son los denominados “semáforos” los cuales nos permiten alternar el acceso de los diferentes procesos que se ejecutan.

Los semáforos son una herramienta importante en la programación concurrente para controlar el acceso a recursos compartidos por múltiples procesos o hilos [1]. En el lenguaje de programación C, los semáforos se implementan a través de la biblioteca de sincronización de hilos (pthread.h) o a través de la biblioteca de semáforos del sistema (semaphore.h). Los semáforos permiten que los procesos o hilos esperen o se bloqueen hasta que un recurso esté disponible, evitando así la competencia por recursos y las condiciones de carrera. En este contexto, es importante comprender cómo funcionan los semáforos, cómo se declaran y utilizan en C, y cómo se pueden aplicar para crear programas eficientes y seguros. En esta introducción, explicaremos los conceptos básicos de los semáforos en C y cómo se pueden utilizar para gestionar recursos compartidos.

El objetivo de esta práctica se centra en aplicar el uso de semáforos para la sincronización de distintos procesos que accedan a la misma región de memoria compartida, así como comprender su funcionamiento dentro de los sistemas de cómputo paralelo.

Para la elaboración de esta práctica se hizo uso del lenguaje de programación C. C es un lenguaje de programación de propósito general el cual permite la comunicación directa con la computadora y provee herramientas útiles para el cómputo paralelo [2]. Estas están dadas por librerías las cuales siguen el estándar POSIX, el cual define una interfaz estándar del sistema operativo y el entorno [3]. Esto conlleva a la necesidad de comprender y definir elementos fundamentales del lenguaje C y de los sistemas operativos, para el desarrollo de programas paralelos.

Marco teórico

Tipos de datos en C

Para el cómputo paralelo el almacenamiento de datos es fundamental [4]. Para el almacenamiento y representación de datos, C define distintas formas, cada una con sus especificaciones y ventajas. En general, los tipos de datos se pueden clasificar en la tabla 1 [4].

Básicos	Son tipos aritméticos y consisten de dos tipos: enteros y de punto flotantes.
Enumerados	Son tipos aritméticos y se usan para definir variables con ciertos valores enteros discretos a lo largo del programa.
Vacíos	Indican variables cuyo valor no está disponible.
Derivados	Incluyen tipos compuestos por otros como arreglos, estructuras, uniones y funciones.

Tabla 1. Clasificación de tipos de datos en C.

Los tipos de datos enteros se utilizan para representar números enteros sin decimales. Existen varios tipos de datos enteros, que se diferencian por el número de bits que utilizan para representar los valores y el rango de valores que pueden representar. Algunos de los tipos de datos enteros más comunes son: "int" (entero de 32 bits), "short" (entero de 16 bits), "long" (entero de 64 bits) y "char" (entero de 8 bits utilizado para representar caracteres).

Los tipos de datos de punto flotante se utilizan para representar números con decimales. Los tipos de punto flotante incluyen "float" (32 bits) y "double" (64 bits).

Los tipos de datos estructurados se utilizan para representar objetos o estructuras de datos complejos. En C, se pueden definir estructuras de datos utilizando la palabra clave "struct", que permite definir un conjunto de variables con diferentes tipos de datos.

Los tipos de datos enumerados se utilizan para definir un conjunto de valores enteros constantes que se pueden utilizar en lugar de valores numéricos en el código. Los tipos de datos enumerados son útiles porque permiten que el código sea más legible y fácil de entender al reemplazar los valores numéricos con nombres descriptivos.

Un arreglo es una estructura de datos que almacena un conjunto de elementos del mismo tipo de datos en una secuencia continua en memoria. Los arreglos se utilizan comúnmente para almacenar colecciones de datos del mismo tipo, como números enteros, caracteres, flotantes, etc.

Apuntadores

Los apuntadores son un concepto fundamental en C. Un apuntador contiene la dirección de memoria de una variable [5]. Esto permite un control de memoria más avanzado ya que la manipulación de esta se hace de manera directa y permite la creación de estructuras dinámicas. Al contener la dirección de memoria de una variable cualquier proceso que conozca esta información, será capaz de acceder y/o modificar la información contenida. La figura 1.1 ilustra los apuntadores en C.

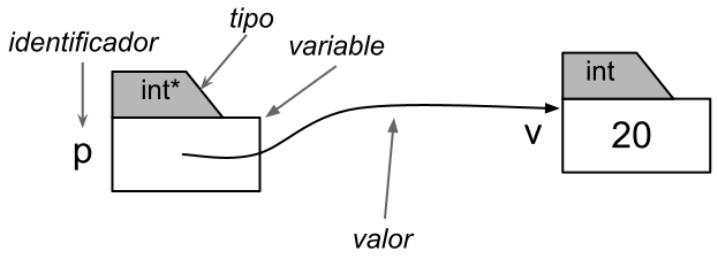

Figura 1. Diagrama de la función de un apuntador.

Los apuntadores se definen utilizando el operador "&", que se utiliza para obtener la dirección de una variable. Por ejemplo, si tenemos una variable llamada "x", podemos obtener su dirección de memoria utilizando "&x". El resultado de esta operación es un apuntador que apunta a la dirección de memoria de la variable "x".

Una vez que tenemos un apuntador, podemos usar el operador "*" para acceder al valor almacenado en la dirección de memoria apuntada por el apuntador. Por ejemplo, si tenemos un apuntador llamado "p" que apunta a la dirección de memoria de la variable "x", podemos obtener el valor de "x" utilizando "*p". Esta operación devuelve el valor almacenado en la dirección de memoria apuntada por "p".

Estructuras de datos

Una estructura de datos es una colección de variables que se agrupan bajo un mismo nombre para formar una única entidad lógica [6]. Las estructuras de datos en C se utilizan para almacenar y organizar datos relacionados de forma más conveniente y eficiente. Se pueden utilizar para almacenar conjuntos de datos relacionados, pasar argumentos a funciones, implementar estructuras de datos complejas, almacenar datos en archivos y comunicar datos entre procesos. Las estructuras de datos son una herramienta poderosa en la programación en C y son esenciales para el desarrollo de aplicaciones complejas.

Archivos

En C, los archivos son utilizados para leer y escribir datos en dispositivos de almacenamiento permanente, como discos duros, memorias USB, entre otros [4]. Los archivos pueden ser utilizados para almacenar datos de manera persistente y para compartir datos entre diferentes programas. Se utilizan tres tipos de archivos [4] archivos de texto, archivos binarios y archivos de acceso aleatorio.

Los archivos de texto son aquellos que contienen datos legibles por humanos, como texto plano. Se pueden abrir y leer usando funciones de la biblioteca estándar de C, como "fopen", "fread" y "fwrite".

Los archivos binarios contienen datos en un formato no legible por humanos, como datos binarios, imágenes, sonidos, entre otros. Se pueden leer y escribir usando las mismas funciones que se utilizan para archivos de texto.

Los archivos de acceso aleatorio permiten el acceso aleatorio a los datos almacenados en el archivo. Esto significa que se pueden leer y escribir datos en cualquier posición dentro del archivo. Se pueden usar funciones como "fseek" y "ftell" para moverse por el archivo y determinar la posición actual.

Procesos

En el cómputo paralelo, un proceso es una unidad de ejecución que realiza una tarea específica. Los procesos se relacionan con el cómputo paralelo porque se utilizan para dividir una tarea compleja en subtareas más pequeñas y ejecutarlas en paralelo en múltiples núcleos de procesamiento [5]. De esta manera, se puede lograr un aumento significativo en la velocidad de procesamiento y reducir el tiempo de respuesta.

El principal concepto en cualquier sistema operativo es el de proceso, un proceso es un programa en ejecución, incluyendo el valor de los registros y las variables [5]. Cuando se inicia un proceso en un sistema, el proceso utiliza una parte de los recursos disponibles en el sistema. Cuando está ejecutándose más de un proceso, un planificador que está incorporado al sistema operativo proporciona a cada proceso su parte de tiempo del sistema, basándose en las prioridades establecidas.

En la siguiente tabla se describen los tipos de procesos [5].

Procesos en primer plano y en segundo plano	Son procesos que necesitan que un usuario los inicie o que interactúe con ellos se denominan procesos en primer plano. Los procesos que se ejecutan con independencia de un usuario se denominan procesos en segundo plano.
Procesos daemon	Son procesos que se ejecutan de forma desatendida. Están constantemente en segundo plano y siempre están disponibles. Los daemons suelen iniciarse cuando se arranca el sistema y se ejecutan hasta que se detiene el sistema.

Procesos zombie	Es un proceso finalizado que ya no se ejecuta pero que sigue reconociéndose en la tabla de procesos (en otras palabras, tiene un número PID). Ya no se asigna espacio del sistema a dicho proceso.

Tabla 1.2. Tipos de procesos en un sistema operativo.

Memoria compartida

La memoria compartida en la computación paralela se refiere a un tipo de arquitectura de memoria en la que varias CPU o núcleos de procesamiento tienen acceso a una misma memoria física compartida. Esto significa que todos los procesadores pueden leer y escribir datos en la misma dirección de memoria.

Este enfoque de memoria compartida es común en sistemas multiprocesador y multicomputador, donde varias CPU o núcleos de procesamiento necesitan trabajar juntos en una tarea. Al tener acceso a una misma memoria compartida, se elimina la necesidad de copiar datos entre procesadores, lo que puede acelerar significativamente la velocidad de procesamiento.

Regiones Críticas

Las regiones críticas determinan una forma de prohibir que más de un proceso lea o escriba en los datos compartidos en un mismo tiempo [6]. Se pueden ver como un estado del proceso en el momento que se está usando el recurso compartido.

Para evitar problemas en situaciones relacionadas con recursos compartidos, la clave es determinar una forma de prohibir que más de un proceso lea o escriba en los datos compartidos a la vez, en otras palabras, lo que se necesita es una forma de garantizar que si un proceso utiliza una variable o archivo compartido, los demás procesos no puedan utilizarlos. A esto se le llama exclusión mutua. Se necesitan 4 condiciones [6] para tener una buena solución. Los cuales son los siguientes:

1) Dos procesos no deben encontrarse al mismo tiempo dentro de sus secciones críticas.

2) No se debe hacer hipótesis sobre la verdad o el número de procesadores.

3) Ninguno de los procesos que estén en ejecución fuera de su sección crítica puede bloquear a otros procesos.

4) Ningún proceso debe esperar demasiado tiempo para entrar en su sección crítica.

Si los procesos comparten recursos no deben estar en su sección crítica al mismo tiempo, ya que puede provocar un conflicto. Si ambos procesos son independientes no debería haber problema, si un proceso desea entrar a su sección crítica no deben hacerlo frecuentemente, para acceder al recurso compartido.

Semáforos

Los semáforos son una herramienta de sincronización que ofrece una solución al problema de la sección crítica. Un semáforo bloquea un proceso cuando éste no puede realizar la operación deseada, en vez de desperdiciar tiempo de CPU [6].Un semáforo provee una simple pero útil abstracción para controlar el acceso de múltiples procesos a un recurso común en el cómputo paralelo.

Hay dos API de semáforos comunes en los sistemas basados en UNIX: semáforos POSIX y semáforos System V. Se considera que este último tiene una interfaz más simple y ofrece las mismas características que la API POSIX. Un semáforo es un número entero mantenido por el kernel, generalmente establecido en el valor inicial mayor o igual a 0 [7].

Se pueden realizar dos operaciones en un objeto semáforo: incrementar o disminuir en uno, lo que corresponde a adquirir y liberar el recurso compartido. POSIX proporciona un tipo especial sem_t para un semáforo sin nombre, una herramienta más común en flujos de trabajo de subprocesos múltiples. La variable sem_t debe inicializarse con la función sem_init que también indica si el semáforo dado debe compartirse entre procesos o subprocesos de un proceso [7].

Una vez inicializada la variable,se implementa la sincronización mediante las funciones sem_post y sem_wait. sem_post incrementa el semáforo, que normalmente corresponde al desbloqueo del recurso compartido. Por el contrario, sem_wait disminuye el semáforo y denota el bloqueo del recurso. Por lo tanto, la sección crítica debería comenzar con sem_wait y terminar con la llamada sem_post.

Desarrollo

Primeramente, se van a generar los archivos para cada uno de los procesos (A, B, C y el principal).

Para ello se va a abrir una ventana de la terminal dentro de la carpeta de la práctica (figura 2).

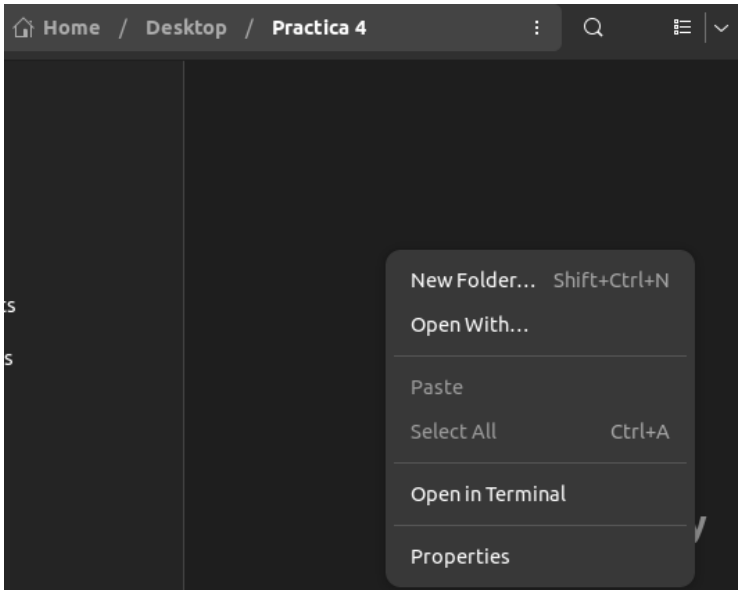

Figura 2. Vista de la carpeta Practica 4.

Dentro de la terminal se van a crear los archivos correspondientes para cada uno de los procesos que van a realizar la suma de los elementos de cada fila a través del comando touch (figura 3).

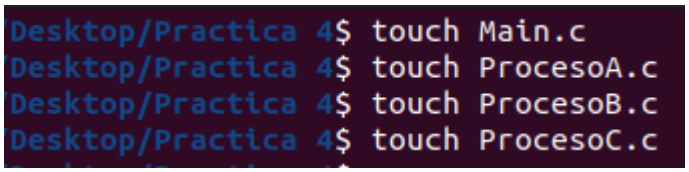

Figura 3 Creación de los archivos para cada uno de los procesos.

Una vez ejecutados los comandos se verifica que se hayan creado todos los archivos (figura 4).

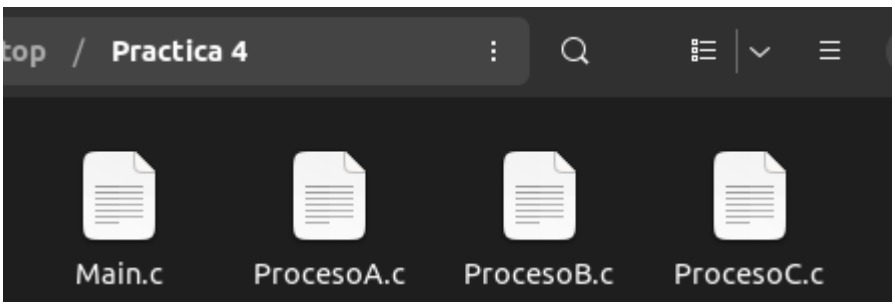

Figura 4 Ventana de la carpeta Práctica 4 con los archivos creados.

Una vez creados los archivos, se van a programar cada uno de los procesos correspondientes.

- Proceso Principal.

Para el proceso principal (llamado archivo “Main”), se van a declarar las variables que nos van a permitir gestionar los espacios de memoria para la matriz que se va a almacenar dentro de la memoria compartida, así como las variables auxiliares para la creación de los espacios de memoria (figura 5).

```
int shmid0, shmid1, i,j;
int *resultados, *datos_matriz;
int **app_mat;
int matriz[3][9]={ {1, 2, 3, 4, 5, 6, 7, 8, 9},
                   {1, 3, 5, 7, 9, 11, 13, 15, 17},
                   {2, 4, 6, 8, 10, 12, 14, 16, 18 }};
key_t llave0, llave1;
pid_t pid;
```

Figura 5. Código del programa Main donde se declaran las variables y se define la matriz de números que se va a almacenar en la memoria compartida.

Posteriormente se van a inicializar las llaves que van a definir cada región de la memoria compartida a través de la función “ftok” [8].

Una vez definidas las llaves se va a definir el espacio de la memoria compartida a manera de arreglos unidimensionales (vectores) por medio de la función “shmget” la cual va a retornar los ID’s correspondientes a cada segmento de la memoria compartida [9].

Nótese que el primer segmento de memoria compartida corresponde con la matriz numérica y en este caso se va a definir por medio de un vector de 3x9 (osease 27 elementos). Mientras que el segundo segmento de la memoria compartida se va a definir como un vector de únicamente 3 elementos para cada resultado obtenido por los procesos A, B y C.

Finalmente, y haciendo uso de la función “shmat” se va a asociar cada segmento de la memoria compartida con una variable de tipo apuntador para poder hacer referencia a ese espacio de memoria (figura 6) [10].

```
llave0=ftok("Archivo1",'k');
llave1=ftok("Archivo2",'k');

shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);

datos_matriz=(int*)shmat(shmid0,0,0);
resultados=(int *)shmat(shmid1,0,0);
```

Figura 6. Código del programa Main donde se definen los segmentos de memoria compartida.

En el paso previo se definió el espacio de memoria para la matriz a manera de vector, en este caso se va a mapear dicho espacio de memoria compartida a un apuntador doble que se va a comportar como una matriz. Para ello, primeramente, se va a reservar el espacio de memoria correspondiente con las 3 filas para dicho apuntador doble y posterior a eso, dentro de un bucle for se va a asignar el espacio en memoria de cada fila correspondiente a cada elemento múltiplo de 9, puesto que la matriz que se va a utilizar consta de 9 columnas.

De tal forma que se obtendrá una matriz de 3 filas por 9 columnas (figura 7).

```
app_mat=(int**)malloc(3*sizeof(app_mat[0]));

for(i=0;i<3;i++)
  {
     app_mat[i]=datos_matriz+i*9;
  }
```

Figura 7. Mapeo del vector de memoria compartida a un apuntador doble.

Posteriormente se va a asignar cada elemento de la matriz numérica a la memoria compartida a través del apuntador doble que se definió previamente (figura 8).

```
for(i=0;i<3;i++)
  {
     for(j=0;j<9;j++)
    {
        app_mat[i][j]=matriz[i][j];
    }
  }
```

Figura 8. Asignación de la matriz numérica a la memoria compartida.

Una vez definida la matriz de números dentro del segmento de memoria compartida, el programa va a estar a la espera de los resultados por los otros procesos, para ello se va a definir un bucle while que va a evaluar el valor almacenado dentro del arreglo de resultados. Una vez que los resultados de todos los procesos se hayan registrado en el arreglo, el programa va a salir del bucle while, para finalmente imprimir los resultados almacenados en el arreglo (figura 9).

```
while(resultados[0]==1 || resultados[1]==2 || resultados[2]==3);

//Evalua si ya se registraron todos los resultados de las sumas
if(resultados[0]!=1 && resultados[1]!=2 && resultados[2]!=3)
{
 for(i=0;i<3;i++)
  {
     printf("Suma Fila %d: %d\n",i+1,resultados[i]); //Imprime los resultados de cada fila
  }
}
```

Figura 9. Código del programa Main donde se espera a que los otros procesos registren su resultado dentro del arreglo correspondiente.

Finalmente, por medio de la función "shmdt" se van a disociar los segmentos de memoria compartida de los apuntadores [11] y posteriormente se van a eliminar dichos segmentos de memoria compartida a través de la función "shmctl" [12] como se muestra en la figura 10.

```
shmdt(&app_mat);
shmdt(&resultados);
shmctl(shmid0,IPC_RMID,0);
shmctl(shmid1,IPC_RMID,0);
return 0;
```

Figura 10. Destrucción de los segmentos de memoria existentes.

Procesos en Paralelo

Para los procesos A, B y C se va a crear e importar un archivo de cabecera (extensión .h) que va a contener las funciones que les van a permitir hacer uso de los semáforos (figura 11).

```
~/Desktop/Practica 4$ touch gestionarSemaforos.h
~/Desktop/Practica 4$
```

Figura 11. Creación del archivo de cabecera para gestionar los semáforos.

Dentro de este archivo de cabecera se va a utilizar la biblioteca sem.h para el uso de los semáforos (figura 12).

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

#define PERMISOS 0644
```

Figura 12. Código de las bibliotecas a utilizar dentro del archivo gestionarSemaforos.h

Se define la función para generar los semáforos, la cual recibe como parámetros la llave del semáforo correspondiente y un valor inicial para dicho semáforo. Dentro de dicha función se va a hacer uso de la función semget para obtener una ID del semáforo a generar y posteriormente se evalúa si dicha ID fue generada correctamente [13]. Posteriormente se va a inicializar dicho semáforo con el valor inicial dado a través de la funcion semctl [13]. Finalmente se retorna la ID del semáforo creado (figura 13).

```
int Crea_semaforo(key_t llave,int valor_inicial)
{
   int semid=semget(llave,1,IPC_CREAT|PERMISOS);
   if(semid==-1)
   {
      return -1;
   }
   semctl(semid,0,SETVAL,valor_inicial);
   return semid;
}
```

Figura 13. Código de la función que crea el semáforo.

Posteriormente se define la función que va a establecer un estado bajo para el semáforo que se le indique por medio de la ID, la cual se va a pasar como argumento para dicha función esto a través de la función semop que aplica una operación con el semaforo [15] (figura 14).

```
void down(int semid)
{
   struct sembuf op_p[]={0,-1,0};
   semop(semid,op_p,1);
}
```

Figura 14. Código de la función que establece un estado bajo para el semáforo.

De igual forma se define la función que va a establecer un estado alto para el semáforo indicado por su ID (figura 15).

```
void up(int semid)
{
   struct sembuf op_v[]={0,+1,0};
   semop(semid,op_v,1);
}
```

Figura 15. Código de la función que establece un estado alto en el semáforo.

- Proceso A.

Para el proceso A se van a declarar e inicializar todas las variables que se declararon previamente en el proceso Main (figura 16).

```
int shmid0, shmid1, i,j,suma=0;
int *resultados, *datos_matriz;
int **app_mat;
key_t llave0, llave1;
pid_t pid;

llave0=ftok("Archivo1",'k');
llave1=ftok("Archivo2",'k');

shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);

datos_matriz=(int*)shmat(shmid0,0,0);
resultados=(int *)shmat(shmid1,0,0);

//Reservar memoria
app_mat=(int**)malloc(3*sizeof(app_mat[0]));

for(i=0;i<3;i++)
  {
     app_mat[i]=datos_matriz+i*9;
  }
```

Figura 16. Código del proceso A con la declaración de las variables.

Se generan las llaves para los semáforos, como estamos trabajando con dos regiones de memoria compartida (una para la matriz numérica y otra para los resultados) se van a generar dos semáforos (figura 17).

```
//Generar las llaves para los semaforos
llave_sem1=ftok("S1",'k');
llave_sem2=ftok("S2",'k');

//Crear semaforos
semaforo_matriz=Crea_semaforo(llave_sem1,1);
semaforo_resultados=Crea_semaforo(llave_sem2,1);
```

Figura 17. Código de la generación de los semáforos.

Para este caso se va a realizar la suma de la primera fila por lo que se recorre la matriz almacenada en el segmento de la memoria compartida indicado por el índice 0 para la fila 1. Haciendo uso de las funciones down y up del archivo de cabecera “gestionarSemaforos.h” se va a sincronizar el acceso al elemento j de la fila correspondiente en nuestra matriz de números almacenada en la región de memoria compartida (figura 18).

```
for(int j=0;j<9;j++)
{
   down(semaforo_matriz);
   sleep(1);
   suma+=app_mat[0][j];
   up(semaforo_matriz);
}
```

Figura 18. Código del proceso A con la suma de los elementos de la fila 1 de la matriz haciendo uso de semáforos.

Se imprime el valor resultante de la suma junto con su ID de proceso (figura 19).

```
printf("\n\nEl proceso %d, termino de sumar la fila 1\n\n",getpid());
printf("Suma de la primer fila: %d\n",suma);
```

Figura 19. Código del proceso A con la impresión del resultado y su ID de proceso asignado

Nuevamente haciendo uso de los semáforos se va a registrar el resultado de la suma hecha previamente en el arreglo de resultados que se encuentra almacenado en otra región de memoria compartida (figura 20).

```
down(semaforo_resultados);
resultados[0]=suma;
up(semaforo_resultados);
```

Figura 20. Código del proceso A donde se registra el resultado de la suma haciendo uso de semáforos.

Finalmente se disocian los elementos del espacio de memoria compartida creado para este proceso. (figura 21)

```
shmdt(&app_mat);
shmdt(&resultados);
shmctl(shmid0,IPC_RMID,0);
shmctl(shmid1,IPC_RMID,0);
return 0;
```

Figura 21. Código del proceso A con la disociación de los apuntadores al espacio de memoria compartida.

- Proceso B.

De igual forma para el proceso B se van a declarar e inicializar todas las variables que se declararon previamente en el proceso Main (figura 22).

```
int shmid0, shmid1, i,j,suma=0;
int *resultados, *datos_matriz;
int **app_mat;
key_t llave0, llave1;
pid_t pid;

llave0=ftok("Archivo1",'k');
llave1=ftok("Archivo2",'k');

shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);

datos_matriz=(int*)shmat(shmid0,0,0);
resultados=(int *)shmat(shmid1,0,0);

//Reservar memoria
app_mat=(int**)malloc(3*sizeof(app_mat[0]));

for(i=0;i<3;i++)
  {
     app_mat[i]=datos_matriz+i*9;
  }
```

Figura 22. Código del proceso B con la declaración de las variables.

Se generan las llaves para los semáforos, como estamos trabajando con dos regiones de memoria compartida (una para la matriz numérica y otra para los resultados) se van a generar dos semáforos (figura 23).

```
//Generar las llaves para los semaforos
llave_sem1=ftok("S1",'k');
llave_sem2=ftok("S2",'k');

//Crear semaforos
semaforo_matriz=Crea_semaforo(llave_sem1,1);
semaforo_resultados=Crea_semaforo(llave_sem2,1);
```

Figura 23. Código de la generación de los semáforos.

Para este caso se va a realizar la suma de la segunda fila por lo que se recorre la matriz almacenada en el segmento de la memoria compartida indicado por el índice 1 para la fila 2. Haciendo uso de las funciones down y up del archivo de cabecera "gestionarSemaforos.h" se va a sincronizar el acceso al elemento j de la fila correspondiente en nuestra matriz de números almacenada en la región de memoria compartida (figura 24).

```
for(int j=0;j<9;j++)
{
   down(semaforo_matriz);
   sleep(1);
   suma+=app_mat[1][j];
   up(semaforo_matriz);
}
```

Figura 24. Código del proceso B con la suma de los elementos de la fila 2 de la matriz haciendo uso de semáforos.

Se imprime el valor resultante de la suma junto con su ID de proceso (figura 25).

```
printf("\n\nEl proceso %d, termino de sumar la fila 2\n\n",getpid());
printf("Suma de la segunda fila: %d\n",suma);
```

Figura 25. Código del proceso B con la impresión del resultado y su ID de proceso asignado

Nuevamente haciendo uso de los semáforos se va a registrar el resultado de la suma hecha previamente en el arreglo de resultados que se encuentra almacenado en otra región de memoria compartida (figura 26).

```
down(semaforo_resultados);
resultados[1]=suma;
up(semaforo_resultados);
```

Figura 26. Código del proceso B donde se registra el resultado de la suma haciendo uso de semáforos.

Finalmente se disocian los elementos del espacio de memoria compartida creado para este proceso. (figura 27)

```
shmdt(&app_mat);
shmdt(&resultados);
shmctl(shmid0,IPC_RMID,0);
shmctl(shmid1,IPC_RMID,0);
return 0;
```

Figura 27. Código del proceso B con la disociación de los apuntadores al espacio de memoria compartida.

- Proceso C.

De igual forma para el proceso C se van a declarar e inicializar todas las variables que se declararon previamente en el proceso Main (figura 28).

```
int shmid0, shmid1, i,j,suma=0;
int *resultados, *datos_matriz;
int **app_mat;
key_t llave0, llave1;
pid_t pid;

llave0=ftok("Archivo1",'k');
llave1=ftok("Archivo2",'k');

shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);

datos_matriz=(int*)shmat(shmid0,0,0);
resultados=(int *)shmat(shmid1,0,0);

//Reservar memoria
app_mat=(int**)malloc(3*sizeof(app_mat[0]));

for(i=0;i<3;i++)
  {
     app_mat[i]=datos_matriz+i*9;
  }
```

Figura 28. Código del proceso C con la declaración de las variables.

Se generan las llaves para los semáforos, como estamos trabajando con dos regiones de memoria compartida (una para la matriz numérica y otra para los resultados) se van a generar dos semáforos (figura 29).

```
//Generar las llaves para los semaforos
llave_sem1=ftok("S1",'k');
llave_sem2=ftok("S2",'k');

//Crear semaforos
semaforo_matriz=Crea_semaforo(llave_sem1,1);
semaforo_resultados=Crea_semaforo(llave_sem2,1);
```

Figura 29. Código de la generación de los semáforos.

Para este caso se va a realizar la suma de la tercera fila por lo que se recorre la matriz almacenada en el segmento de la memoria compartida indicado por el índice 2 para la fila 3. Haciendo uso de las funciones down y up del archivo de cabecera "gestionarSemaforos.h" se va a sincronizar el acceso al elemento j de la fila correspondiente en nuestra matriz de números almacenada en la región de memoria compartida (figura 30).

```
for(int j=0;j<9;j++)
{
   down(semaforo_matriz);
   sleep(1);
   suma+=app_mat[2][j];
   up(semaforo_matriz);
}
```

Figura 30. Código del proceso C con la suma de los elementos de la fila 3 de la matriz haciendo uso de semáforos.

Se imprime el valor resultante de la suma junto con su ID de proceso (figura 31).

```
printf("\n\nEl proceso %d, termino de sumar la fila 3\n\n",getpid());
printf("Suma de la tercer fila: %d\n",suma);
```

Figura 31. Código del proceso C con la impresión del resultado y su ID de proceso asignado

Nuevamente haciendo uso de los semáforos se va a registrar el resultado de la suma hecha previamente en el arreglo de resultados que se encuentra almacenado en otra región de memoria compartida (figura 32).

```
down(semaforo_resultados);
resultados[2]=suma;
up(semaforo_resultados);
```

Figura 32. Código del proceso C donde se registra el resultado de la suma haciendo uso de semáforos.

Finalmente se disocian los elementos del espacio de memoria compartida creado para este proceso. (figura 33)

```
shmdt(&app_mat);
shmdt(&resultados);
shmctl(shmid0,IPC_RMID,0);
shmctl(shmid1,IPC_RMID,0);
return 0;
```

Figura 33. Código del proceso C con la disociación de los apuntadores al espacio de memoria compartida.

- Resultados

Para la ejecución de los procesos se van a abrir 4 ventanas de la terminal (1 para el proceso principal y 3 para los procesos A, B y C) dentro de la carpeta de la práctica 4.Una vez creadas las 4 ventanas de la terminal se va a compilar cada archivo por separado como se muestra en la figura 34.

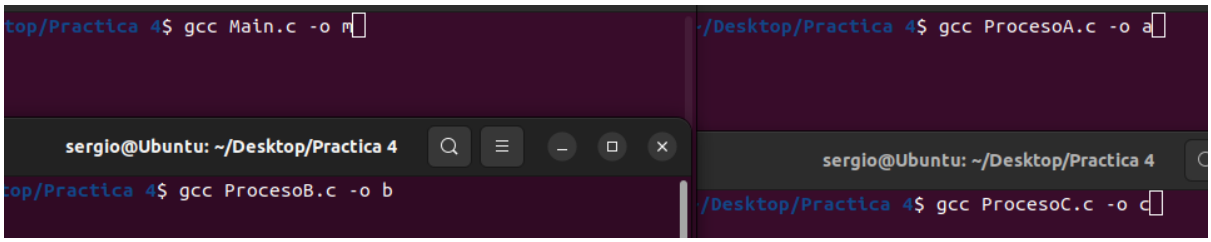

Figura 34. Compilación de los programas en cada ventana de la terminal.

Finalmente se van a ejecutar los programas en cada terminal como se muestra en la figura 35.

Figura 35. Ejecución de los programas de forma simultánea.

Como se puede observar en los resultados anteriores, cada proceso accede a los elementos de la matriz que se está almacenando en la memoria compartida por medio de los semáforos de tal forma que se mantiene cierta sincronicidad a la hora de acceder a los datos de cada matriz por cada proceso. Dicha sincronicidad esta complementada con el uso de sleeps ya que esto permite poder esperar a la ejecución de los demás procesos antes de que alguno de los procesos termine con su ejecución de forma rápida.

Conclusiones

Sánchez De Los Ríos Flavio Josué

El lenguaje de programación C es muy útil para el cómputo paralelo debido a que permite la comunicación directa con la computadora y ofrece herramientas útiles para este fin. El conocimiento de elementos fundamentales del lenguaje C y de los sistemas operativos es crucial para el desarrollo de programas paralelos. Los tipos de datos en C, como enteros, de punto flotante, estructurados y enumerados, son importantes para el almacenamiento y representación de datos en cómputo paralelo. Los apuntadores son una herramienta fundamental en C, ya que permiten la manipulación de la memoria de manera directa y la creación de estructuras dinámicas. Las estructuras de datos en C se utilizan para almacenar y organizar datos relacionados de forma más conveniente y eficiente, y son esenciales para el desarrollo de aplicaciones complejas. Los semáforos son una herramienta importante en la programación concurrente para controlar el acceso a recursos compartidos y evitar condiciones de carrera entre procesos o hilos. En el lenguaje de programación C, los semáforos se implementan a través de la biblioteca de sincronización de hilos (pthread.h) o la biblioteca de semáforos del sistema (semaphore.h). Es importante comprender cómo funcionan los semáforos, cómo se declaran y utilizan en C, y cómo se pueden aplicar para crear programas eficientes y seguros. Al utilizar semáforos en C, se pueden gestionar recursos compartidos de manera efectiva, evitando problemas como el acceso no autorizado a recursos críticos y el bloqueo de procesos o hilos. En resumen, los semáforos son una herramienta esencial para la programación concurrente en C, y su correcto uso puede contribuir significativamente a la eficiencia y seguridad de los programas.

Tinoco Videgaray Sergio Ernesto

El uso de semáforos a la hora de acceder a ciertas regiones de memoria en el caso de la memoria compartida resulta un elemento de gran utilidad ya que estamos manteniendo un flujo con cierta sincronicidad a la hora de acceder o incluso de modificar estos datos, cabe destacar que el uso de sleeps junto con los semáforos ayuda a balancear esta sincronicidad ya que a la hora de ejecutar los programas siempre habrá uno que inicia su procedimiento antes que los demás, consumiendo más recursos del procesador y de esta forma, realizando su operación de forma más rápida, lo que puede causar un bloqueo de memoria en dicha región, por lo que nuevamente los sleeps resultan un elemento clave en el desarrollo de esta práctica. Por último, me gustaría mencionar que el uso de un lenguaje de nivel medio como el caso de C resulta de gran utilidad a la hora de trabajar con semáforos y de igual forma con la memoria compartida ya que el uso de los apuntadores resulta indispensable para la gestión de la memoria principal del sistema y junto con ayuda de Linux para la gestión de los procesos ejecutándose en tiempo real.

Ibarra Gonzaléz Emilio Francisco

Los semáforos son un mecanismo importante para la sincronización. En el cómputo paralelo, múltiples procesos o hilos pueden ejecutarse simultáneamente en diferentes núcleos o CPU, y pueden acceder a los mismos recursos compartidos, como variables o archivos. El uso de semáforos en C se realiza mediante la biblioteca POSIX, que proporciona funciones para crear y administrar semáforos.

Los semáforos son de importancia en la paralelizarían de programas, al garantizar la sincronización y la comunicación entre procesos o hilos y evitar errores como bloqueos. Los semáforos permiten que los procesos o hilos se coordinen y se comuniquen entre sí para evitar conflictos y garantizar que los recursos compartidos se utilicen de manera segura y eficiente. Funcionan como una especie de bloqueo que permite a un proceso o hilo esperar hasta que se libere un recurso antes de acceder a él, y evita que varios procesos o hilos accedan al mismo recurso al mismo tiempo.

Referencias

[1] K. Hwang. “Advanced Computer architecture: Paralelism, Scalability, Programability”. Mc Graw-Hill, 1981.

[2] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," Prentice Hall, 1988.

[3] The Open Group. "unistd.h - standard symbolic constants and types". IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008).

[4] S. Oualline. “Practical C programming”. O'Reilly Media, Inc.1997.

[5]P. Bovet, M. Cesati, “Understanding The Linux Kernel Understanding The Linux Kernel” 3rd Edition 3rd Edition O'Reilly Media, Inc., 2005.

[6] M. Aldea.,M. Gonzalez., “Programación Concurrente”. Universidad de Cantabria. [Online] Disponible en: https://www.ctr.unican.es/asignaturas/mc_procon/Doc/ProCon_II_06-sincronizacion_3en1.pdf

[7] J. Duran. (2015). “Programación paralela en C : Semáforos” Disponible en: https://www.somosbinarios.es/programacion-paralela-en-c-semaforos/

[8] Ubuntu Manpage Repository. (2019). “ftok” Disponible en: https://manpages.ubuntu.com/manpages/trusty/es/man3/ftok.3.html

[9] Ubuntu Manpage Repository. (2019). “shmget” Disponible en: https://manpages.ubuntu.com/manpages/bionic/es/man2/shmget.2.html

[10] Kerrisk M. (2022). “shmat” Disponible en: https://man7.org/linux/man-pages/man3/shmat.3p.html

[11] Kerrisk M. (2022). “shmdt” Disponible en: https://linux.die.net/man/2/shmdt

[12] Ubuntu Manpage Repository. (2019). “shmctl” Disponible en: https://manpages.ubuntu.com/manpages/bionic/es/man2/shmctl.2.html

[13] Ubuntu Manpage Repository. (2019). “semget” Disponible en: https://manpages.ubuntu.com/manpages/bionic/es/man2/semget.2.html

[14] Ubuntu Manpage Repository. (2019). “semctl” Disponible en: https://manpages.ubuntu.com/manpages/bionic/es/man2/semctl.2.html

[15] Ubuntu Manpage Repository. (2019). “semop” Disponible en: https://manpages.ubuntu.com/manpages/bionic/es/man2/semop.2.html

Anexo. Códigos completos.

Código del programa Main.c

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

int main()
{
   int shmid0, shmid1, i,j;
   int *resultados, *datos_matriz;
   int **app_mat;
   int matriz[3][9]={ {1, 2, 3, 4, 5, 6, 7, 8, 9},
                      {1, 3, 5, 7, 9, 11, 13, 15, 17},
                      {2, 4, 6, 8, 10, 12, 14, 16, 18 }};
   key_t llave0, llave1;
   pid_t pid;

   llave0=ftok("Archivo1",'k');
   llave1=ftok("Archivo2",'k');

   shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|0777);
//Region de la matriz
   shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|0777);  //Region de los
resultados

   datos_matriz=(int*)shmat(shmid0,0,0);
   resultados=(int *)shmat(shmid1,0,0);


   //Reservar memoria para la matriz
   app_mat=(int**)malloc(3*sizeof(app_mat[0]));

   for(i=0;i<3;i++)
     {
        app_mat[i]=datos_matriz+i*9;
     }


  //Guardar matriz en la memoria compartida
```

```
   for(i=0;i<3;i++)
     {
        for(j=0;j<9;j++)
       {
          app_mat[i][j]=matriz[i][j];
       }
     }



   printf("\n\nEl escritor de matriz ha terminado\n\n");

   while(resultados[0]==1 || resultados[1]==2 || resultados[2]==3);
//Espera hasta que todos los resultados se registren

   if(resultados[0]!=1 && resultados[1]!=2 && resultados[2]!=3)
//Evalua si ya se registraron todos los resultados de las sumas
   {
    for(i=0;i<3;i++)
     {
        printf("Suma Fila %d: %d\n",i+1,resultados[i]); //Imprime los
resultados de cada fila
     }
   }

   sleep(5);
   shmdt(&app_mat);
   shmdt(&resultados);
   shmctl(shmid0,IPC_RMID,0);
   shmctl(shmid1,IPC_RMID,0);
   return 0;
}
```

Código del archivo de cabecera gestionarSemaforos.h

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

#define PERMISOS 0644

int Crea_semaforo(key_t llave,int valor_inicial)
{
   int semid=semget(llave,1,IPC_CREAT|PERMISOS);
   if(semid==-1)
   {
      return -1;
   }
   semctl(semid,0,SETVAL,valor_inicial);
   return semid;
}

void down(int semid)
{
   struct sembuf op_p[]={0,-1,0};
   semop(semid,op_p,1);
}

void up(int semid)
{
   struct sembuf op_v[]={0,+1,0};
   semop(semid,op_v,1);
}
```

Código del programa ProcesoA.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "gestionarSemaforos.h"

#define PERMISOS 0644



int main()
{
   int *resultados=NULL, *datos_matriz=NULL;
   int semaforo_matriz,semaforo_resultados,suma=0,j=0;

   int shmid0, shmid1, i;
   int **app_mat;
   key_t llave0, llave1,llave_sem1,llave_sem2;
   pid_t pid;

   //Generar las llaves para la memoria compartida
   llave0=ftok("Archivo1",'m');
   llave1=ftok("Archivo2",'m');


shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|PERMISOS);
//Region de la matriz
   shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|PERMISOS);  //Region de
los resultados

   datos_matriz=(int*)shmat(shmid0,0,0);
   resultados=(int *)shmat(shmid1,0,0);

   //Reservar memoria
   app_mat=(int**)malloc(3*sizeof(app_mat[0]));

   for(i=0;i<3;i++)
     {
```

```
        app_mat[i]=datos_matriz+i*9;
     }

  //Generar las llaves para los semaforos
   llave_sem1=ftok("S1",'k');
   llave_sem2=ftok("S2",'k');

   //Crear semaforos
   semaforo_matriz=Crea_semaforo(llave_sem1,1);
   semaforo_resultados=Crea_semaforo(llave_sem2,1);

   //Sumar fila 1 usando semaforos
   for(int j=0;j<9;j++)
   {
      down(semaforo_matriz);
      sleep(1);
      suma+=app_mat[0][j];
      up(semaforo_matriz);
   }

   printf("\n\nEl proceso %d, termino de sumar la fila
1\n\n",getpid());
   printf("Suma de la primer fila: %d\n",suma);

   //Registrar resultado usando semaforos
      down(semaforo_resultados);
      resultados[0]=suma;
      up(semaforo_resultados);

   //Disociar apuntadores de los segmentos de memoria
   shmdt(&app_mat);
   shmdt(&resultados);
   shmctl(shmid0,IPC_RMID,0);
   shmctl(shmid1,IPC_RMID,0);
   return 0;
}
```

Código del programa ProcesoB.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "gestionarSemaforos.h"

#define PERMISOS 0644


int main()
{
   int *resultados=NULL, *datos_matriz=NULL;
   int semaforo_matriz,semaforo_resultados,suma=0,j=0;

   int shmid0, shmid1, i;
   int **app_mat;
   key_t llave0, llave1,llave_sem1,llave_sem2;
   pid_t pid;

   //Generar las llaves para la memoria compartida
   llave0=ftok("Archivo1",'m');
   llave1=ftok("Archivo2",'m');


shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|PERMISOS);
//Region de la matriz
   shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|PERMISOS);  //Region de
los resultados

   datos_matriz=(int*)shmat(shmid0,0,0);
   resultados=(int *)shmat(shmid1,0,0);

   //Reservar memoria
   app_mat=(int**)malloc(3*sizeof(app_mat[0]));

   for(i=0;i<3;i++)
     {
        app_mat[i]=datos_matriz+i*9;
```

```
    }

  //Generar las llaves para los semaforos
   llave_sem1=ftok("S1",'k');
   llave_sem2=ftok("S2",'k');

   //Crear semaforos
   semaforo_matriz=Crea_semaforo(llave_sem1,1);
   semaforo_resultados=Crea_semaforo(llave_sem2,1);

   //Sumar fila 2 usando semaforos
   for(int j=0;j<9;j++)
   {
      down(semaforo_matriz);
      sleep(1);
      suma+=app_mat[1][j];
      up(semaforo_matriz);
   }

   printf("\n\nEl proceso %d, termino de sumar la fila
2\n\n",getpid());
   printf("Suma de la segunda fila: %d\n",suma);

   //Registrar resultado usando semaforos
      down(semaforo_resultados);
      resultados[1]=suma;
      up(semaforo_resultados);

   //Disociar apuntadores de los segmentos de memoria
   shmdt(&app_mat);
   shmdt(&resultados);
   shmctl(shmid0,IPC_RMID,0);
   shmctl(shmid1,IPC_RMID,0);
   return 0;
}
```

Código del programa ProcesoC.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "gestionarSemaforos.h"

#define PERMISOS 0644


int main()
{
   int *resultados=NULL, *datos_matriz=NULL;
   int semaforo_matriz,semaforo_resultados,suma=0,j=0;

   int shmid0, shmid1, i;
   int **app_mat;
   key_t llave0, llave1,llave_sem1,llave_sem2;
   pid_t pid;

   //Generar las llaves para la memoria compartida
   llave0=ftok("Archivo1",'m');
   llave1=ftok("Archivo2",'m');


shmid0=shmget(llave0,3*9*sizeof(datos_matriz[0]),IPC_CREAT|PERMISOS);
//Region de la matriz
   shmid1=shmget(llave1,3*sizeof(int),IPC_CREAT|PERMISOS);  //Region de
los resultados

   datos_matriz=(int*)shmat(shmid0,0,0);
   resultados=(int *)shmat(shmid1,0,0);

   //Reservar memoria
   app_mat=(int**)malloc(3*sizeof(app_mat[0]));

   for(i=0;i<3;i++)
     {
        app_mat[i]=datos_matriz+i*9;
```

```
    }

  //Generar las llaves para los semaforos
   llave_sem1=ftok("S1",'k');
   llave_sem2=ftok("S2",'k');

   //Crear semaforos
   semaforo_matriz=Crea_semaforo(llave_sem1,1);
   semaforo_resultados=Crea_semaforo(llave_sem2,1);

   //Sumar fila 3 usando semaforos
   for(int j=0;j<9;j++)
   {
      down(semaforo_matriz);
      sleep(1);
      suma+=app_mat[2][j];
      up(semaforo_matriz);
   }

   printf("\n\nEl proceso %d, termino de sumar la fila
3\n\n",getpid());
   printf("Suma de la tercer fila: %d\n",suma);

   //Registrar resultado usando semaforos
      down(semaforo_resultados);
      resultados[2]=suma;
      up(semaforo_resultados);

   //Disociar apuntadores de los segmentos de memoria
   shmdt(&app_mat);
   shmdt(&resultados);
   shmctl(shmid0,IPC_RMID,0);
   shmctl(shmid1,IPC_RMID,0);
   return 0;
}
```