



**Instituto Politécnico Nacional  
Escuela Superior de Cómputo  
“ESCOM”**



**Unidad de Aprendizaje:  
Cómputo Paralelo**

**Práctica No. 5  
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: 14/05/23**

## Resumen

En los sistemas de cómputo paralelo se utilizan varios recursos y herramientas que facilitan el desarrollo de tareas de forma simultánea, algunos de estos recursos son la memoria compartida que nos permite acceder a un mismo sector dentro de la memoria principal con el fin de leer o escribir un dato que pueda ser accedido por uno o más procesos dentro de la misma región de memoria, también están los semáforos que nos permite mantener la sincronización entre los distintos procesos que se ejecutan de tal forma que no se presente algún inconveniente a la hora de acceder al mismo elemento de la memoria compartida. Y uno de los recursos más utilizados que se van a abordar en el desarrollo de esta práctica son los hilos o también llamados “threads”, los cuales se definen como una secuencia de instrucciones o conjunto de instrucciones que puede ser ejecutada de forma independiente dentro de un proceso. Muchos sistemas operativos hacen uso de estos hilos, en algunos casos llamados como “subprocesos” cumplen la función de realizar múltiples operaciones “simples” para de esta forma dividir la carga del trabajo de cada proceso.

Existen muchas formas de implementar estos hilos en los diferentes lenguajes de programación, algunas de ellas son muy similares ya que implementan paquetes o bibliotecas que permiten el uso de estos hilos dentro del sistema. En el caso del lenguaje C no resulta muy difícil puesto que C es un lenguaje de programación de nivel medio, lo que nos permite trabajar de forma más “directa” con los recursos del sistema operativo.

En el desarrollo de esta práctica se va a codificar un programa (programa 1) que haciendo uso de hilos y memoria compartida, va a realizar distintos cálculos y los va a almacenar en un arreglo de resultados dentro de la memoria compartida. Para que posteriormente un segundo proceso (programa 2) utilice nuevamente hilos y semáforos para acceder a los resultados almacenados en dicho arreglo y mostrarlos en la terminal correspondiente.

# Índice

## CONTENIDO

Introducción .....	1
Marco teórico.....	2
Desarrollo.....	9
Conclusiones .....	21
Referencias .....	24
Anexo. Códigos completos. ....	26

## Introducción

Uno de los recursos más utilizados en el cómputo paralelo son los hilos o también llamados “threads”, los cuales se definen como una secuencia de instrucciones o conjunto de instrucciones que puede ser ejecutada de forma independiente dentro de un proceso. Muchos sistemas operativos hacen uso de estos hilos, en algunos casos llamados como “subprocesos” cumplen la función de realizar múltiples operaciones “simples” para de esta forma dividir la carga del trabajo de cada proceso.

Estos hilos son elementos fundamentales en el ámbito de la programación concurrente y paralela. Cada hilo puede entenderse como una secuencia de instrucciones o un flujo de ejecución que es independiente de un proceso. A medida que los sistemas informáticos han evolucionado y se han vuelto cada vez más complejos, la capacidad de ejecutar múltiples tareas simultáneamente se ha vuelto cada vez más importante.

Otros elementos fundamentales son los semáforos, los cuales 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.

El objetivo de esta práctica se centra en aplicar y comprender el uso y funcionamiento de varios recursos de cómputo paralelo como lo son: la memoria compartida para almacenar datos que puedan ser accedidos por más de un solo proceso, los semáforos para la sincronización de distintos procesos que accedan a la misma región de memoria compartida y los hilos que nos van a permitir dividir las operaciones a realizar por cada proceso.

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.

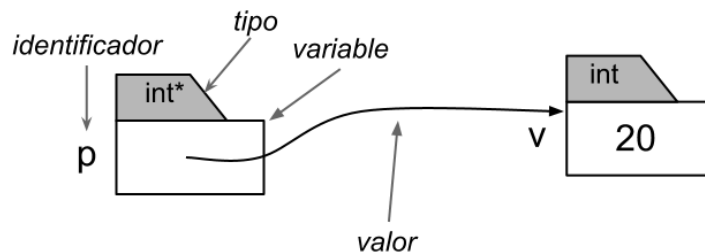


Figura 1. Diagrama de la función de un apuntador.

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 subtarefas 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].

<b>Procesos en primer plano y en segundo plano</b>	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.
<b>Procesos daemon</b>	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.
<b>Procesos zombie</b>	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 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`.



## Hilos

Los hilos de proceso, también conocidos como hilos o “threads”, son elementos fundamentales en el ámbito de la programación concurrente y paralela. Un hilo puede entenderse como una secuencia de instrucciones o un flujo de ejecución independiente dentro de un proceso.

Un proceso puede contener uno o varios hilos, y cada uno de ellos representa una unidad de trabajo que puede realizar tareas en paralelo con otros hilos del mismo proceso. Aunque los hilos comparten los recursos del proceso, como la memoria y los archivos abiertos, cada hilo tiene su propio contador de programa, pila de ejecución y estado de registro. Esto les proporciona una ejecución independiente y les permite comunicarse y cooperar entre sí de manera eficiente.

Una de las ventajas de utilizar hilos de proceso es la capacidad de realizar múltiples tareas simultáneamente dentro de una aplicación. Por ejemplo, en una aplicación de procesamiento de imágenes, se puede utilizar un hilo para cargar imágenes desde el disco, otro hilo para procesar las imágenes y un tercer hilo para mostrar los resultados en la interfaz de usuario. Esto permite que la aplicación sea más receptiva y evita que una tarea prolongada bloquee por completo la ejecución de otras tareas.

Además, los hilos son especialmente útiles en sistemas con múltiples núcleos de procesamiento o también denominados “multicore”. Cada núcleo puede ejecutar un hilo diferente, lo que permite una ejecución paralela real y un mejor aprovechamiento de los recursos del hardware. Esto puede resultar en un aumento significativo en el rendimiento de las aplicaciones que hacen un uso intensivo de la CPU.

Sin embargo, trabajar con hilos también presenta algunos desafíos. Los hilos comparten memoria y otros recursos, lo que puede llevar a condiciones de carrera y problemas de sincronización. Una condición de carrera ocurre cuando varios hilos intentan acceder y modificar los mismos datos simultáneamente, lo que puede llevar a resultados inesperados o incorrectos. Por lo tanto, es necesario utilizar mecanismos de sincronización, como cerrojos o semáforos, para garantizar que los hilos accedan a los recursos compartidos de manera segura y ordenada [8].

Los principales estados de los hilos son: Ejecución, Listo y Bloqueado. Si un proceso es expulsado de la memoria principal, todos sus hilos deberán estarlo ya que todos comparten el espacio de direcciones del proceso [9].

- *Creación:* Cuando se crea un proceso se crea un hilo para ese proceso. Luego, este hilo puede crear otros hilos dentro del mismo proceso, proporcionando un puntero de instrucción y los argumentos del nuevo hilo. El hilo tendrá su propio contexto y su propio espacio de la columna, y pasará al final de los Listos.
- *Bloqueo:* Cuando un hilo necesita esperar por un suceso, se bloquea (salvando sus registros de usuario, contador de programa y punteros de pila). Ahora el procesador podrá pasar a ejecutar otro hilo que esté al principio de los Listos mientras el anterior permanece bloqueado.
- *Desbloqueo:* Cuando el suceso por el que el hilo se bloquea se produce, el mismo pasa a la final de los Listos.
- *Terminación:* Cuando un hilo finaliza se liberan tanto su contexto como sus columnas.

## Suma de Gauss

La suma de Gauss es un concepto matemático importante que se atribuye al famoso matemático alemán Carl Friedrich Gauss, tiene diversas aplicaciones en matemáticas, física y ciencias de la computación. Por ejemplo, puede utilizarse para calcular la suma total de elementos en una serie o para determinar el número de operaciones realizadas en un algoritmo iterativo. Además, la fórmula de la suma de Gauss puede ser extendida para sumar progresiones aritméticas y series más complejas [10].

Para calcular la suma de Gauss, Gauss propuso un método ingenioso. En lugar de sumar los números en orden, decidió sumar el primer y el último número, luego el segundo y el penúltimo número, y así sucesivamente. La fórmula para la suma de Gauss se expresa de la siguiente manera:  $S = 1 + 2 + 3 + \dots + n$ . Al final, todas las parejas suman  $n + 1$ :

$$S = (n + 1) + (n + 1) + (n + 1) + \dots + (n + 1) \text{ [se suman "n" términos]}$$

$$2S = n(n + 1)$$

Por lo tanto, la suma de Gauss se puede expresar como el producto de  $n$  y  $n + 1$ , dividido por 2:  $S = n(n + 1)/2$ , dando como resultado la ecuación de la figura 2 [10].

$$\sum_{i=1}^n i = \frac{n(n + 1)}{2}$$

Figura 2. Fórmula de la suma de Gauss.

## Factorial

Los números factoriales se utilizan sobre todo en combinatoria, para calcular combinaciones y permutaciones. A través de la combinatoria, los factoriales también se suelen utilizar para calcular probabilidades [11].

La función factorial se representa con un signo de exclamación “!” detrás de un número. Esta exclamación quiere decir que hay que multiplicar todos los números enteros positivos que hay entre ese número y el 1 siguiendo la ecuación de la figura 3.

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$$

Figura 3. Ecuación del factorial de  $n$ .

## Sucesión de Fibonacci

Descubierta por el matemático italiano Leonardo de Pisa, más conocido como Fibonacci, en el siglo XIII. Su aprendizaje se produjo gracias a los viajes que hacía junto a su padre, que era comerciante. El curioso origen de la sucesión está en la observación que hizo el mencionado matemático de cómo se propagan las parejas de conejos a partir de una pareja de cachorros. Posteriormente, se ha comprobado que numerosos fenómenos de la naturaleza están relacionados con esta sucesión. Aparece en la estructura espiral del caparazón de algunos moluscos y en la disposición de las hojas de algunas plantas. Asimismo, se aplica también a cuestiones relacionadas con computación y teoría de juegos [12].

Es una secuencia infinita de números naturales cuyos dos primeros términos son 1 y 1 y tal que, cualquier otro término se obtiene sumando los dos inmediatamente anteriores. Por tanto, se cumple la igualdad de la ecuación mostrada en la figura 4:

$$f_n = f_{n-1} + f_{n-2}$$

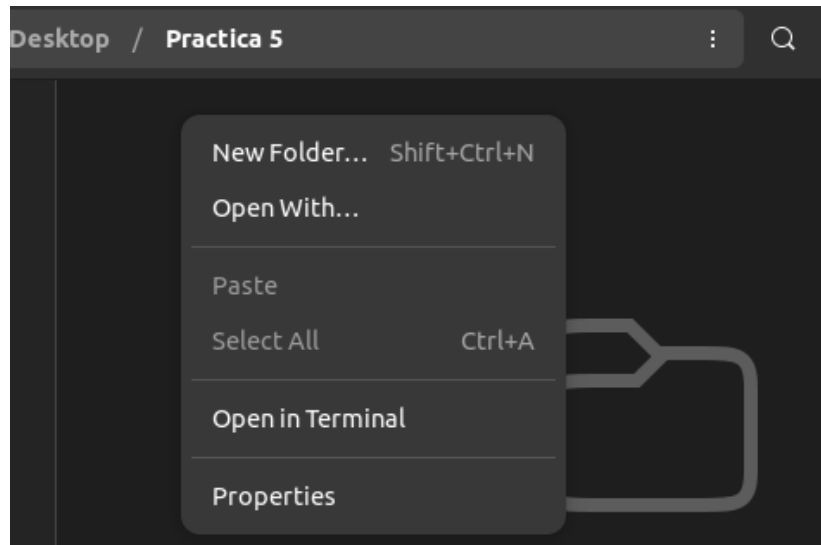
Figura 4. Ecuación de la sucesión de Fibonacci.

De manera explícita, tendríamos que es: 1, 1, 2, 3, 5, 8, 13, 21, 34... hasta n.

## Desarrollo

Primeramente, se van a generar los archivos para cada uno de los procesos.

Para ello se va a abrir una ventana de la terminal dentro de la carpeta de la práctica (figura 5).



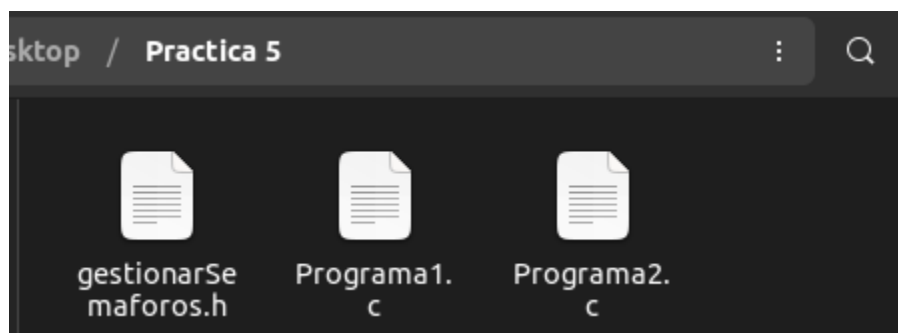
*Figura 5. Vista de la carpeta Practica 5.*

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 6).

```
~/Desktop/Practica 5$ touch Programa1.c  
~/Desktop/Practica 5$ touch Programa2.c  
~/Desktop/Practica 5$ touch gestionarSemaforos.h  
~/Desktop/Practica 5$
```

*Figura 6. 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 7).



*Figura 7. 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.

- Archivo de cabecera.

Para los programas 1 y 2 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, así como un tipo de dato “struct” para definir los parámetros de cada hilo (figura 8).

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

*Figura 8. 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 9).

```
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/sem.h>  
  
#define PERMISOS 0644
```

*Figura 9. Código de las bibliotecas a utilizar dentro del archivo gestionarSemaforos.h*

Se define un tipo de dato abstracto por medio de una estructura con la cual se va a definir el semáforo y la región de memoria compartida que va a utilizar cada hilo de proceso (figura 10).

```
typedef struct  
{  
    int *resultados;  
    int semaforo;  
}Parametros;
```

*Figura 10. Código de la estructura para definir el tipo de dato “Parámetros”.*

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 [18]. Posteriormente se va a inicializar dicho semáforo con el valor inicial dado a través de la función semctl [17]. Finalmente se retorna la ID del semáforo creado (figura 11).

```

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 11. 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 semáforo [20] (figura 12).

```

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

```

*Figura 12. 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 haciendo uso de la función “semop” [20] (figura 13).

```

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

```

*Figura 13. Código de la función que establece un estado alto en el semáforo.*

- Programa 1.

En el programa 1, se van a definir las funciones de cada hilo así como las funciones que cada hilo va a ejecutar para realizar los cálculos correspondientes.

Se define la función “sumaGauss” la cual va a recibir un número “n” y va a calcular la sumatoria desde 1 hasta “n” utilizando la fórmula de la suma Gauss (figura 2 y 14).

```
int sumaGauss(int n)
{
    return (n*(n+1))/2;
}
```

*Figura 14. Código de la función que realiza la suma de Gauss.*

De igual forma se define la función que calcula el factorial de un número dado utilizando la recursividad, es decir, la función se invoca a sí misma para realizar el cálculo completo (figura 3 y 15).

```
int factorial(int n)
{
    if(n<=1)
    {
        return 1;
    }

    return n*factorial(n-1);
}
```

*Figura 15. Código de la función recursiva que calcula el factorial de un número dado.*

Por último, se define la función que calcula la sucesión de Fibonacci hasta un número dado, esto de forma recursiva como en la función del factorial (figura 4 y 16).

```

int fibonacci(int n)
{
    if(n<=1)
    {
        return n;
    }

    return fibonacci(n-1) + fibonacci(n-2);
}

```

*Figura 16. Código de la función recursiva que calcula la sucesión de Fibonacci de un número dado.*

Posteriormente se van a programar las funciones de cada hilo para el proceso 1.

Se define la función del hilo 1 que va a recibir como único parámetro un tipo de dato “parámetros” que se definió previamente en la estructura del archivo de cabecera (figura 10).

Dentro de la función del hilo 1 se va invocar a la función sumaGauss pasando como parámetro el número 100 para realizar la sumatoria del 1 al 100. Para ello se invocan las funciones down y up del archivo de cabecera para hacer uso del semáforo que se va a generar y de esta forma llevar un control más sincronizado con los otros hilos.

Finalmente se termina la ejecución del hilo por medio de la función “pthread\_exit”.

Nótese que para el uso de los semáforos y el arreglo de resultados se realizó un “casting” a los elementos del struct “parámetros” indicado por el operador flecha (figura 17).

```

void Hilo1(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int semaforo=(int)parametros->semaforo;

    printf("\nDentro del hilo 1-1 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());
    down(semaforo);
    resultados[0]=sumaGauss(100);
    up(semaforo);

    pthread_exit(0);
}

```

*Figura 17. Código de la función Hilo 1-1 donde se va a invocar la función de la suma de Gauss utilizando semáforos.*



De igual forma, se define la función del hilo 2 la cual va a realizar la operación del factorial de 10 haciendo uso de las funciones down y up de los semáforos. Y al igual que el hilo 1 se va a realizar el casting correspondiente a los elementos del struct “parámetros” que la función recibe como argumento (figura 18).

```
void Hilo2(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int semaforo=(int)parametros->semaforo;

    printf("\nDentro del hilo 1-2 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());
    down(semaforo);
    resultados[1]=factorial(10);
    up(semaforo);

    pthread_exit(0);
}
```

*Figura 18. Código de la función Hilo 1-2 donde se va a invocar la función factorial utilizando semáforos.*

Por último, se define la función del hilo 3 la cual va a realizar el cálculo de los 20 primeros números de la sucesión de Fibonacci. Para ello se va a definir un bucle for que va a iterar desde 1 hasta 20 para registrar cada valor en el espacio de memoria correspondiente.

De igual forma para el uso de los semáforos y el arreglo de resultados se hace un casting a los elementos del struct “parámetros” (figura 19).

```
void Hilo3(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int i,semaforo=(int)parametros->semaforo;

    sleep(1);
    down(semaforo);
    printf("\nDentro del hilo 1-3 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());
    for(i=1;i<=20;i++)
    {
        resultados[i+1]=fibonacci(i);
    }
    up(semaforo);
    pthread_exit(0);
}
```

*Figura 19. Código de la función Hilo 1-3 donde se va a invocar la función fibonacci para los primeros 20 números haciendo uso de semáforos.*

Posteriormente se define la función principal del programa 1.

Primeramente, se definen las variables correspondientes para el uso de los hilos, en este caso se va a definir un arreglo de 3 elementos ya que se van a utilizar 3 hilos únicamente.

De igual forma se definen las variables necesarias para el uso de la memoria compartida y los semáforos.

Por último se define una variable de tipo “Parámetros” que corresponde con el tipo de dato abstracto o “TDA” que se definió previamente por medio de un struct, dicha variable va a

almacenar el identificador del semáforo correspondiente a la región de memoria compartida así como el apuntador a dicha región de memoria compartida, esto debido a que la función de cada hilo solo puede recibir un parámetro a la vez, y ya que se deben utilizar al menos estos dos parámetros, se optó por utilizar una estructura que encapsule estos dos elementos y nos permita manejarlos de manera individual (figura 10 y 20).

```
pthread_attr_t atributos;  
pthread_t identificadores[3]; //Arreglo de 3 hilos  
  
int shmId1, semaforo;  
int *resultados;  
key_t llave1, llave_sem;  
Parametros parametros;
```

Figura 20. Código de la función main del programa 1 donde se declaran las variables a utilizar.

Por medio de la función `ftok` se genera la llave para generar un segmento de memoria compartida [13] y posteriormente haciendo uso de la función `shmget` y `shmat` [14, 15], se genera la región de memoria compartida comprendida por 22 enteros la cual va a almacenar los resultados de cada hilo y se asocian al elemento resultado del struct “parámetros” (figura 21).

```
llave1=ftok("Archivo1", 'k');  
shmId1=shmget(llave1, 22*sizeof(int), IPC_CREAT|0777);  
parametros.resultados=(int *)shmat(shmId1, 0, 0);
```

Figura 21. Código de la función main del programa 1 donde se define el espacio de memoria compartida para los resultados de cada hilo.

Se definen los atributos de cada hilo y se desvinculan del proceso padre (figura 22).

```
pthread_attr_init(&atributos);  
pthread_attr_setdetachstate(&atributos, PTHREAD_CREATE_DETACHED);  
printf("\nSoy el proceso padre 1 con ID: %d\n", getpid());
```

Figura 22. Código de la función main del programa 1 donde se inicializan los atributos de cada hilo.

Posteriormente se genera la ID del semáforo que se va a utilizar para la región de memoria compartida que se generó previamente utilizando la función “Crea\_semaforo” definida en el archivo de cabecera, dicha ID se va a almacenar en la variable de la estructura “parámetros” (figura 11 y 23).

```
//Crear semaforos
llave_sem=ftok("S1",'k');
parametros.semaforo=Crea_semaforo(llave_sem,1);
```

*Figura 23. Código de la función main del programa 1 donde se inicializa el semáforo a utilizar.*

Se ejecuta la función “pthread\_create” para generar y ejecutar los hilos correspondientes [21], para ello se pasa como parámetro el ID del hilo, la dirección de memoria de la variable atributos, el nombre de la función del hilo y la dirección de memoria de la variable parámetros (que corresponde con el struct donde se almacena el ID del semáforo y el apuntador a la memoria compartida) figura 24.

```
pthread_create(&identificadores[0],&atributos,(void*)Hilo1,(void*)&parametros);
pthread_create(&identificadores[1],&atributos,(void*)Hilo2,(void*)&parametros);
pthread_create(&identificadores[2],&atributos,(void*)Hilo3,(void*)&parametros);
```

*Figura 24. Código de la función main del programa 1 donde se ejecutan los hilos.*

Finalmente, por medio de la función “shmdt” se van a disociar los segmentos de memoria compartida de los apuntadores [16] y posteriormente se van a eliminar dichos segmentos de memoria compartida a través de la función “shmctl” [17] como se muestra en la figura 25.

```
shmdt(&parametros.resultados);
shmctl(shmid1,IPC_RMID,0);
return 0;
```

*Figura 25. Destrucción de los segmentos de memoria existentes.*

- Programa 2.

Al igual que el programa 1, en el programa 2, se van a definir las funciones de cada hilo y se van a crear las variables correspondientes para hacer uso de la memoria compartida y los semáforos.

Primeramente, se define la función del hilo 1 para el proceso 2 (hilo 2-1) que va a recibir como único parámetro un tipo de dato “parámetros” que se definió previamente en la estructura del archivo de cabecera (figura 10) y va a mostrar el resultado del hilo 1-1 que se almacenó en la memoria compartida.

Dentro de la función del hilo 1 se va a evaluar el resultado generado en el hilo 1 del proceso 1 (hilo 1-1), hasta que dicho valor se actualice para posteriormente mostrar el resultado en la terminal haciendo uso del semáforo correspondiente.

Finalmente se termina la ejecución del hilo por medio de la función “pthread\_exit”.

Nótese que al igual que el proceso 1, se realizó un “casting” a los elementos del struct “parámetros” indicado por el operador flecha con el fin de hacer uso de los semáforos y el arreglo de resultados (figura 26).

```
void Hilo1(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int semaforo=(int)parametros->semaforo;

    while(resultados[0]==0);

    down(semaforo);
    printf("\nDentro del hilo 2-1 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());
    printf("Resultado del hilo 1-1: %d\n",resultados[0]);
    up(semaforo);

    pthread_exit(0);
}
```

*Figura 26. Código de la función Hilo 2-1 donde se va a mostrar el resultado del hilo 1-1 haciendo uso de semáforos.*

De igual forma, se define la función del hilo 2 la cual va a mostrar el resultado del hilo 1-2 que se almacenó en la región de memoria compartida.

Para ello se va a hacer uso de un bucle while que evalúe constantemente el resultado del hilo 2, hasta que dicho valor cambie de valor.

De igual forma se va a realizar el casting correspondiente para el uso del semáforo y la región de memoria compartida (figura 27).

```
void Hilo2(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int semaforo=(int)parametros->semaforo;
    while(resultados[1]==0);

    down(semaforo);
    printf("\nDentro del hilo 2-2 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());
    printf("Resultado del hilo 1-2: %d\n",resultados[1]);
    up(semaforo);

    pthread_exit(0);
}
```

*Figura 27. Código de la función Hilo 2-2 donde se va a mostrar el resultado del hilo 1-1 haciendo uso de semáforos.*

Por último, se define la función del hilo 3 la cual va a mostrar el resultado del hilo 1-3.

Al igual que las funciones del hilo 2-1 y 2-2 se va a realizar el casting de los elementos del struct parámetros y se va a evaluar constantemente el valor del tercer elemento en el arreglo.

Posteriormente y haciendo uso de los semáforos, se va a imprimir los elementos del arreglo desde la posición 2 hasta la 21 que corresponden con los 20 primeros números de la sucesión de Fibonacci (figura 28).

```

void Hilo3(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int i,semaforo=(int)parametros->semaforo;
    while(resultados[2]==0);

    down(semaforo);
    printf("\nDentro del hilo 2-3 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());
    printf("Resultado del hilo 1-3:\n");
    for(i=2;i<22;i++)
    {
        printf("%d ",resultados[i]);
    }
    up(semaforo);
    printf("\n");

    pthread_exit(0);
}

```

*Figura 28. Código de la función Hilo 2-3 donde se va mostrar el resultado del hilo 1-3 haciendo uso de semáforos.*

Posteriormente se define la función principal del programa 2.

Al igual que el programa 1, se definen las variables correspondientes para el uso de los hilos, en este caso se va a definir un arreglo de 3 elementos ya que se van a utilizar 3 hilos únicamente.

De igual forma se definen las variables necesarias para el uso de la memoria compartida y los semáforos.

Por último, se define la variable de tipo “Parámetros” que corresponde con el tipo de dato abstracto o “TDA” que se definió previamente por medio de un struct (figura 10 y 29).

```

pthread_attr_t atributos;
pthread_t identificadores[3]; //Arreglo de 3 hilos

int shmid1,semaforo;
int *resultados;
key_t llave1, llave_sem;
Parametros parametros;

```

*Figura 29. Código de la función main del programa 2 donde se declaran las variables a utilizar.*

Se genera la región de memoria compartida comprendida por 22 enteros la cual va a almacenar los resultados de cada hilo y se asocian al elemento resultado del struct “parámetros” (figura 30).

```
llave1=ftok("Archivo1",'k');
shmId1=shmget(llave1,22*sizeof(int),IPC_CREAT|0777);
parametros.resultados=(int *)shmat(shmId1,0,0);
```

*Figura 30. Código de la función main del programa 2 donde se define el espacio de memoria compartida para los resultados de cada hilo.*

Posteriormente se genera la ID del semáforo que se va a utilizar para la región de memoria compartida que se generó previamente utilizando la función “Crea\_semaforo” definida en el archivo de cabecera, dicha ID se va a almacenar en la variable de la estructura “parámetros” (figura 11 y 31).

```
//Crear semaforos
llave_sem=ftok("S1",'k');
parametros.semaforo=Crea_semaforo(llave_sem,1);
```

*Figura 31. Código de la función main del programa 2 donde se inicializa el semáforo a utilizar.*

Se invoca la función “pthread\_create” para generar y ejecutar los hilos correspondientes, para ello se pasa como parámetro el ID del hilo, se omite el uso de los atributos (NULL), el nombre de la función del hilo y la dirección de memoria de la variable parámetros (que corresponde con el struct donde se almacena el ID del semáforo y el apuntador a la memoria compartida) [21] figura 32.

```
pthread_create(&identificadores[0],NULL,(void*)Hilo1,(void*)&parametros);
pthread_create(&identificadores[1],NULL,(void*)Hilo2,(void*)&parametros);
pthread_create(&identificadores[2],NULL,(void*)Hilo3,(void*)&parametros);
```

*Figura 32. Código de la función main del programa 1 donde se ejecutan los hilos.*

Se ejecuta la función “pthread\_join” para cada uno de los hilos con el fin de que el proceso principal espere a que los 3 hilos terminen de ejecutarse [22] (figura 33).

```
for(j=0;j<3;j++)
{
    pthread_join(identificadores[j],NULL);
}
```

*Figura 33. Código de la función main del programa 2 donde se ejecuta la función pthread\_join.*

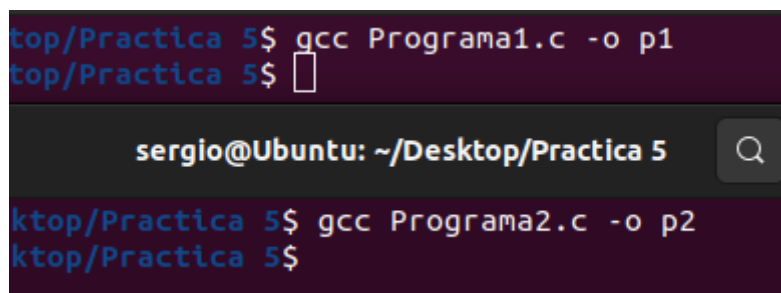
Finalmente, por medio de la función “shmdt” se van a disociar los segmentos de memoria compartida de los apuntadores [16] y posteriormente se van a eliminar dichos segmentos de memoria compartida a través de la función “shmctl” [17] como se muestra en la figura 34.

```
shmdt(&parametros.resultados);  
shmctl(shmid1,IPC_RMID,0);  
return 0;
```

*Figura 34. Destrucción de los segmentos de memoria existentes.*

- Resultados

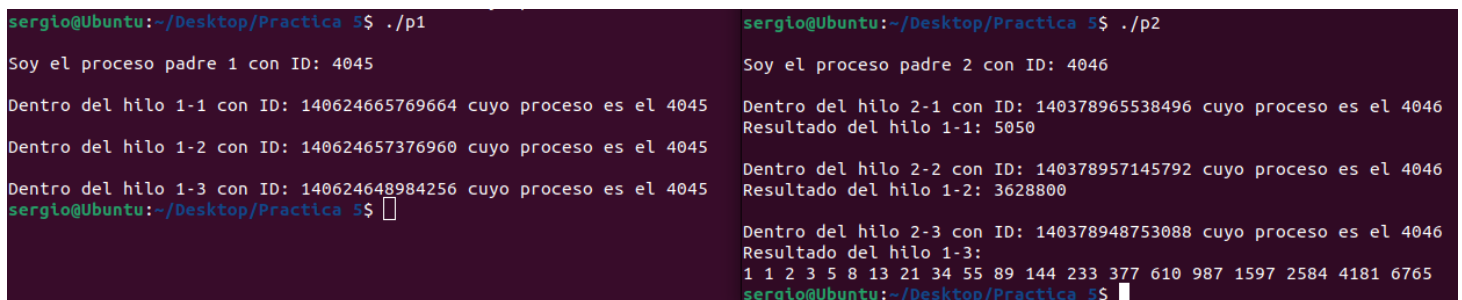
Para la ejecución de los programas se van a abrir 2 ventanas de la terminal (una para programa) dentro de la carpeta de la práctica 5. Una vez creadas las ventanas de la terminal se va a compilar cada archivo por separado como se muestra en la figura 35.



```
top/Practica 5$ gcc Programa1.c -o p1  
top/Practica 5$  
  
sergio@Ubuntu: ~/Desktop/Practica 5  
  
ktop/Practica 5$ gcc Programa2.c -o p2  
ktop/Practica 5$
```

*Figura 35. Compilación de los programas en cada ventana de la terminal.*

Finalmente se ejecutan los programas en cada terminal como se muestra en la figura 36.



```
sergio@Ubuntu:~/Desktop/Practica 5$ ./p1  
Soy el proceso padre 1 con ID: 4045  
Dentro del hilo 1-1 con ID: 140624665769664 cuyo proceso es el 4045  
Dentro del hilo 1-2 con ID: 140624657376960 cuyo proceso es el 4045  
Dentro del hilo 1-3 con ID: 140624648984256 cuyo proceso es el 4045  
sergio@Ubuntu:~/Desktop/Practica 5$  
  
sergio@Ubuntu:~/Desktop/Practica 5$ ./p2  
Soy el proceso padre 2 con ID: 4046  
Dentro del hilo 2-1 con ID: 140378965538496 cuyo proceso es el 4046  
Resultado del hilo 1-1: 5050  
Dentro del hilo 2-2 con ID: 140378957145792 cuyo proceso es el 4046  
Resultado del hilo 1-2: 3628800  
Dentro del hilo 2-3 con ID: 140378948753088 cuyo proceso es el 4046  
Resultado del hilo 1-3:  
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765  
sergio@Ubuntu:~/Desktop/Practica 5$
```

*Figura 36. Ejecución de los programas de forma simultánea.*

Como se puede observar en los resultados, cada proceso tiene su propio identificador el cual es de cierta forma “heredado” por cada uno de los hilos y a su vez cada hilo tiene un identificador que se caracteriza por ser un número entero largo o “long int”. Cada hilo del proceso 1 es ejecutado de forma simultánea y una vez se que dicho hilo calcula el resultado de su operación lo almacena en la región de memoria compartida que le corresponde para que posteriormente el hilo correspondiente del proceso 2 acceda al espacio de memoria donde se almacenó el resultado de dicho hilo y lo muestra en la terminal junto con su información.

## **Conclusiones**

**Sánchez De Los Ríos Flavio Josué**

Se abordó el tema del cómputo paralelo y la importancia de las herramientas de memoria compartida, semáforos e hilos para calcular y acceder a la información en cada proceso. Creo que lo que más nos costó trabajo fue la sincronización de la memoria compartida pues nos planteó un gran desafío, como la incoherencia de datos y la exclusión mutua. Para ello, se han desarrollado diversos algoritmos y métodos, entre ellos los "semáforos".

Los hilos son una herramienta crucial en la programación concurrente que permiten dividir las tareas de un proceso. En el lenguaje de programación C, se implementan a través de bibliotecas como pthread.h, y permiten a los hilos crearse y ejecutarse así como permitir al proceso principal esperar a que la ejecución de dichos hilos termine, lo cual evita la competencia por recursos y las condiciones de carrera.



## **Tinoco Videgaray Sergio Ernesto**

Los hilos son un recurso fundamental en el cómputo paralelo que permiten dividir la carga de trabajo a un nivel más profundo en el que ya no se hablan de procesos “padres” o procesos “no emparentados” sino de únicamente un solo proceso realizando múltiples operaciones de forma simultánea o “cuasi paralela”.

Como se pudo observar en esta práctica, el uso de hilos resulta un factor clave en la paralelización de tareas ya que nos permite dividir el conjunto de operaciones aún más que si se utilizaran únicamente procesos no emparentados que necesiten comunicarse entre sí, no obstante el uso de hilos también tiene sus desventajas como el hecho de que los hilos dependan del proceso al que pertenecen y es que en el dado caso de que dicho proceso terminará su ejecución antes que los hilos dentro de él, resultaría en una ejecución incompleta de dichos hilos y es por ello que los hilos tienen cierta dependencia a su proceso “padre” por decirlo de alguna forma.

De igual forma considero que el uso de hilos requiere de mucha sincronización tanto con su proceso “padre” como con los demás hilos y esto hace que el uso de hilos se vuelva un tanto complicado para su programación, aun así, los distintos lenguajes y frameworks que existen actualmente nos permiten implementar estas herramientas de forma más simple y funcional.

Ibarra González Emilio Francisco

La importancia de los procesos y los hilos radica en su capacidad para aprovechar al máximo los recursos del sistema, como la memoria y la capacidad de procesamiento, y permitir que los programas realicen múltiples tareas simultáneamente. Sin embargo, también es importante tener en cuenta que el uso inadecuado de procesos e hilos puede conducir a problemas de rendimiento y seguridad, como cuellos de botella, errores de sincronización y problemas de acceso a datos compartidos. Por lo tanto, se deben implementar semáforos, para permitir que los procesos o hilos puedan coordinarse evitando conflictos en el acceso a los recursos, lo que puede prevenir errores y corrupción de datos.

En resumen, los hilos de proceso son unidades de ejecución dentro de un proceso que permiten la ejecución simultánea de múltiples tareas. Su uso adecuado puede mejorar el rendimiento y la capacidad de respuesta de las aplicaciones, pero también requiere un manejo cuidadoso para evitar problemas de concurrencia. A medida que los sistemas informáticos continúan evolucionando, comprender y aprovechar los hilos de proceso se vuelve cada vez más importante para desarrollar aplicaciones eficientes y escalables.

## 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](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] M . Kerrisk. "The Linux Programming Interface". San Francisco. 2022. No Starch Press. (617-631).
- [9] Baeldung. (2022). "Linux Process vs. Thread". Disponible en:  
<https://www.baeldung.com/linux/process-vs-thread#:~:text=A%20thread%20is%20a%20lightweight%20process%20also%20called%20an%20LWP,slower%20due%20to%20isolated%20memory.>
- [10] E. Soto. (2020) "La suma de Gauss". Disponible en:  
<https://www.aprendematematicas.org.mx/la-suma-de-gauss/>
- [11] I. Del Amo. (2022) "Factoriales. ¿Para qué los utilizamos?". Disponible en:  
<https://www.smartick.es/blog/matematicas/numeros-enteros/factoriales/>

[12] Universidad de Almería. (2018) "Sucesión de Fibonacci". Disponible en:

<https://www2.ual.es/jardinmatema/sucesion-de-fibonacci/>

[13] Ubuntu Manpage Repository. (2019). "ftok" Disponible en:

<https://manpages.ubuntu.com/manpages/trusty/es/man3/ftok.3.html>

[14] Ubuntu Manpage Repository. (2019). "shmget" Disponible en:

<https://manpages.ubuntu.com/manpages/bionic/es/man2/shmget.2.html>

[15] M . Kerrisk (2022). "shmat" Disponible en:

<https://man7.org/linux/man-pages/man3/shmat.3p.html>

[16] M . Kerrisk (2022). "shmdt" Disponible en:

<https://linux.die.net/man/2/shmdt>

[17] Ubuntu Manpage Repository. (2019). "shmctl" Disponible en:

<https://manpages.ubuntu.com/manpages/bionic/es/man2/shmctl.2.html>

[18] Ubuntu Manpage Repository. (2019). "semget" Disponible en:

<https://manpages.ubuntu.com/manpages/bionic/es/man2/semget.2.html>

[19] Ubuntu Manpage Repository. (2019). "semctl" Disponible en:

<https://manpages.ubuntu.com/manpages/bionic/es/man2/semctl.2.html>

[20] Ubuntu Manpage Repository. (2019). "semop" Disponible en:

<https://manpages.ubuntu.com/manpages/bionic/es/man2/semop.2.html>

[21] M . Kerrisk (2022). "pthread\_create" Disponible en:

[https://man7.org/linux/man-pages/man3/pthread\\_create.3.html](https://man7.org/linux/man-pages/man3/pthread_create.3.html)

[22] M . Kerrisk. (2022). "pthread\_join" Disponible en:

[https://man7.org/linux/man-pages/man3/pthread\\_join.3.html](https://man7.org/linux/man-pages/man3/pthread_join.3.html)

## Anexo. Códigos completos.

### Código del archivo de cabecera gestionarSemaforos.h

```
#include <stdio.h>

#include <stdlib.h>

#include<sys/sem.h>

#define PERMISOS 0644

//Defino un tipo de dato abstracto por medio de un struct para los
parametros

typedef struct
{
    int *resultados;

    int semaforo;
}Parametros;

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 de archivo Programa1.c

```

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

int sumaGauss(int n)
{
    return (n*(n+1))/2;
}

int factorial(int n)

```

```

{

    if (n<=1)

    {

        return 1;

    }

    return n*factorial(n-1);

}

int fibonacci(int n)

{

    if (n<=1)

    {

        return n;

    }

    return fibonacci(n-1) + fibonacci(n-2);

}

void Hilo1 (Parametros *parametros)

{

    int *resultados=(int*)parametros->resultados;

    int semaforo=(int)parametros->semaforo;

    printf("\nDentro del hilo 1-1 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());

    down(semaforo);

    resultados[0]=sumaGauss(100);

    up(semaforo);

```

```
    pthread_exit(0);
}

void Hilo2(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int semaforo=(int)parametros->semaforo;

    printf("\nDentro del hilo 1-2 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());

    down(semaforo);

    resultados[1]=factorial(10);

    up(semaforo);

    pthread_exit(0);
}

void Hilo3(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int i,semaforo=(int)parametros->semaforo;

    sleep(1);

    down(semaforo);

    printf("\nDentro del hilo 1-3 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());

    for(i=1;i<=20;i++)
```



```

    {
        resultados[i+1]=fibonacci(i);
    }

    up(semaphore);
    pthread_exit(0);
}

int main()
{

    pthread_attr_t atributos;
    pthread_t identificadores[3];    //Arreglo de 3 hilos

    int shmidl, semaphore;
    int *resultados;
    key_t llave1, llave_sem;
    Parametros parametros;

    //Se define el segmento de memoria compartida
    llave1=ftok("Archivo1", 'k');
    shmidl=shmget(llave1, 22*sizeof(int), IPC_CREAT|0777);    //Region de los
resultados

    parametros.resultados=(int *)shmat(shmidl, 0, 0);

    pthread_attr_init(&atributos);
    pthread_attr_setdetachstate(&atributos, PTHREAD_CREATE_DETACHED);
    printf("\nSoy el proceso padre 1 con ID: %d\n", getpid());

```

```

    //Crear semaforos

    llave_sem=ftok("S1",'k');

    parametros.semaforo=Crea_semaforo(llave_sem,1);

    sleep(2);

pthread_create(&identificadores[0],&atributos,(void*)Hilo1,(void*)&parametros);

pthread_create(&identificadores[1],&atributos,(void*)Hilo2,(void*)&parametros);

pthread_create(&identificadores[2],&atributos,(void*)Hilo3,(void*)&parametros);


    sleep(3);

    shmdt(&parametros.resultados);

    shmctl(shmid1,IPC_RMID,0);

    return 0;

}

```

## Código del archivo Programa2.c

```

#include<pthread.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include <sys/ipc.h>

```

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

#include <sys/wait.h>

#include "gestionarSemaforos.h"

void Hilo1 (Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;

    int semaforo=(int)parametros->semaforo;

    while(resultados[0]==0);

    down(semaforo);

    printf("\nDentro del hilo 2-1 con ID: %ld cuyo proceso es el %d\n",pthread_self(),getpid());

    printf("Resultado del hilo 1-1: %d\n",resultados[0]);

    up(semaforo);

    pthread_exit(0);
}

void Hilo2 (Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;

    int semaforo=(int)parametros->semaforo;

    while(resultados[1]==0);

    down(semaforo);
```

```

    printf("\nDentro del hilo 2-2 con ID: %ld cuyo proceso es el
%d\n",pthread_self(),getpid());

    printf("Resultado del hilo 1-2: %d\n",resultados[1]);

    up(semaforo);

    pthread_exit(0);
}

void Hilo3(Parametros *parametros)
{
    int *resultados=(int*)parametros->resultados;
    int i,semaforo=(int)parametros->semaforo;

    while(resultados[2]==0);

    down(semaforo);

    printf("\nDentro del hilo 2-3 con ID: %ld cuyo proceso es el
%d\n",pthread_self(),getpid());

    printf("Resultado del hilo 1-3:\n");

    for(i=2;i<22;i++)
    {
        printf("%d ",resultados[i]);

    }

    up(semaforo);

    printf("\n");

    pthread_exit(0);
}

```

```

int main()
{

    pthread_t identificadores[3];    //Arreglo de 3 hilos

    int shmid1, semaforo, j;

    key_t llave1, llave_sem;

    Parametros parametros;

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

    shmid1=shmget(llave1, 22*sizeof(int), IPC_CREAT|0777); //Region de los
resultados

    parametros.resultados=(int *)shmat(shmid1, 0, 0);

    printf("\nSoy el proceso padre 2 con ID: %d\n", getpid());

    //Crear semaforos

    llave_sem=ftok("S1", 'k');

    parametros.semaforo=Crea_semaforo(llave_sem, 1);

pthread_create(&identificadores[0], NULL, (void*)Hilo1, (void*)&parametros
);

pthread_create(&identificadores[1], NULL, (void*)Hilo2, (void*)&parametros
);

pthread_create(&identificadores[2], NULL, (void*)Hilo3, (void*)&parametros
);

```

```
for(j=0;j<3;j++)  
{  
    pthread_join(identificadores[j],NULL);  
}  
  
shmdt(&parametros.resultados);  
shmctl(shmid1,IPC_RMID,0);  
  
return 0;  
}
```