



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I Edgar Tista García

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 9

No de Práctica(s): 11

Integrante(s): Díaz Hernández Marcos Bryan

*No. de Equipo de
cómputo empleado:* Equipo Personal

No. de Lista o Brigada: 9

Semestre: 2021-1

Fecha de entrega: 13 de enero de 2021

Observaciones:

CALIFICACIÓN: _____

Objetivo de la practica

El estudiante conocerá y aprenderá a utilizar algunas de las directivas de openmp utilizadas para realizar programas paralelos.

Introducción

El siguiente reporte se realizó el análisis de los ejercicios de la guía del laboratorio y los semáforos, cada uno de acuerdo a la teoría vista en las clases, los análisis realizados desde una perspectiva de primera implementación del OpenMP, con la intención de poder comprender los elementos teóricos envueltos en el desarrollo de los ejemplos.

Ejercicios de la practica:

Sección 1:

Constructores:

- Semaphore(int permits): constructor que indica el número de hilos que pueden acceder a un conjunto de información, uno a la vez
- Semaphore(int permits, boolean fair): constructor que el número de hilos y también indica que los hilos tendrán acceso a la información en el orden que fueron requiriéndola.

Métodos:

- Acquire(): como su nombre lo indica este método otorga un permiso para poder acceder a la información, de forma que bloquea el acceso a hasta que esté disponible.
- Release(): permite el bloqueo del acceso una vez que el hilo que lo haya utilizado, termino con la ejecución.
- availablePermits(): indica el número de permisos disponibles en el semáforo
- IsFair(): regresa un booleano que indica si el semáforo acepta el orden de las solicitudes.

Explicación de los programas.

- SemaphoreDemo:

El programa permite la implementación de los semáforos por medio de los hilos, por lo que se tiene como atributos a una instancia de la clase Semaphore, además de que utiliza la clase Thread para poder asignar hilos de procesamiento.

Algo relevante dentro del programa es que se coloca al Semaphore como un atributo lo que hace necesario que en cada uno de los objetos de la clase sea necesario el colocar un objeto de Semaphore, para que pueda implementar los métodos.

```
class MyThread extends Thread
{
    Semaphore sem;
    String threadName;
```

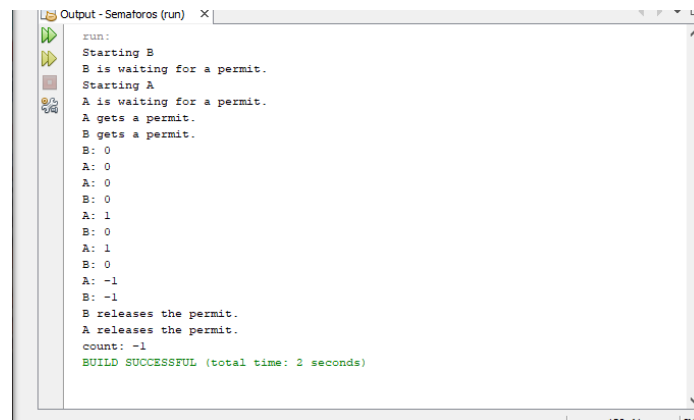
Herencia de Thread.

Dentro de la clase principal se crea un objeto de la clase Semaphore, para poder utilizar los métodos Acquire() y Release(), y posteriormente se crean las dos instancias de la clase MyThread, donde el constructor es la parte medular, ya que con este se le asigna a cada uno de los hilos que se crean el mismo semáforo que detendrá o permitirá la ejecución de un hilo o del otro para el procesamiento de la variable count que se encuentra en la clase Shared, en caso de que se crearan dos semáforos y se asignara uno a cada hilo, no se detendría la ejecución y los dos modificarían la variable al mismo tiempo.

```
// wait number of permits 1
Semaphore sem = new Semaphore(1);

// creating two threads with name A and B
// Note that thread A will increment t
// and thread B will decrement the count
MyThread mt1 = new MyThread(sem, "A");
MyThread mt2 = new MyThread(sem, "B");
```

Creación de los objetos.



```
run:
Starting B
B is waiting for a permit.
Starting A
A is waiting for a permit.
A gets a permit.
B gets a permit.
B: 0
A: 0
A: 0
B: 0
A: 1
B: 0
A: 1
B: 0
A: -1
B: -1
B releases the permit.
A releases the permit.
count: -1
BUILD SUCCESSFUL (total time: 2 seconds)
```

Asignando a cada Thread su propio semáforo.

Para poder ejecutar el semáforo se hace una sobre escritura del método run, el cual se activa por medio del método start() de la clase Thread. El método run() permite la ejecución de los hilos en base a los permisos que da el semáforo, y se plantea una condición que elige a uno de los dos hilos en base a su nombre, como ambos se ejecutan al mismo tiempo, se comienza un proceso de carrera para ver cuál de los dos hilos cumple primero la condición, una vez esto sucede se solicita el acceso, Acquire(), y al primero que se le otorgue puede modificar la variable count.

```
System.out.println(threadName + " is waiting for a permit.");

// acquiring the lock
sem.acquire();

System.out.println(threadName + " gets a permit.");
```

Acceso a la modificación de la variable.

Una vez que uno de los hilos haya terminado, se libera la variable por medio del `Release()` que libera el acceso al otro hilo. Una vez ejecutados los dos hilos se termina el programa y la variable regresa a su estado normal. En este caso el `sleep()` permite que la entrada a la variable por el hilo sea pausada 10 mls lo que permite ver mejor la interacción del hilo con la variable. Por esta razón del uso del `sleep()` se debe colocar el try-catch de la excepción de `InterruptedException`, ya que se detiene el procesamiento de un hilo de forma abrupta y puede generar una acción no contemplada dentro del programa.

- SemaphoreTest:

El programa realiza prácticamente lo mismo que el Demo, pero solo en este caso no existe un criterio para que cierto hilo se seleccione, sino que se ponen a todos los hilos en una condición de carrera, de tal forma que el primero que tenga acceso al `Acquire()` podrá comenzar a procesar la información. Así mismo se permite que cuatro de seis hilos tengan acceso al try,

```
A : got the permit.
B : is performing operation 1, available Semaphore permits : 3
F : is performing operation 1, available Semaphore permits : 0
E : is performing operation 1, available Semaphore permits : 0
C : is performing operation 2, available Semaphore permits : 0
F : is performing operation 2, available Semaphore permits : 0
B : is performing operation 2, available Semaphore permits : 0
E : is performing operation 2, available Semaphore permits : 0
B : is performing operation 3, available Semaphore permits : 0
C : is performing operation 3, available Semaphore permits : 0
F : is performing operation 3, available Semaphore permits : 0
E : is performing operation 3, available Semaphore permits : 0
F : is performing operation 4, available Semaphore permits : 0
B : is performing operation 4, available Semaphore permits : 0
C : is performing operation 4, available Semaphore permits : 0
E : is performing operation 4, available Semaphore permits : 0
C : is performing operation 5, available Semaphore permits : 0
F : is performing operation 5, available Semaphore permits : 0
B : is performing operation 5, available Semaphore permits : 0
E : is performing operation 5, available Semaphore permits : 0
F : releasing lock...
```

Ejecución de 4 hilos de 6.

Y una vez los primeros cuatro lograron terminar las operaciones, terminan su vida y los restantes dos comienzan a realizar las operaciones, para concluir con el programa.

```
A : releasing lock...
A : available Semaphore permits now: 3
D : releasing lock...
D : available Semaphore permits now: 4
```

Se van liberando los espacios.

En cuanto a la forma que se implementa el semáforo de acuerdo con el paradigma se toma como un atributo estático el objeto de la clase Semaphore, lo cual permite que todos los objetos de la clase puedan acceder a esta y ejecutar los métodos que permitan el modificar o no.

```
public class SemaphoreTest {
    // max 4 people
    static Semaphore semaphore = new Semaphore(4); //Numero de ele
```

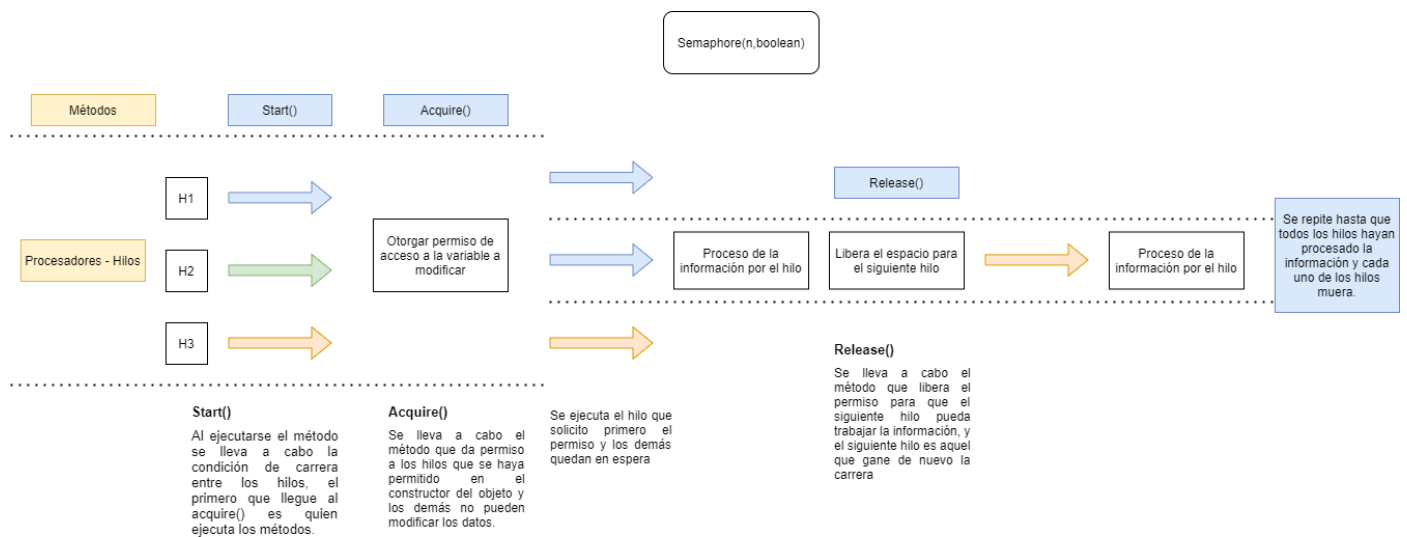
Objeto de Semaphore como un atributo estático.

- Indica en que aplicaciones se podrían utilizar

Por la experiencia de otros proyectos creo se podría aplicar a la Polifase e incluso al Radix, siento que este tipo de algoritmos paralelos son muy útiles para los algoritmos de ordenamiento, igual se le podría aplicar al QuickSort o el MergeSort, ya que son procesos que se van fragmentando y se podrían resolver con hilos y el momento de la unión que uno de los hilos lo resolviera.

Incluso que es posible utilizarlo como en un servidor, ya que puede ordenar o controlar el número de solicitudes que se pueden manejar en un determinado tiempo, o en una cierta actividad donde es necesario acceder por n usuarios.

- Diagrama de funcionamiento:



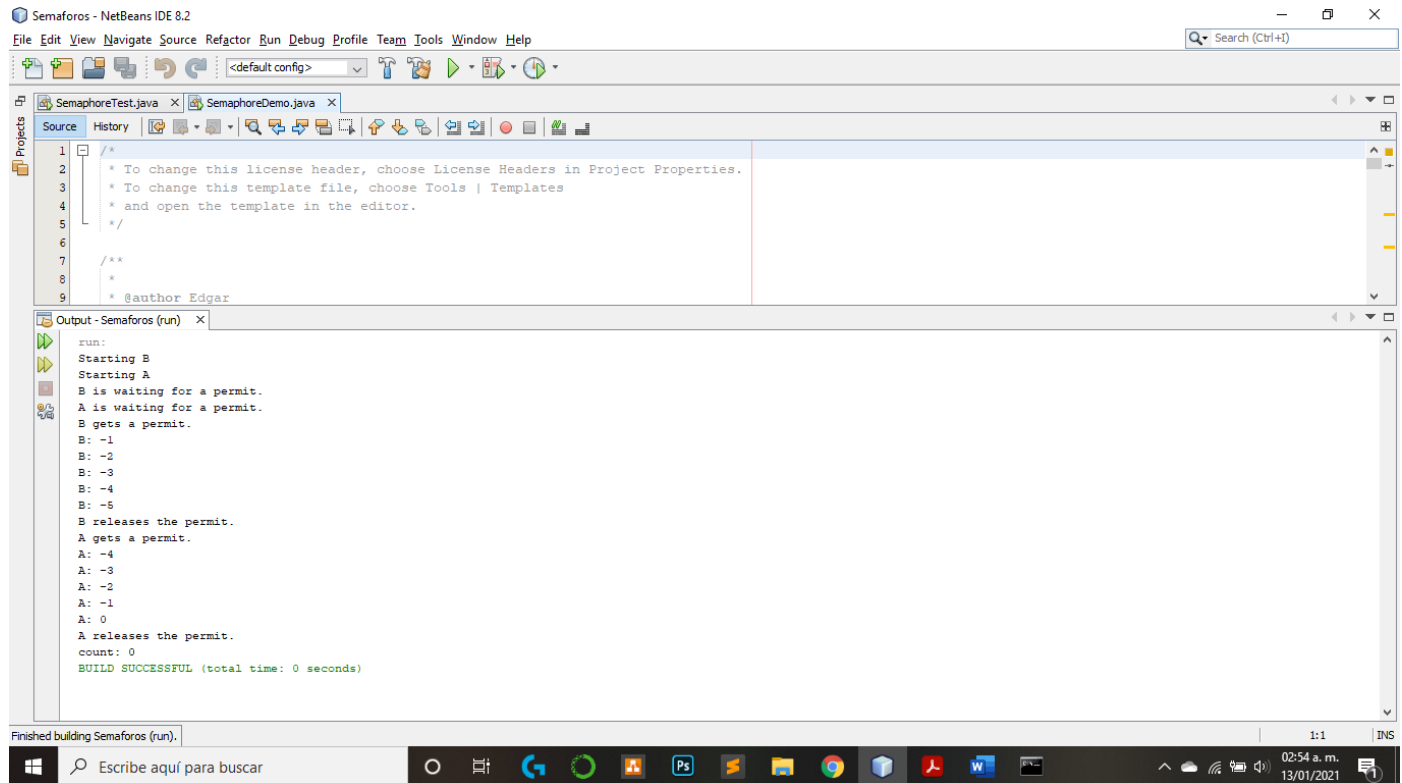
- Relación con la teoría:

Con lo visto en clase con respecto a la forma de comunicación que establecen los algoritmos de paralelismo, pues este es uno de ellos, ya que permite que los datos no sean modificados por n hilos, sino que regula la interacción de cada uno con la información y en base a esto limita la cantidad de los hilos que pueden estar modificando al mismo tiempo. Lo que tiene sus ventajas en cuanto a la integridad de la información, pero que no cumple al cien por ciento con ser paralelo ya que es más concurrente por esta condición.

En base a lo visto en la clase la granularidad que posee es fina pero con tintes de gruesa, ya que no se comunican directamente, sino que hay una clase que permite que sean controlados por medio de un permiso, lo que podría significar un que sea un paralelismo de Farm, ya que son controlados los hilos por medio de una Manager que sería el permiso y que cuando este da la opción de poder entrar a modificar los datos los hilos comienzan a trabajar.

Por último el nivel de paralelismo, la dependencia que existe no es tanta, ya que lo que haga un hilo con la información no afecta al otro, por lo menos con las sentencias que se colocaron, así que no hay dependencia entre los resultados, por lo que es un paralelismo a nivel de instrucción.

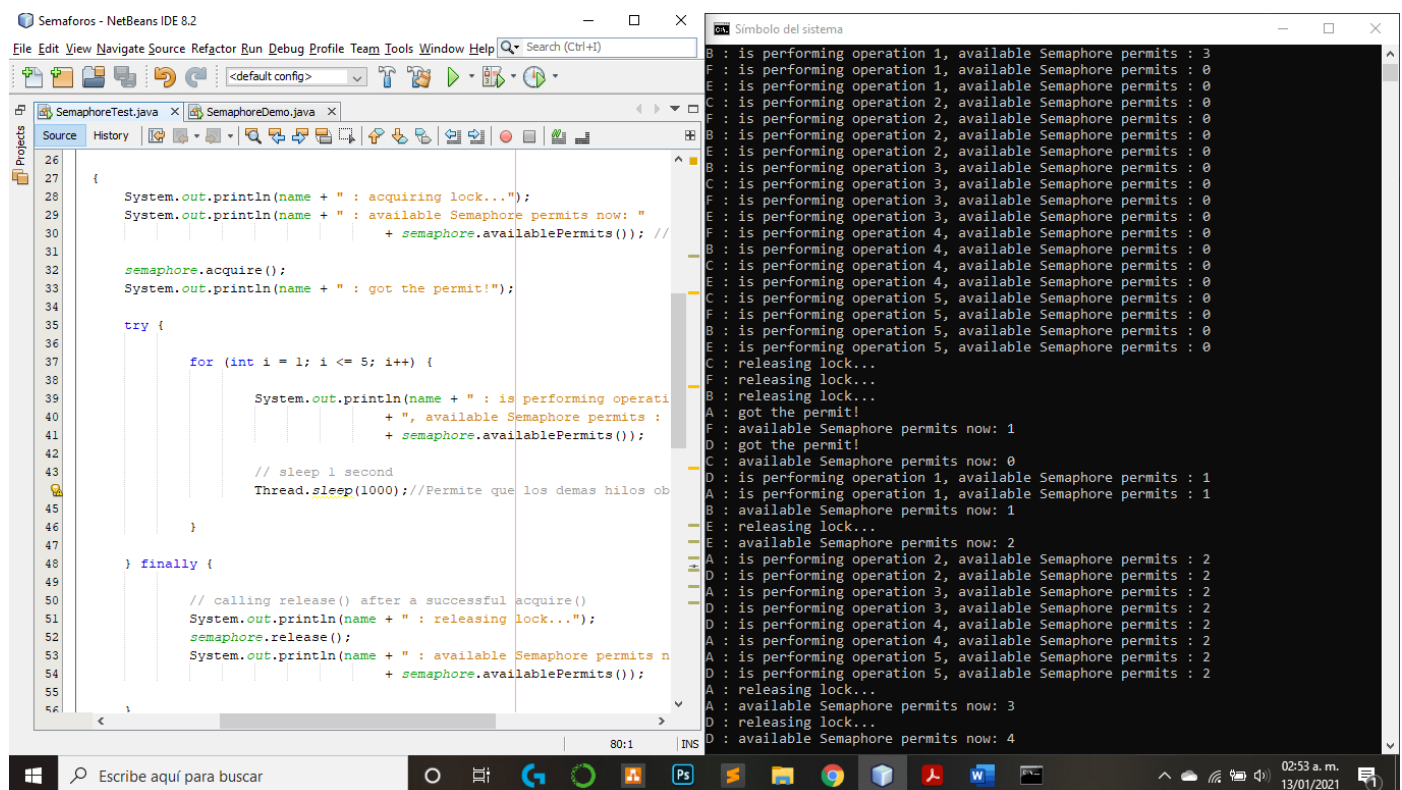
- Evidencia de implementación



The screenshot shows the NetBeans IDE with the file `SemaphoreDemo.java` open. The code is a simple test for a semaphore. The output window shows the execution results:

```
run:
Starting B
Starting A
B is waiting for a permit.
A is waiting for a permit.
B gets a permit.
B: -1
B: -2
B: -3
B: -4
B: -5
B releases the permit.
A gets a permit.
A: -4
A: -3
A: -2
A: -1
A: 0
A releases the permit.
count: 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

SemaphoreDemo



The screenshot shows the NetBeans IDE with the file `SemaphoreTest.java` open. The code is a more complex test for a semaphore, involving multiple threads and operations. The output window shows the execution results:

```
B : is performing operation 1, available Semaphore permits : 3
F : is performing operation 1, available Semaphore permits : 0
E : is performing operation 1, available Semaphore permits : 0
C : is performing operation 2, available Semaphore permits : 0
F : is performing operation 2, available Semaphore permits : 0
B : is performing operation 2, available Semaphore permits : 0
E : is performing operation 2, available Semaphore permits : 0
C : is performing operation 3, available Semaphore permits : 0
F : is performing operation 3, available Semaphore permits : 0
E : is performing operation 3, available Semaphore permits : 0
B : is performing operation 4, available Semaphore permits : 0
C : is performing operation 4, available Semaphore permits : 0
E : is performing operation 4, available Semaphore permits : 0
F : is performing operation 5, available Semaphore permits : 0
B : is performing operation 5, available Semaphore permits : 0
E : is performing operation 5, available Semaphore permits : 0
C : releasing lock...
F : releasing lock...
B : releasing lock...
A : got the permit!
F : available Semaphore permits now: 1
D : got the permit!
C : available Semaphore permits now: 0
D : is performing operation 1, available Semaphore permits : 1
A : is performing operation 1, available Semaphore permits : 1
B : available Semaphore permits now: 1
E : releasing lock...
E : available Semaphore permits now: 2
A : is performing operation 2, available Semaphore permits : 2
D : is performing operation 2, available Semaphore permits : 2
A : is performing operation 3, available Semaphore permits : 2
D : is performing operation 3, available Semaphore permits : 2
D : is performing operation 4, available Semaphore permits : 2
A : is performing operation 4, available Semaphore permits : 2
A : is performing operation 5, available Semaphore permits : 2
D : is performing operation 5, available Semaphore permits : 2
A : releasing lock...
A : available Semaphore permits now: 3
D : releasing lock...
D : available Semaphore permits now: 4
```

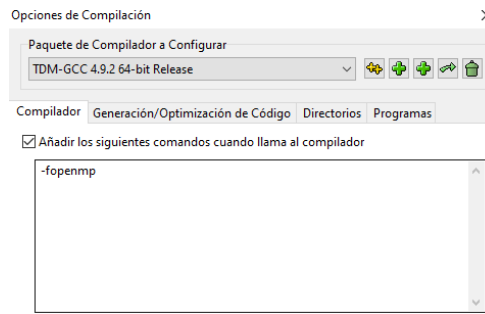
SemaphoreTest

Sección 2:

Actividad 1, 2 y 3:

- Análisis:

Estas dos primeras actividades permitieron que pudiera comprender el uso del OpenMP, porque en un determinado momento al querer instalarlo le estaba dando muchas vueltas, ya que desde el Dev-C++ es posible implementarlo sin instalar nada solo modificando la consola y permitiendo ciertos comandos.



Habilita el uso de #pragma.

Además de ese primer acercamiento en esta actividad se comienza a dar un paralelismo, que es la sentencia `#pragma omp parallel`, con la cual es posible dar la condición de paralelismo para el conjunto de instrucciones que están en el scope de la sentencia. Por otro lado esta sentencia da una condición de carrera, lo que significa que los n procesadores – n hilos, comienzan a competir por entrar a resolver o ejecutar las operaciones que se indican.

De igual forma se la arquitectura del OpenMP es de tipo tenedor (fork-join) ya que con esto permite que sea paralela una parte de código y después se tenga que cumplir la unión de los hilos, o la ejecución de un hilo maestro, para la resolución de una determinada sentencia. Esto significa que es concurrente.

Una vez que se coloca el `#pragma`, se puede determinar el número de hilos que se van a utilizar además de poder incluso darle un número determinado de hilos para todo el programa, con la sentencia: `omp_set_num_threads(n)` y con la sentencia: `num_threads(4)`, seleccionar los hilos que se van a utilizar para esa parte del código.

```
int main(){
    omp_set_num_threads(8); //Indica los
    #pragma omp parallel num_threads(4)
    {
        . . .
    }
```

Establecer número de hilos.

No es totalmente cierto que se puedan establecer n hilos, ya que dependerá del número de procesadores – núcleos del procesador, y al establecer esta sentencia pues cada uno de los procesadores se reparte cierta cantidad de hilos, en este caso yo tengo 4 entonces le corresponde a cada uno un hilo.

Con lo anterior se retoma la velocidad de procesamiento que es mucho mayor a una secuencial, pero que al ser una condición de carrera, todos los hilos la ejecutan y eso hace que sea iterativo el programa por los hilos. Esto es porque los hilos tienen acceso a la misma variable ya que es como un atributo aplicable a cada uno de los hilos, entonces buscan el modificarlo todos. Esto causa que el orden de las operaciones sea impredecible y que se obtengan resultados no esperados.

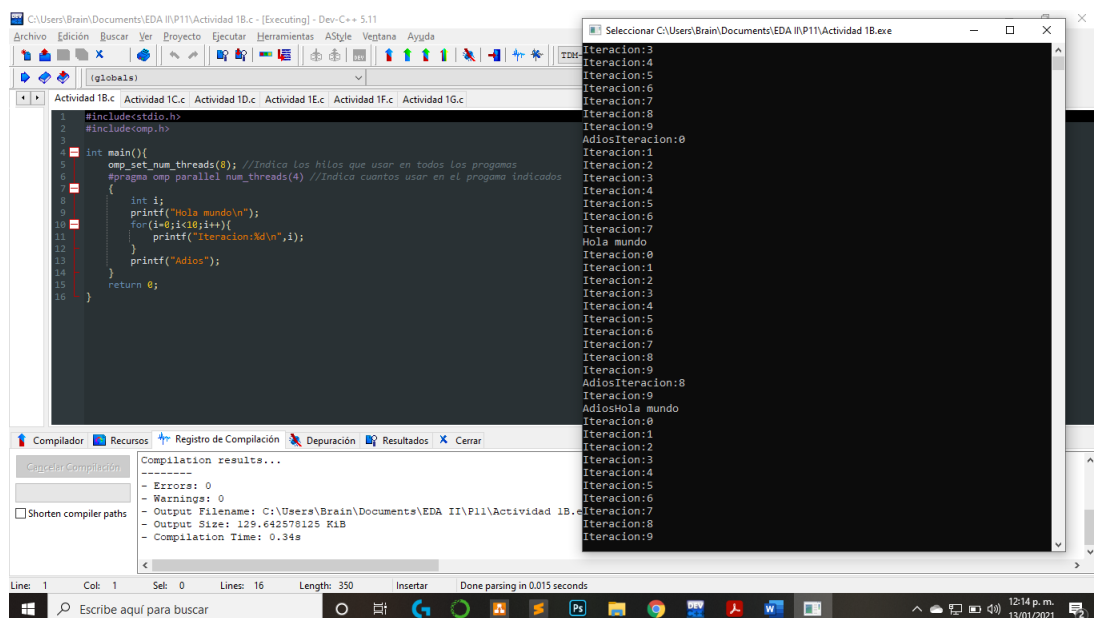
Algo importante en la forma en que implementan los hilos, es la privacidad de las variables, ya que es posible que las compartan como los objetos con las variables, que algunas son modificables por todos los objetos como las static, y que algunas son privadas para cada objeto, esto funciona de forma similar, ya que las variables que estén fuera de la sentencia `#pragma` o antes de esta, serán compartidas por cada hilo, y las que estén dentro de esta, serán privadas para cada hilo, es decir solo el hilo "a" podrá modificar su variable "b" y así con los n hilos.

```
int i;
omp_set_num_threads(8); //Indica los
#pragma omp parallel num_threads(4)
{
    printf("Hola mundo\n");
    for(i=0;i<10;i++){
        printf("Iteracion:%d\n",i);
    }
    printf("Adios");
}
```

Acceso a las variables: público o privado

Este tipo de algoritmos se me hacen bastante confusos en un inicio por la variedad de usos y aplicaciones que es posible darles, pero una vez comprendo lo que sucede o cual es la implicación teórica detrás, siento que son muy curiosos para su estudio ya que viéndolos de cierta forma como los de ordenamiento siento que tienen ciertos elementos clave que permiten ver como se procesa la información y que instrucción se llevan a cabo por cual hilo.

- Evidencia de implementación:



Actividad 4:

- Análisis:

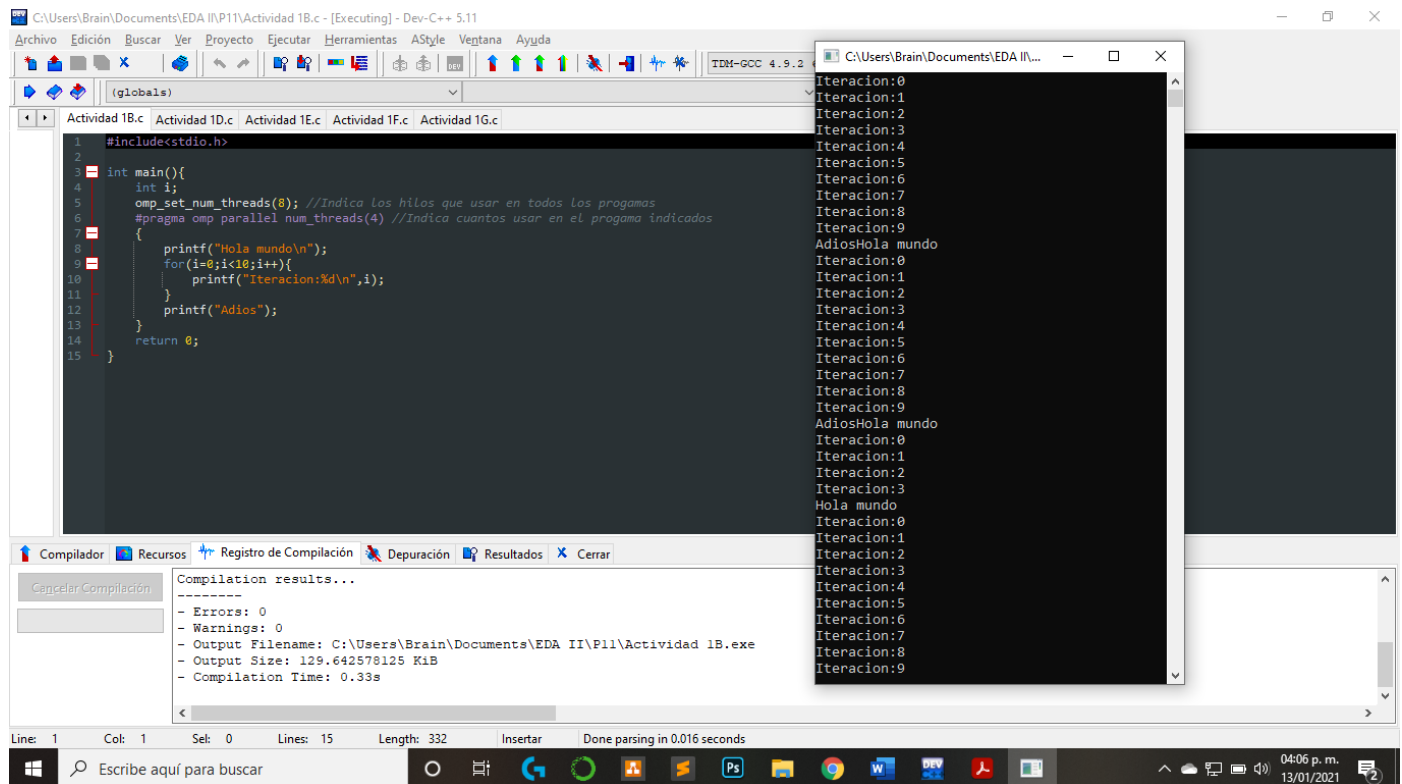
Dentro del `#pragma` existen las sentencias de acceso que son `Shared()` y `Private()`, están permiten como se mencionaba tener copias privada o compartida de las variables, en el caso del `Shared()` se toma por default a las variables que están fuera del scope de `#pragma`, y el `Private()` se aplica seguido del `#pragma`.

```
int main(){
    int i;
    #pragma omp parallel private(i)
```

Modificador de acceso.

En el caso del programa al hacer esto cada hilo conserva una copia de la variable, lo que permite que los cuatro hilos no modifiquen la misma variable y cada uno pueda ejecutar el ciclo con las iteraciones en orden.

- Evidencia de implementación:



Actividad 5:

- Análisis:

En la implementación del `#pragma` se pueden identificar a los hilos y al numero de hilos, mediante las funciones: `omp_get_num_threads()` y `omp_get_thread_num()` que permiten el obtener el número de hilos disponibles, y obtener el número de identificador del hilo respectivamente.

```
tid=omp_get_thread_num();
nth=omp_get_num_threads();
printf("Hola mundo desde el
```

Variables que conservan el identificador y numero de hilos.

En este caso la variable tid, guarda el identificador de los hilos, pero al ser una variable que se coloca con un acceso privado, cada uno de los hilos modifica su valor y lo imprime, de igual forma al quitar el acceso privado se comparte la variable tid a todos los hilos, lo que significa que cada uno la modifica pero al tener cada hilo un identificador único, los cambios no modifican los resultados finales ya que la impresión del identificador es aleatoria en un rango de [1-4] para el caso de mis procesadores.

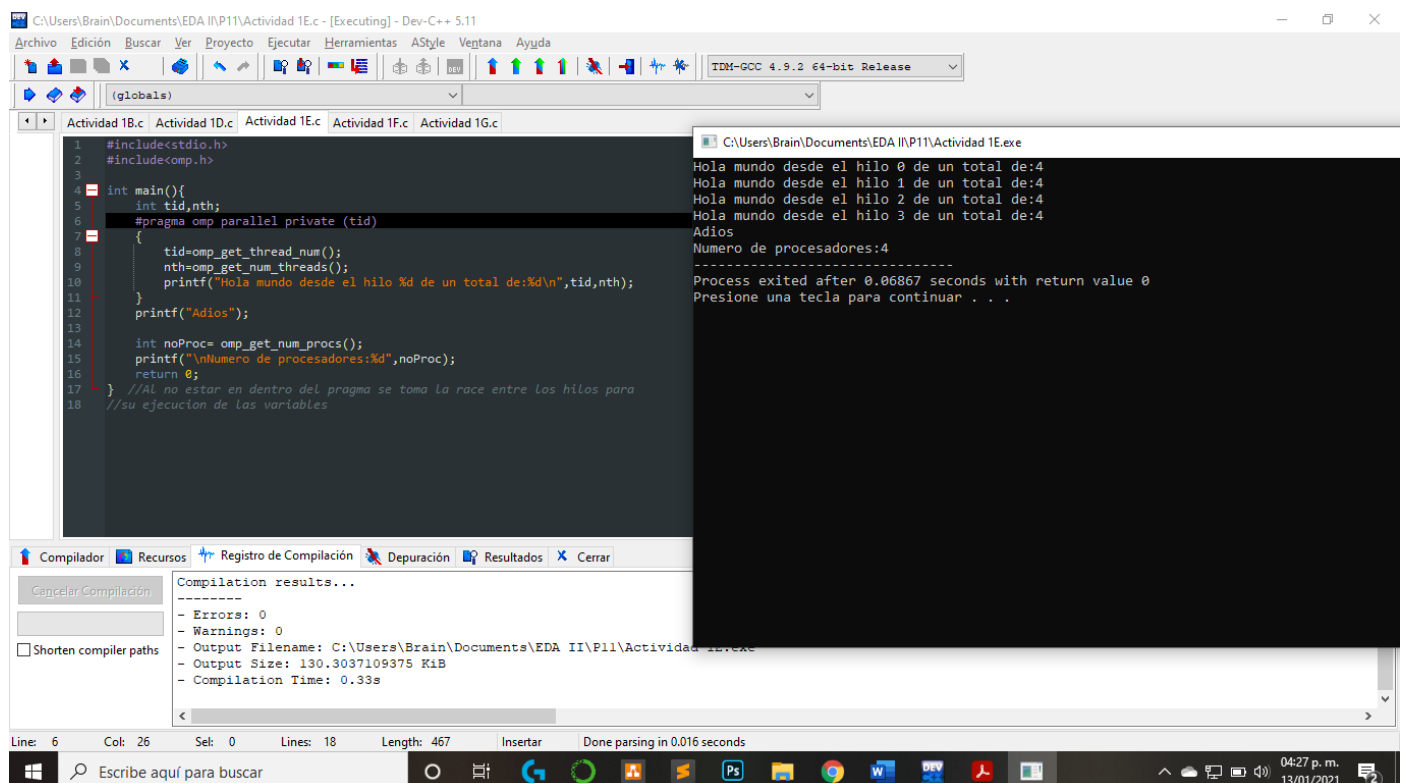
Además existe otra función que es capaz de poder indicar el numero de procesadores que nuestro equipo tiene: omp_get_num_procs(), que regresa el número de núcleos que el procesador tiene, en este caso son 4 núcleos.

```
int noProc= omp_get_num_procs();
printf("\nNumero de procesadores:%d",noProc);
return 0;
//Al no estar en dentro del pragma se toma la ra
```

Muestra la cantidad de procesadores.

Para poder utilizar las funciones es necesario incluir la biblioteca #include<omp.h>.

- Evidencia de implementación:



Actividad 6 - 8:

- Análisis:

Para el programa se planteó la suma de dos arreglos, la suma se parte en dos hilos, donde el primer hilo va del origen a la mitad del arreglo y el segundo hilo va de la mitad al final del arreglo. Se implementan arreglos dinámicos, después se llenan con la función llenar, y posteriormente en la función de suma se implementa el paralelismo.

Para realizar esto se consideran los límites de la suma en los arreglos, en base a la variable tid, que permite al hilo 0 sumar de la localidad [0-4], y al segundo hilo de [5-9], mediante el acceso privado las variables que están fuera del #pragma se crea una copia en cada hilo, posteriormente se tiene que cada hilo modifica los rangos mencionados, y se guardan los resultados en el tercer arreglo.

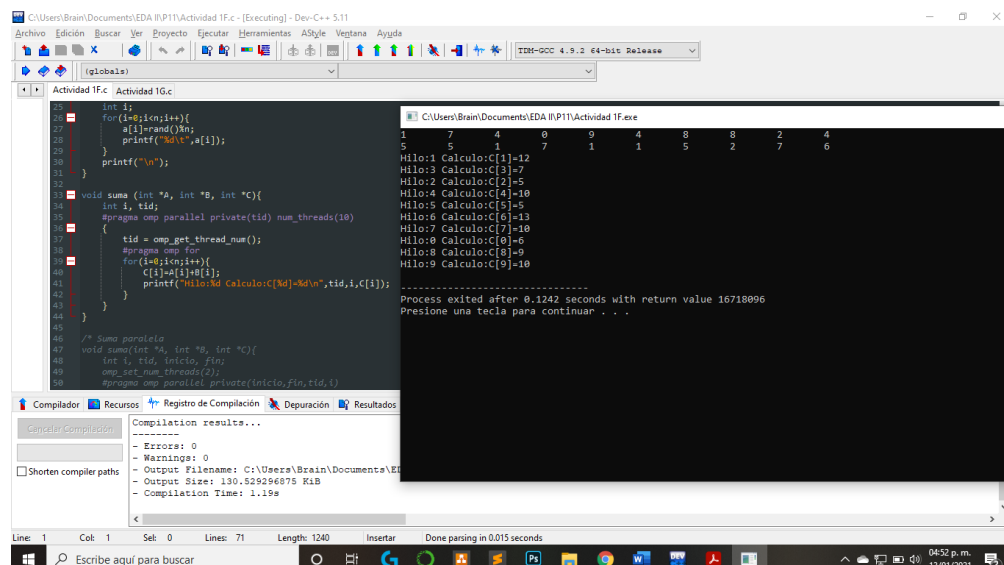
La última implementación de este código utiliza la sentencia: #pragma omp for que permite que los hilos se distribuyan los valores de una estructura for, por lo que entre cuatro de los hilos se pueden repartir las tareas de que están en el for, esto es posible siempre y cuando no haya dependencia en los valores que se generan dentro del for.

Con esta versión de la suma de los arreglos se pueden indicar la cantidad de hilos que van a estar en la partición del for, mediante la sentencia de num_threads(n), donde se coloca el valor de los hilos, y estos mediante la sentencia de #pragma omp for, se reparten las operaciones del for.

```
void suma (int *A, int *B, int *C){
    int i, tid;
    #pragma omp parallel private(tid) num_threads(4)
    {
        tid = omp_get_thread_num();
        #pragma omp for
```

Indicar la cantidad de hilos.

- Evidencia de implementación:



```
int i;
for(i=0;i<n;i++){
    a[i]=rand()%n;
    printf("a[%d]=%d\n",i,a[i]);
}
printf("\n");

void suma (int *A, int *B, int *C){
    int i, tid;
    #pragma omp parallel private(tid) num_threads(10)
    {
        tid = omp_get_thread_num();
        for(i=0;i<n;i++){
            C[i]=A[i]+B[i];
            printf("Hilo:%d Calculo:C[%d]=%d\n",tid,i,C[i]);
        }
    }

    /* Suma paralela
    void suma(int *A, int *B, int *C){
        int i, tid, inicio, fin;
        omp_set_num_threads(2);
        #pragma omp parallel private(inicio,fin,tid,i)
        {
            inicio = 0;
            fin = n;
            #pragma omp for
            for(i=0;i<n;i++){
                C[i]=A[i]+B[i];
            }
        }
    }
    */
```

Process exited after 0.1242 seconds with return value 16718096
Presione una tecla para continuar . . .

Actividad 7:

- Análisis:

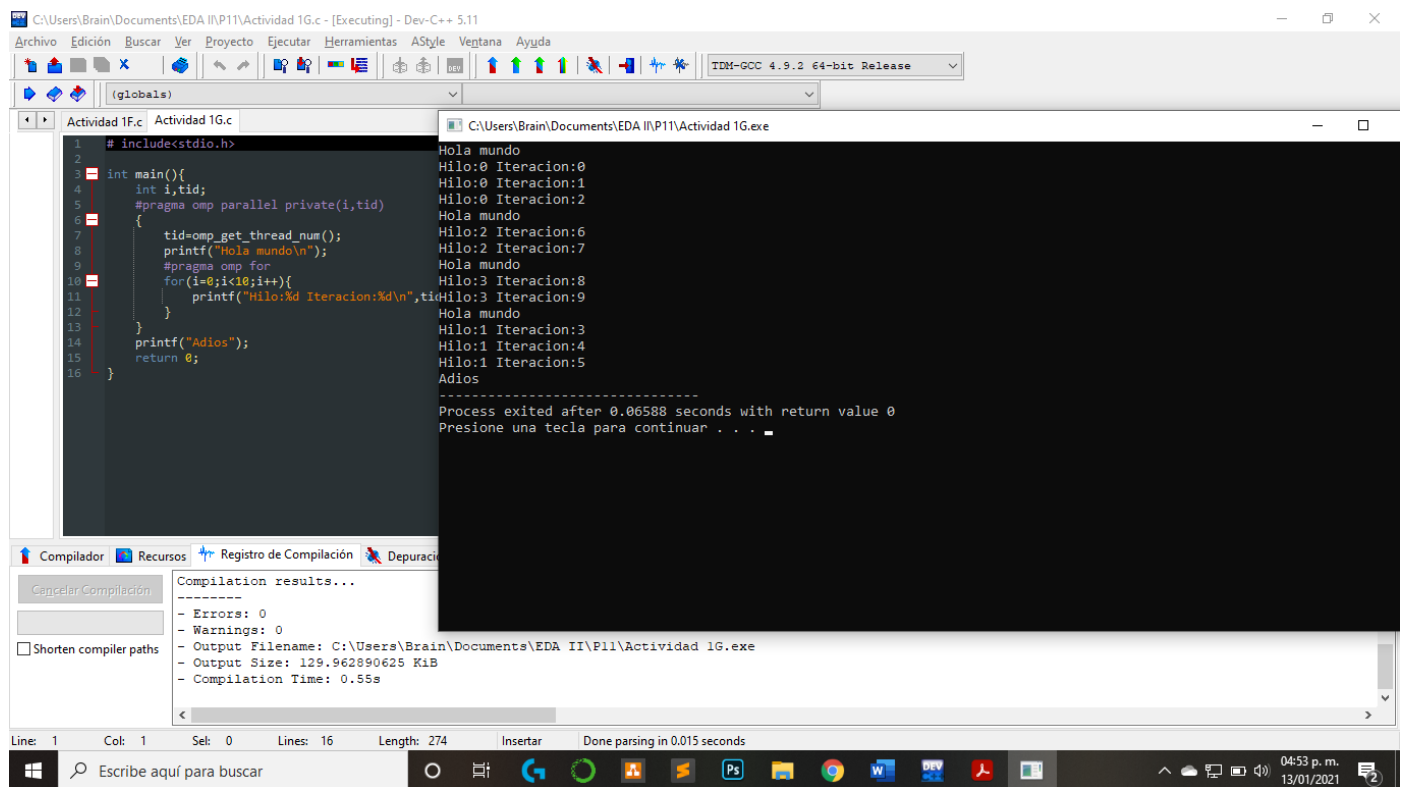
La actividad consistió en implementar la sentencia `#pragma omp for`, en el código de la primera actividad, y agregando el `tid`, o la sentencia: `omp_get_thread_num()`, se puede verificar que hilos hacen las iteraciones y como se reparten las operaciones del ciclo `for`.

```
int i,tid;
#pragma omp parallel private(tid)
{
    tid=omp_get_thread_num();
    printf("Hola mundo\n");
    #pragma omp for
    for(i=0;i<10;i++){
        printf("Hilo:%d Iteracion:%d\n",tid,i);
    }
}
```

División del ciclo `for` entre los hilos.

De esta forma es más rápida la ejecución del ciclo `for` y es posible visualizar que hace cada uno de los hilos que ejecutan las sentencias del código.

- Evidencia de implementación:



```
#include<stdio.h>
int main(){
    int i,tid;
    #pragma omp parallel private(i,tid)
    {
        tid=omp_get_thread_num();
        printf("Hola mundo\n");
        #pragma omp for
        for(i=0;i<10;i++){
            printf("Hilo:%d Iteracion:%d\n",tid,i);
        }
    }
    printf("Adios");
    return 0;
}
```

Output:

```
Hola mundo
Hilo:0 Iteracion:0
Hilo:0 Iteracion:1
Hilo:0 Iteracion:2
Hola mundo
Hilo:2 Iteracion:6
Hilo:2 Iteracion:7
Hola mundo
Hilo:3 Iteracion:8
Hilo:3 Iteracion:9
Hola mundo
Hilo:1 Iteracion:3
Hilo:1 Iteracion:4
Hilo:1 Iteracion:5
Adios
-----
Process exited after 0.06588 seconds with return value 0
Presione una tecla para continuar . . .
```

Compilation results...

```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Brain\Documents\EDA II\P11\Actividad 1G.exe
- Output Size: 129.962890625 KiB
- Compilation Time: 0.55s
```

Conclusiones.

En la práctica lleve a cabo la ejecución de los distintos ejercicios, además de buscar un poco más de cada uno para poder comprender mejor, de igual forma trate de comprender como utilizar las distintas funciones, además de aprender a cómo utilizar el paralelismo en C, y de esa forma es como pude comprender el OpenMP y cumplir con el objetivo de la práctica.