



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

# Laboratorios de computación salas A y B

*Profesor:* Tista García Edgar Ing.

*Asignatura:* Estructura de Datos y Algoritmos II

*Grupo:* 5

*No de Práctica(s):* 11

*Integrante(s):* Téllez González Jorge Luis

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

*No. de Lista o Brigada:* X

*Semestre:* 2020-2

*Fecha de entrega:* 16/05/2020

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_



# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Objetivos</b>	<b>4</b>
<b>3. Semáforos</b>	<b>4</b>
3.1. Clase Semaphore . . . . .	5
3.2. Análisis . . . . .	6
3.2.1. SemaphoreDemo . . . . .	7
3.2.2. SemaphoreTest . . . . .	9
<b>4. Introducción a OpenMP</b>	<b>11</b>
4.1. Actividad 1 . . . . .	11
4.2. Actividad 2 . . . . .	13
4.3. Actividad 3 . . . . .	14
4.4. Actividad 4 . . . . .	15
4.5. Actividad 5 . . . . .	16
4.6. Actividad 6 . . . . .	17
4.7. Actividad 7 . . . . .	18
4.8. Actividad 8 . . . . .	19
<b>5. Conclusiones</b>	<b>19</b>

# 1. Introducción

La sincronización de recursos es un problema frecuente en los programas que utilizan procesos de concurrencia. Si los diferentes procesos de un programa concurrente tienen acceso a secciones comunes de memoria, la transferencia de datos a través de ella representa una forma habitual de comunicación y sincronización entre ellos. No obstante, tal vía de comunicación puede llevar a errores de ejecución debido a que la acción de un proceso puede terminar interfiriendo con las acciones de otro.

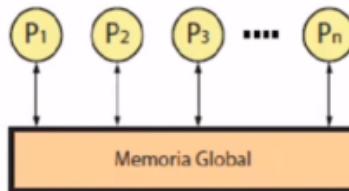


Figura 1: Memoria global compartida por 2 o más procesos de un programa.

Con el fin de evitar tales errores es posible identificar ciertas regiones de los procesos que acceden a variables en memoria compartidas y darles la posibilidad de ejecutarse como una única instrucción. Se denomina como **sección crítica** a aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma **concurrente**; desde el punto de vista de otro proceso pueden visualizarse como si fuesen una única instrucción. Lo anterior implica que si un proceso inicia su ejecución haciendo uso de una sección crítica de variables compartidas, otro proceso no puede ejecutarse con tal sección crítica.

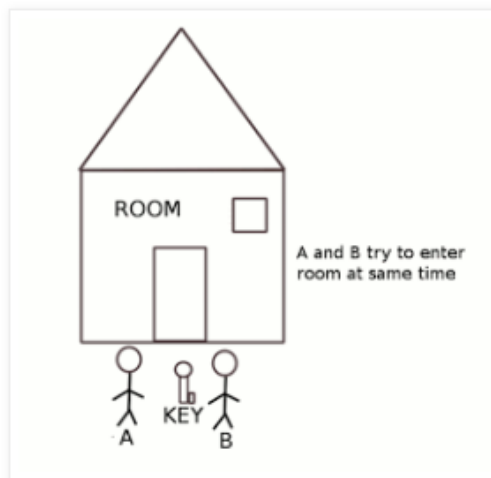


Figura 2: Una sección crítica puede imaginarse como una situación donde dos personas intentan entrar a una casa al mismo tiempo.

El científico de la computación **Edsger Dijkstra** propuso en los años 60's una solución conceptual al problema de las secciones críticas con el fin de lograr una adecuada sincronización de los procesos

en un entorno recurrente. A esta técnica se le denominó como *Semáforo* y permite resolver gran parte de los problemas de sincronización entre procesos, por tanto, forma parte del diseño de muchos sistemas operativos.

Su uso permite resolver problemas frecuentes en la programación recurrente como el caso del *productor-consumidor*, el *lector-escritor* y el famoso *problema de los filósofos* que ilustra la importancia del adecuado manejo de los recursos compartidos.

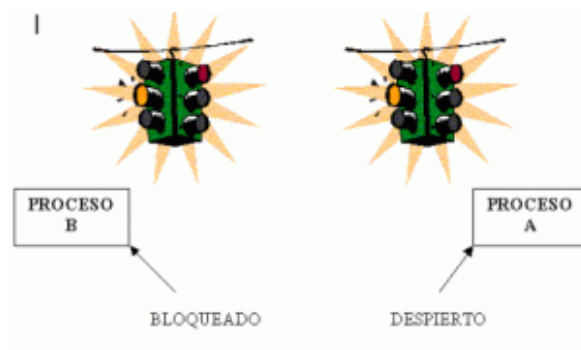


Figura 3: *Semáforo como vía de sincronización de acceso a un recurso compartido en memoria.*

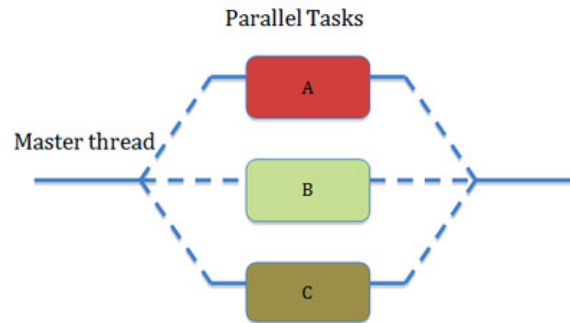
Además de las bibliotecas ofrecidas por **Java** para la escritura de programas paralelos, se encuentra la interfaz de programación de aplicaciones **OpenMP**, utilizada para la escritura de programas que aprovechen el paradigma paralelo y que cuenten una arquitectura de memoria compartida. Esta API brinda herramientas útiles para escribir programas paralelos en lenguajes como C, C++ o Fortran.



Figura 4: *API OpenMP.*

**OpenMP** trabaja haciendo uso de la arquitectura *fork-join*, en el que a partir de un proceso o hilo principal se generan un cierto número de hilos que serán utilizados para ejecutarse paralelamente en una región denominada *zona paralela*. Posteriormente, se unirán para finalmente tener únicamente el hilo principal. Por lo anterior, es posible afirmar que **OpenMP** combina código *serial* y *paralelo*.

La paralelización de un programa por medio de esta herramienta debe de hacerse de forma **explícita**, es decir, es tarea del programador analizar e identificar las partes que pueden realizarse de forma concurrente y, por tanto, que puedan utilizar un conjunto de hilos para resolver un determinado problema.

Figura 5: *Modelo Fork-Join.*

En el siguiente reporte escrito se revisará la clase *Semaphore* implementada en **Java**, así mismo, se analizarán las clases del proyecto *Semaforos* proporcionado. Finalmente, se realizarán y analizarán los ejemplos contenidos en la *Práctica 11* de la guía del laboratorio de EDA II haciendo uso del lenguaje C y de **OpenMP**. Para escribir, compilar y ejecutar los ejemplos, se ha optado por utilizar el IDE **CodeBlocks** con el soporte habilitado para OpenMP.

## 2. Objetivos

- El estudiante conocerá y aprenderá a utilizar algunas de las directivas de OpenMP utilizadas para realizar programas paralelos.
- Comprender el concepto de Semáforos y analizar a detalle la implementación del concepto aplicado en Java.

## 3. Semáforos

Los *semáforos* pueden definirse como componentes pasivos de bajo nivel de abstracción utilizados para arbitrar el acceso a un recurso compartido de memoria. Suelen ser de fácil comprensión y de gran capacidad funcional, sin embargo, su uso inadecuado puede resultar en errores graves durante la ejecución de un programa paralelo. Cada semáforo tiene asociado una **lista de procesos**, en la que se incluyen todos los procesos que se encuentran en estado de *suspensión* a la espera de acceder al recurso compartido en cuestión.

Un semáforo representa a un tipo de abstracto de dato, y sus dos atributos básicos se refieren al conjunto de valores que puede tomar y el conjunto de operaciones que admite. Un semáforo *binario* es aquel que puede tomar solo valores 0 y 1 (Abierto/Cerrado), en cambio, un semáforo *general* que puede tomar cualquier valor natural.

### 3.1. Clase Semaphore

La clase *Semaphore* se define como una implementación de un semáforo que controla el acceso a un recurso compartido haciendo uso de un contador: si el contador es mayor a 0, el acceso se permite. En cambio, si el valor del contador es 0, el acceso es denegado. Este contador registra los permisos que conceden el acceso al recurso compartido.

De forma general, un hilo que desea acceder a un recurso compartido en primer lugar intenta adquirir un permiso de acceso por medio del semáforo. En caso de que el permiso no sea concedido, el hilo quedará bloqueado hasta que pueda acceder al permiso. Cuando un hilo ya no requiere acceso al recurso compartido, libera el permiso concedido; lo cuál provocará que el contador del semáforo vuelva a incrementarse. Si en ese momento existe otro hilo esperando por un permiso, el permiso liberado le será concedido.

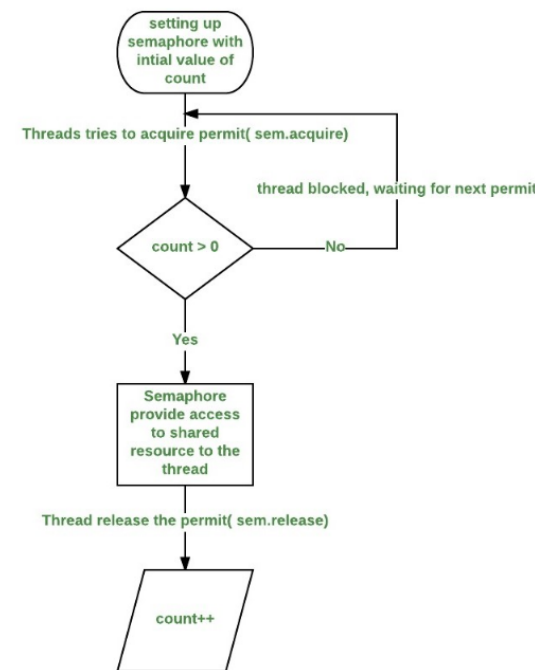


Figura 6: Diagrama de flujo del funcionamiento de un semáforo en Java.

Esta clase cuenta con dos constructores principales:

*Semaphore(int permits)*

El parámetro *num* especifica el valor inicial de permisos. Es decir, se especifica el número de hilos que pueden acceder al recurso compartido al mismo tiempo: si su valor es uno, entonces únicamente un hilo puede acceder a tal recurso en un tiempo determinado.



*Semaphore(int permits, boolean fair)*

Por defecto, a todos los hilos que esperan un permiso se les concede en un orden no definido. Un valor booleano *true* en el segundo parámetro del constructor asegura que a todos los hilos que esperan un permiso se les concederán de forma ordenada conforme lo solicitaron. El valor *false* provocará que el orden de otorgamiento de permisos no sea ordenado.

A continuación se enlistan los métodos más destacables de esta clase:

- **acquire()**: Permite que un hilo adquiera un permiso del semáforo, permaneciendo con el mismo hasta que termine de utilizarlo o su ejecución sea interrumpida.
- **acquireUninterruptibly()**: Funciona similar al método **acquire**, sin embargo, el permiso se desbloquea únicamente hasta que el hilo termina su ejecución sin ser interrumpido.
- **availablePermits()**: Regresa el número actual de permisos disponibles en el semáforo.
- **drainPermits()**: Regresa todos los permisos disponibles actualmente.
- **getQueuedThreads()**: Permite conocer los hilos que se encuentran en espera de obtener un permiso de acceso a un recurso compartido.
- **getQueueLength()**: Devuelve la cantidad de hilos que esperan permisos del semáforo.
- **release()**: Libera un permiso y lo devuelve al semáforo.
- **reducePermits(int reduction)**: Reduce el número de permisos existentes en el semáforo de acuerdo al valor introducido como parámetro.
- **tryAcquire()**: Cualquier objeto de tipo Thread que invoque a este método tratará de adquirir un permiso del semáforo, el cual solo se le otorgará si al momento de realizarse la solicitud se encuentra un permiso disponible. En caso de no estar disponible, no se le agrega a un lista de espera; como sucede con el método **acquire**.

### 3.2. Análisis

El proyecto de **NetBeans** proporcionado para la practica contiene dos clases: *SemaphoreDemo* y *SemaphoreTest*, las cuales se analizarán a continuación.

### 3.2.1. SemaphoreDemo

La clase *SemaphoreDemo* contiene en su interior otras clases que declaran elementos para la ejecución del método *main* que se encuentra en su interior:

- La clase *Shared* contiene un atributo estático de tipo entero denominado *contador*, el cual se encuentra inicializado en 0. Este atributo representa el recurso compartido al que el semáforo concederá o no un permiso de acceso.

La clase *MyThread* hereda de *Thread* e inicia haciendo una declaración del objeto de tipo *Semaphore* y una cadena utilizada para almacenar el nombre de un determinado hilo. Posteriormente, se encuentra el constructor de la clase, el cuál inicializa un hilo haciendo referencia al constructor de *Thread*, y asignándole un nombre dado y el semáforo en cuestión. Los hilos que modela esta clase en particular serán los utilizados para ejemplificar el funcionamiento del semáforo.

```
super(threadName);  
this.sem = sem;  
this.threadName = threadName;
```

Figura 7: Inicialización de un hilo.

Esta clase contiene una sobreescritura al método **run** y considera dos situaciones:

1. Si el hilo que invoca al método **run** tiene como nombre A, se tratará de adquirir un permiso para el mismo. En caso de que el permiso sea concedido por el semáforo, se imprimirá un mensaje confirmando la operación y el hilo en cuestión por medio de un ciclo **for** incrementará el valor de la variable compartida *contador* un total de 5 veces. Posteriormente, el hilo en cuestión se envía a dormir un total de 10 milisegundos para permitir al hilo B inicializarse y solicitar un permiso. *El procedimiento se encuentra en un bloque try-catch con el fin detectar cualquier interrupción en la ejecución de uno de los hilos.*

```
// Ahora, se accede al recurso compartido.  
// Otros hilos en espera permanecieran en ese estado hasta que  
// este hilo libere el permiso.  
for (int i = 0; i < 5; i++) {  
    //Se incrementa el valor de la variable compartida.  
    Shared.contador++;  
    System.out.println(threadName + ": " + Shared.contador);  
  
    // Ahora, permitiendo un cambio en la ejecución,  
    // de ser posible, para permitir al hilo B ejecutarse.  
    Thread.sleep(10);  
}
```

Figura 8: Operación del hilo A.



Finalmente, se libera el permiso por medio del método **release** y se imprime un mensaje en pantalla confirmando la liberación del mismo.

2. Si el hilo que invoca al método `run` tiene como nombre *B*, el procedimiento anterior será seguido de forma idéntica, con la diferencia de que ahora en el ciclo **for** la variable compartida será decrementada un total de 5 veces.

Finalmente, la clase principal de prueba *SemaphoreDemo* contiene en su interior el método principal donde se crea un nuevo objeto de tipo *Semaphore* el cuál se inicializa con un único permiso. Posteriormente, se crean dos hilos de tipo *MyThread* denominados *A* y *B*, respectivamente. A ambos hilos se les asigna el semáforo inicializado por medio de su constructor.

```
// Se crea un objeto de tipo Semaphore
// con un único permiso (Numero de permisos: 1)
Semaphore sem = new Semaphore(1);

// Se crean dos hilos con nombres A y B, respectivamente
// Notese que el hilo A incrementará el contador
// y el hilo B lo decrementará
MyThread mt1 = new MyThread(sem, "A");
MyThread mt2 = new MyThread(sem, "B");
```

Figura 9: Inicialización del semáforo y los hilos

Se inicializa la ejecución de ambos hilos por medio del método **start**. Además, se utiliza el método **join** para sincronizar la ejecución de ambos hilos y que uno de los hilos se mantenga a la espera de que el otro termine su ejecución. Finalmente, se imprime el valor final de la variable contadora compartida por ambos hilos; que debe de tener como valor 0 una vez que ambos hilos terminen su ejecución exitosamente.

run:	run:
Iniciando B	Iniciando B
Iniciando A	Iniciando A
A está esperando por un permiso.	B está esperando por un permiso.
A ha obtenido un permiso.	A está esperando por un permiso.
B esta esperando por un permiso.	B ha obtenido un permiso.
A: 1	B: -1
A: 2	B: -2
A: 3	B: -3
A: 4	B: -4
A: 5	B: -5
A ha liberado el permiso.	B ha liberado el permiso.
B ha obtenido un permiso.	A ha obtenido un permiso.
B: 4	A: -4
B: 3	A: -3
B: 2	A: -2
B: 1	A: -1
B: 0	A: 0
B ha liberado el permiso.	A ha liberado el permiso.
Contador: 0	Contador: 0
BUILD SUCCESSFUL (total time: 0 seconds)	BUILD SUCCESSFUL (total time: 0 seconds)

Figura 10: Ejecuciones del programa.

Resulta importante destacar que en la programación paralela, las salidas generadas por un programa no siguen un orden secuencial estricto. Por tanto, diferentes ejecuciones darán como resultado diferentes órdenes en la salida; hecho observable en [30]. El uso adecuado del semáforo garantiza que ambos hilos podrán acceder al recurso de forma ordenada sin interferir en las operaciones del otro y provocar errores de ejecución.

### 3.2.2. SemaphoreTest

La segunda clase contenida en el proyecto contiene como atributo un objeto de tipo *Semaphore* inicializado con un total de 4 permisos y, además, otra clase en su interior: una clase estática denominada *PruebaHilos* y dos métodos, siendo el primero una sobrescritura del método *run* (considerando que *PruebaHilos* hereda de la clase *Thread*) y un método principal. Esta clase tiene únicamente como atributo el nombre del hilo en cuestión. El constructor de la clase *PruebaHilos* únicamente asigna un nombre al hilo creado en la propiedad mencionada anteriormente.

- La sobrescritura del método *run* inicia con un bloque *try*, donde cada hilo que invoque a este método tratará de obtener un permiso para realizar una operación, además, se imprimirá un mensaje en pantalla indicando la cantidad de permisos disponibles. En caso de que un permiso le sea concedido al hilo por medio de **acquire**, se imprimirá otro mensaje confirmando la obtención del permiso y, a continuación, se intentará ejecutar un ciclo *for* que indicará que el hilo se encuentra realizando una operación *i* (de un total de 5) y seguirá imprimiendo un mensaje del estado actual de los permisos. Posteriormente, el hilo se enviará a dormir 1 segundo.

```
for (int i = 1; i <= 5; i++) {  
  
    System.out.println(name + " : se encuentra realizando la operacion " + i  
        + ", permisos disponibles en el semaforo : "  
        + semaphore.availablePermits());  
  
    // Envía el hilo a dormir un segundo.  
    Thread.sleep(1000);  
}
```

Figura 11: Operación del hilo con un permiso adquirido.

Una vez que esta operación haya sido completada con éxito, se ejecuta un bloque **finally** que libera el permiso adquirido por el hilo con el método **release** e imprime una vez más el número de permisos disponibles en el semáforo.

- En el método principal se imprime un mensaje indicando el número total de permisos que el semáforo tiene disponibles, utilizando el método **availablePermits** para obtener tal información. Posteriormente, se crean 6 hilos de tipo *PruebaHilos* y se utiliza el método **start** sobre cada uno de ellos.

```
/*Seis hilos realizan operaciones con un total de 4 permisos.*/

PruebaHilos t1 = new PruebaHilos("A");
t1.start();

PruebaHilos t2 = new PruebaHilos("B");
t2.start();

PruebaHilos t3 = new PruebaHilos("C");
t3.start();

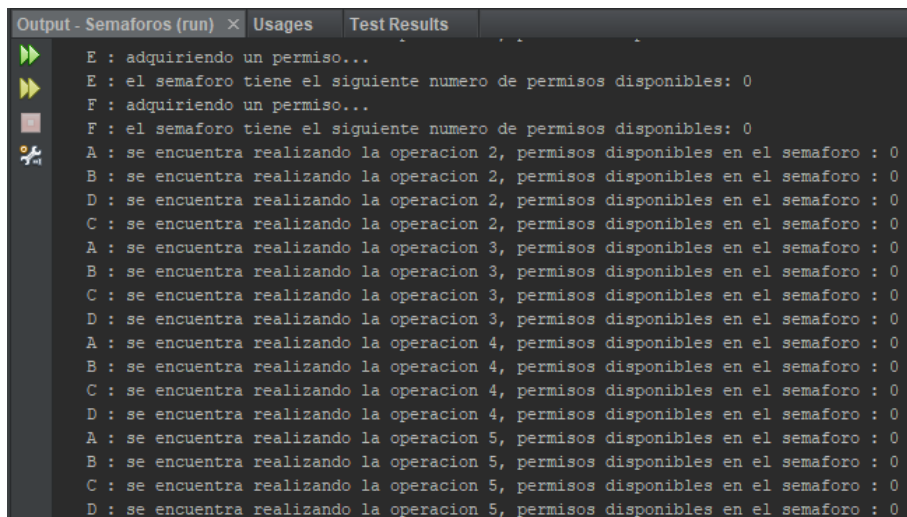
PruebaHilos t4 = new PruebaHilos("D");
t4.start();

PruebaHilos t5 = new PruebaHilos("E");
t5.start();

PruebaHilos t6 = new PruebaHilos("F");
t6.start();
```

Figura 12: *Hilos creados.*

Al realizar la ejecución del programa puede notarse que, en primer lugar, los hilos A, B, C y D son quienes obtienen los 4 permisos disponibles del semáforo; dejando afuera a los hilos E y F. Una vez que todos los hilos solicitaron un permiso, los 4 hilos con el permiso comienzan a realizar sus respectivas 5 operaciones. Cada hilo se envía a dormir tras haber realizado una iteración.

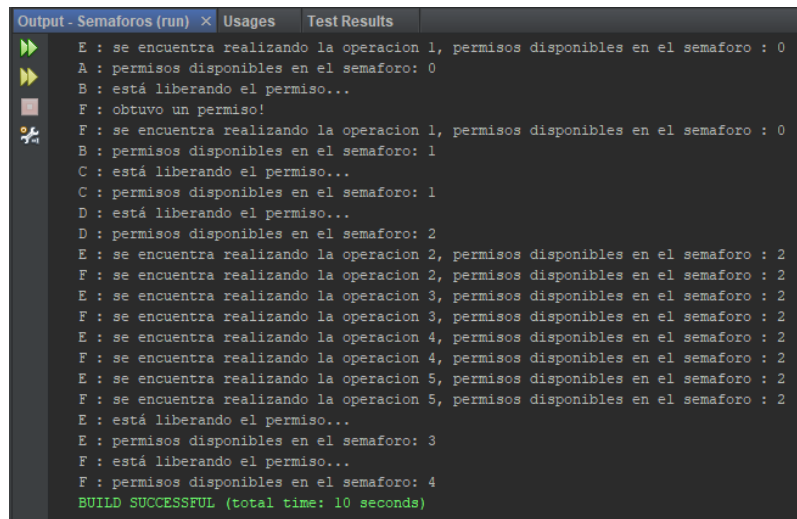


```
Output - Semaforos (run) x Usages Test Results
E : adquiriendo un permiso...
E : el semaforo tiene el siguiente numero de permisos disponibles: 0
F : adquiriendo un permiso...
F : el semaforo tiene el siguiente numero de permisos disponibles: 0
A : se encuentra realizando la operacion 2, permisos disponibles en el semaforo : 0
B : se encuentra realizando la operacion 2, permisos disponibles en el semaforo : 0
D : se encuentra realizando la operacion 2, permisos disponibles en el semaforo : 0
C : se encuentra realizando la operacion 2, permisos disponibles en el semaforo : 0
A : se encuentra realizando la operacion 3, permisos disponibles en el semaforo : 0
B : se encuentra realizando la operacion 3, permisos disponibles en el semaforo : 0
C : se encuentra realizando la operacion 3, permisos disponibles en el semaforo : 0
D : se encuentra realizando la operacion 3, permisos disponibles en el semaforo : 0
A : se encuentra realizando la operacion 4, permisos disponibles en el semaforo : 0
B : se encuentra realizando la operacion 4, permisos disponibles en el semaforo : 0
C : se encuentra realizando la operacion 4, permisos disponibles en el semaforo : 0
D : se encuentra realizando la operacion 4, permisos disponibles en el semaforo : 0
A : se encuentra realizando la operacion 5, permisos disponibles en el semaforo : 0
B : se encuentra realizando la operacion 5, permisos disponibles en el semaforo : 0
C : se encuentra realizando la operacion 5, permisos disponibles en el semaforo : 0
D : se encuentra realizando la operacion 5, permisos disponibles en el semaforo : 0
```

Figura 13: *Hilos creados.*

Una vez que los 4 hilos terminaron sus iteraciones, liberan sus permisos y los hilos E y F tratarán nuevamente de conseguir un permiso, el cual les será concedido al liberarse los anteriores. Realizarán las mismas 5 operaciones que los hilos anteriores y finalmente liberarán los 2 permisos que ocuparon. Al terminar la ejecución del programa, el hilo F indicará que existen 4 permisos disponibles, es decir, los que

fueron establecidos al inicio del programa.



```

Output - Semaforos (run) X Usages Test Results
>> E : se encuentra realizando la operacion 1, permisos disponibles en el semaforo : 0
>> A : permisos disponibles en el semaforo: 0
>> B : está liberando el permiso...
>> F : obtuvo un permiso!
>> F : se encuentra realizando la operacion 1, permisos disponibles en el semaforo : 0
>> B : permisos disponibles en el semaforo: 1
>> C : está liberando el permiso...
>> C : permisos disponibles en el semaforo: 1
>> D : está liberando el permiso...
>> D : permisos disponibles en el semaforo: 2
>> E : se encuentra realizando la operacion 2, permisos disponibles en el semaforo : 2
>> F : se encuentra realizando la operacion 2, permisos disponibles en el semaforo : 2
>> E : se encuentra realizando la operacion 3, permisos disponibles en el semaforo : 2
>> F : se encuentra realizando la operacion 3, permisos disponibles en el semaforo : 2
>> E : se encuentra realizando la operacion 4, permisos disponibles en el semaforo : 2
>> F : se encuentra realizando la operacion 4, permisos disponibles en el semaforo : 2
>> E : se encuentra realizando la operacion 5, permisos disponibles en el semaforo : 2
>> F : se encuentra realizando la operacion 5, permisos disponibles en el semaforo : 2
>> E : está liberando el permiso...
>> E : permisos disponibles en el semaforo: 3
>> F : está liberando el permiso...
>> F : permisos disponibles en el semaforo: 4
BUILD SUCCESSFUL (total time: 10 seconds)

```

Figura 14: Hilos E y F en ejecución.

Resulta particular notar que, en este caso, a los hilos no se les 'asignó' directamente el semáforo al momento de crearlos. En cambio, se estableció su uso al declararlo como un atributo de clase y definir otra clase estática dentro de la clase original.

En ambos ejemplos el concepto básico del semáforo se aplica principalmente en la capacidad de los objetos *Semaphore* de arbitrar el acceso a un recurso o una determinada operación en un programa. Además, el concepto se ve enriquecido por el hecho de que, además de conceder o negar el permiso de acceso, pueden establecerse que una determinada cantidad adicional de hilos puedan acceder de forma controlada al recurso establecido. Por tanto, resulta notable su capacidad de resolver el problema de las *secciones críticas* al arbitrar recursos entre varios elementos; evitando conflictos derivados de una pobre administración de los hilos.

## 4. Introducción a OpenMP

### 4.1. Actividad 1

La primer actividad consiste en teclear el programa secuencial proporcionado en la práctica y ejecutarlo. A cotinuación, se escribirá un versión paralela del mismo.

La salida obtenida es la siguiente:

```
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Adios

Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

Figura 15: Salida del programa proporcionado en forma secuencial.

```
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Adios

Process returned 0 (0x0)   execution time : 0.122 s
Press any key to continue.
```

Figura 16: Salida del programa proporcionado en forma paralela.

- **¿Qué diferencia hay en la salida del programa con respecto a la secuencial?:** La versión secuencial del código se ejecutó únicamente una sola vez. En cambio, en la versión paralela el código fue ejecutado un total de **4 veces**. Además, la ejecución no se realizó en un orden *secuencial* a partir del tercer *Hola Mundo* impreso en pantalla.
- **¿Por qué se obtiene esa salida?:** Al código proporcionado le fue añadido el constructor básico de OpenMP: **parallel**. A través de este constructor es posible generar una *región paralela*, creando un cierto número de hilos que ejecutarán instrucciones y que, cuando terminen su actividad, únicamente se tenga un hilo *maestro*; atendiendo al modelo fork-join mostrado en [5].

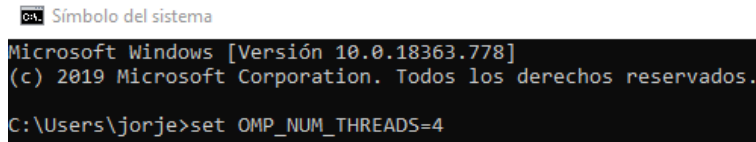
```
#pragma omp parallel
{
    //Bloque de código
}
```

Figura 17: Sintaxis de creación de una región paralela.

## 4.2. Actividad 2

Al momento de crear una región paralela, el número de hilos por defecto generados corresponde al número de núcleos que tenga el procesador en uso. El objetivo en esta práctica es modificar el número de hilos a un valor  $n$  establecido. Se realiza el siguiente procedimiento:

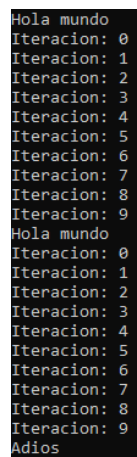
1. Se modifica la variable de ambiente `OMP_NUM_THREADS` desde la consola.



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.18363.778]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.
C:\Users\jorje>set OMP_NUM_THREADS=4
```

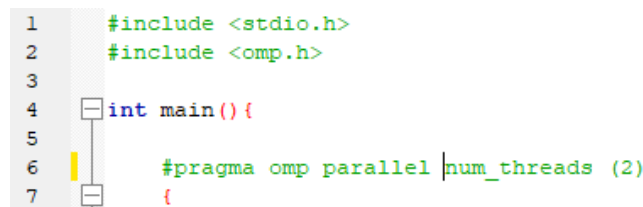
Figura 18: *Modificación realizada.*

2. Se modifica el número de hilos a  $n$ , un entero que llama a la función `omp_set_num_threads(n)` que se encuentra en la biblioteca `omp.h`.
3. Se agrega la cláusula `num_threads(n)` seguida del constructor `parallel`, como puede verse a continuación con  $n=2$ :



```
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Adios
```

Figura 19: *Salida obtenido con la modificación del número de hilos con  $n=2$ .*



```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main() {
5
6      #pragma omp parallel num_threads (2)
7      {
```

Figura 20: *Cláusula añadida.*

- **¿Qué sucedió en la ejecución con respecto al de la actividad 1?:** La computadora en la que se realizan los ejemplos es un i3 2100 con 2 núcleos y 4 hilos de ejecución, por lo que el valor por defecto para el número de hilos será de 4; como pudo observarse en la Actividad 1. Al modificar este valor por medio de **pragma** a  $n = 2$ , el código definido en la región paralela únicamente se ejecuta un total de 2 veces.

### 4.3. Actividad 3

En la programación paralela puede darse una situación denominada *race condition*, la cual sucede cuando diversos hilos tienen acceso sin control a recursos compartidos de memoria; tratando de operar con una misma dirección de memoria y escribir sobre la misma localidad al mismo tiempo. Tal situación puede provocar **salidas incorrectas o impredecibles**.

**OpenMP** permite definir qué partes de la memoria (por tanto, variables) se comparten o no entre los hilos. Dentro del código, cualquier variable que esté declarada fuera de la región paralela será compartida y cualquier variable declarada al interior de la región paralela será de carácter privado.

A partir del código trabajado, se saca de la región paralela la declaración de la variable entera  $i$ , se compila y se ejecuta el programa, con los siguientes resultados:

```

Iteracion: 0   Iteracion: 0 Hola mundo
Hola mundo   Hola mundo   Iteracion: 0
Iteracion: 0   Iteracion: 0 Iteracion: 1
Iteracion: 1   Iteracion: 5 Hola mundo
Iteracion: 2   Iteracion: 2 Iteracion: 0
Iteracion: 3   Iteracion: 1 Hola mundo
Iteracion: 4   Iteracion: 4 Hola mundo
Iteracion: 5   Iteracion: 5 Iteracion: 2
Iteracion: 6   Iteracion: 6 Iteracion: 1
Iteracion: 7   Iteracion: 7 Iteracion: 2
Iteracion: 8   Iteracion: 8 Iteracion: 3
Iteracion: 9   Iteracion: 9 Iteracion: 4
Hola mundo   Iteracion: 1 Iteracion: 5
Iteracion: 0   Hola mundo   Iteracion: 6
Iteracion: 1   Iteracion: 0 Iteracion: 7
Iteracion: 2   Iteracion: 1 Iteracion: 8
Iteracion: 3   Iteracion: 2 Iteracion: 9
Iteracion: 4   Iteracion: 3 Iteracion: 1
Iteracion: 5   Iteracion: 4 Iteracion: 1
Iteracion: 6   Iteracion: 5 Iteracion: 2
Iteracion: 7   Iteracion: 6 Iteracion: 3
Iteracion: 8   Iteracion: 7 Iteracion: 4
Iteracion: 9   Iteracion: 3 Iteracion: 5
Iteracion: 1   Iteracion: 9 Iteracion: 7
Iteracion: 1   Iteracion: 8 Iteracion: 8
Adios        Adios        Iteracion: 9
                Iteracion: 0
Process returned 0 (0x0)   Iteracion: 0
Press any key to continue. Iteracion: 6
                        Adios

```

Figura 21: Salidas obtenidas.

- **¿Qué sucedió? Y ¿Por qué?:** Las salidas obtenidas en las actividades anteriores, a pesar de no contar con un orden en su ejecución, esta fue consistente. En cambio, las salidas obtenidas con la modificación realizada resultaron en salidas muy diferentes entre sí; resultado de una *condición de carrera* al establecer a  $i$  como una variable pública.

## 4.4. Actividad 4

Por medio de 2 cláusulas puede forzarse a que una variable privada sea compartida y viceversa:

- **shared()**: Por medio de esta cláusula las variables colocadas separadas por coma dentro del paréntesis serán compartidas entre todos los hilos de la región paralela. Únicamente existe una copia, y todos los hilos pueden acceder y modificar tal copia.
- **private()**: Las variables colocadas separadas por coma dentro del paréntesis serán privadas, es decir, se crean  $p$  copias para cada hilo, las cuales no se inicializan y no tienen valor definido al final de la región paralela ya que sea destruyen al terminar la ejecución de los hilos.

Al código de la Actividad 3 se le ha añadido la cláusula **private** después del constructor **parallel** y se coloca la variable  $i$  en su interior. La salida obtenida es la siguiente:

```
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Hola mundo
Iteracion: 0
Iteracion: 1
Iteracion: 2
Iteracion: 3
Iteracion: 4
Iteracion: 5
Iteracion: 6
Iteracion: 7
Iteracion: 8
Iteracion: 9
Adios
Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.
```

Figura 22: Salida obtenida.



- **¿Qué sucedió?:** Al forzar que la variable *i* sea privada, se obtuvo una ejecución ordenada del código un total de cuatro veces. Es interesante destacar que la salida resultante es similar a una ejecución secuencial realizada un total de 4 veces. De esta forma, la ejecución resulta limpia y sin salidas inesperadas resultado de una *race condition*.

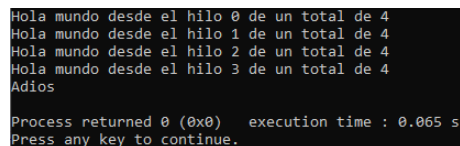
## 4.5. Actividad 5

Dentro de una región paralela cada hilo generado tiene asignado un identificador; tales datos pueden conocerse en tiempo de ejecución llamando a las funciones **omp\_get\_num\_threads()** y **omp\_get\_thread\_num()**, respectivamente.

En el ejemplo proporcionado se declara una variable denominada *tid* utilizada para almacenar el identificador de un determinado hilo. Para que este funcione adecuadamente, *tid* debe de ser **privada** dentro de la región paralela, ya que de lo contrario, se producirá una *race condition*; lo cual implica que todos los hilos escribirán sobre esa dirección de memoria sin control, 'compitiendo' para ver quién puede operar antes. Por consecuencia, el resultado obtenido puede resultar inesperado o inconsistente.

```
int tid, nth;
#pragma omp parallel private(tid)
{
    tid = omp_get_thread_num();
    nth = omp_get_num_threads();
    printf("Hola mundo desde el hilo %d de un total de %d\n", tid, nth);
}
```

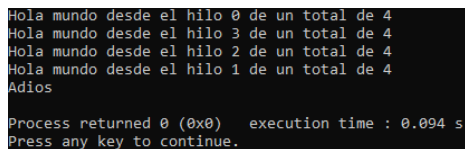
Figura 23: Variable *tid* identificadora.



```
Hola mundo desde el hilo 0 de un total de 4
Hola mundo desde el hilo 1 de un total de 4
Hola mundo desde el hilo 2 de un total de 4
Hola mundo desde el hilo 3 de un total de 4
Adios
Process returned 0 (0x0)   execution time : 0.065 s
Press any key to continue.
```

Figura 24: Ejecución del ejemplo.

A continuación, se elimina la cláusula **private** para observar el comportamiento del programa:



```
Hola mundo desde el hilo 0 de un total de 4
Hola mundo desde el hilo 3 de un total de 4
Hola mundo desde el hilo 2 de un total de 4
Hola mundo desde el hilo 1 de un total de 4
Adios
Process returned 0 (0x0)   execution time : 0.094 s
Press any key to continue.
```

Figura 25: Ejecución del ejemplo.

- **¿Qué sucedió?:** Ocurrió una situación inesperada. Al ejecutar el ejemplo, aunque en un inicio se obtuvieron salidas ordenadas, tras ejecutar varias ocasiones el ejemplo se obtuvieron salidas desordenadas pese a que la variable *tid* se estableció como privada. El comportamiento fue el mismo al

eliminar la cláusula **private**. En general, esta situación ocurre cuando los hilos se encuentran compitiendo para realizar una operación sobre un recurso compartido. Se desconoce por qué el programa presentó tal comportamiento al ejecutarlo a pesar de tener la cláusula correcta.

## 4.6. Actividad 6

En la siguiente actividad propuesta se requiere realizar la suma de dos arreglos unidimensionales de 10 elementos de forma paralela haciendo uso de únicamente dos hilos. Para ello, se hará uso de un paralelismo de datos o descomposición de dominio, es decir, cada hilo trabajará con diferentes elementos de los arreglos A y B a sumar, pero ambos ocuparán el mismo algoritmo para realizar la suma.

A partir de la versión serial, y asumiendo que A y B tiene valores para ser sumados, se obtiene la siguiente salida:

```
1      7      4      0      9      4      8      8      2      4
5      5      1      7      1      1      5      2      7      6
6     12      5      7     10      5     13     10      9     10
Process returned 3 (0x3)   execution time : 0.079 s
Press any key to continue.
```

Figura 26: Ejecución del ejemplo en su versión serial.

En la versión paralela de este algoritmo, el hilo 0 sumará la primera mitad de A con la primera de B y el hilo 1 sumará la segunda mitad de A con la segunda de B. Cada hilo realiza las mismas instrucciones pero utilizando índices diferentes para referirse y operar con diferentes elementos de los arreglos. Implementando el programa con las instrucciones brindadas, se obtiene una versión paralela del algoritmo que produce la siguiente salida:

```
1      7      4      0      9      4      8      8      2      4
5      5      1      7      1      1      5      2      7      6
hilo 0 calculo C[0]= 6
hilo 0 calculo C[1]= 12
hilo 0 calculo C[2]= 5
hilo 0 calculo C[3]= 7
hilo 1 calculo C[5]= 5
hilo 1 calculo C[6]= 13
hilo 1 calculo C[7]= 10
hilo 1 calculo C[8]= 9
Process returned 0 (0x0)   execution time : 0.307 s
Press any key to continue.
```

Figura 27: Ejecución del ejemplo en su versión paralela.

Lo anterior ilustra la posibilidad de *paralelizar* un algoritmo a partir de su versión serial. La posibilidad de que un algoritmo sea candidato de este proceso depende principalmente de que los ciclos puedan ser ejecutados por diversos hilos. La viabilidad de este proceso dependerá principalmente de las ganancias obtenidas en su funcionamiento paralelo. En el caso particular de este ejemplo, las ganancias no fueron apreciables, pues al correr ambos ejemplos en varias ocasiones podía suceder que la versión serial se ejecutara en menor tiempo o, al contrario, que la versión paralela se ejecutara en menor tiempo.

## 4.7. Actividad 7

Otro de los constructores de OpenMP es el **for**, el cual divide las iteraciones de una estructura repetitiva **for**. Su uso requiere que se encuentre dentro de una *región paralela*.

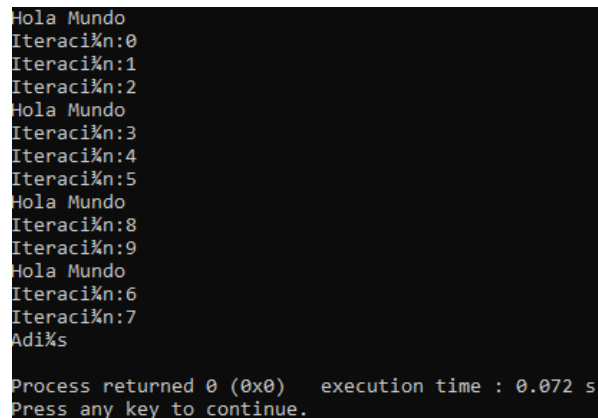
```
#pragma omp parallel
{
    ...
    #pragma omp for
    for(i=0;i<12;i++) {
        Realizar Trabajo();
    }
    ...
}
```

Figura 28: *Sintáxis del constructor for.*

De esta manera, la variable de control *i* será privada automáticamente, es decir, cada hilo trabajará con una copia de *i*.

El constructor descrito anteriormente se utiliza principalmente en el paralelismo de dato, cuando es detectable que una sección de un algoritmo puede trabajarse con varios hilos, pero sobre diferentes datos y cuando no existan *dependencias de datos* con las iteraciones, es decir, que otras iteraciones no dependan de los resultados obtenidos por medio de anteriores iteraciones. Por tal situación, no siempre resulta conveniente dividir las iteraciones de un ciclo **for**.

A partir del código de la Actividad 1, se han dividido las iteraciones del ciclo **for** utilizando el constructor **for**. Lo anterior ha dado como resultado la siguiente salida:



```
Hola Mundo
Iteraci% n:0
Iteraci% n:1
Iteraci% n:2
Hola Mundo
Iteraci% n:3
Iteraci% n:4
Iteraci% n:5
Hola Mundo
Iteraci% n:8
Iteraci% n:9
Hola Mundo
Iteraci% n:6
Iteraci% n:7
Adi% s

Process returned 0 (0x0)   execution time : 0.072 s
Press any key to continue.
```

Figura 29: *Sintáxis del constructor for.*

En este caso, la división del ciclo for provocó que, a partir de los últimos *Hola mundo*, se viera ligeramente alterado. Pese a ello, su ejecución fue relativamente correcta y no tuvo resultados inesperados.

## 4.8. Actividad 8

Retomando la *Actividad 6*, los hilos realizan las mismas operaciones de sum, pero sobre diferentes elementos del arreglo, lo cual se consigue al hacer que cada hilo inicie y termine sus iteraciones en valores diferentes con el fin de referirse a los diferentes elementos presentes en  $A_{ij}$  y  $B_{ij}$ . Por medio del constructor **for** puede realizarse tal operación al dividirse las iteraciones de cada hilo que trabajan con distintos valores para el índice de control.

La solución proporcionada fue implementada en el código de la *Actividad 6* y la salida obtenida es la siguiente:

```
1      7      4      0      9      4      8      8      2      4
5      5      1      7      1      1      5      2      7      6
hilo 1 calculo C[3]= 7
hilo 1 calculo C[4]= 10
hilo 1 calculo C[5]= 5
hilo 3 calculo C[8]= 9
hilo 3 calculo C[9]= 10
hilo 0 calculo C[0]= 6
hilo 0 calculo C[1]= 12
hilo 0 calculo C[2]= 5
hilo 2 calculo C[6]= 13
hilo 2 calculo C[7]= 10

Process returned 0 (0x0)   execution time : 0.104 s
Press any key to continue.
```

Figura 30: Salida obtenida con la solución implementada.

## 5. Conclusiones

A través del trabajo desarrollado pude observar por primera la creación de código paralelo por medio de la API de **OpenMP**, marcando un regreso al uso del lenguaje C en el curso. A pesar de que los ejemplos tenían como objetivo principal mostrar aspectos muy básicos de su uso, considero que son muy enriquecedores ya que brindan los cimientos más básicos para comenzar a escribir versiones paralelas de otros algoritmos de origen serial.

Es importante destacar que las ejecuciones varían unas con otras debido a la gestión de los procesos por el sistema operativo. Por esto, resulta especialmente importante realizar una adecuada gestión del flujo de lectura/escritura que los hilos realizan en su ejecución; con el fin de evitar una posible *sección crítica* o una *condición de carrera* que provoque conflictos entre ellos. Las implementaciones del *semáforo* resultaron exitosas en ejemplificar vías de resolver tales problemas durante procesos recurrentes.

El único ejemplo que me provocó dudas fue el proporcionado en la *Actividad 5*, ya que en las salidas obtenidas no pude apreciar un cambio notable con respecto a la versión serial; puede que haya cometido



un error al escribir el ejemplo (aunque se verifíco que no fuese así) o que el ejemplo en sí contenga algún detalle que haya pasado por alto. Espero que tal situación pueda ser aclarada debidamente.

Finalmente, y fuera de ese pequeño contratiempo, el mayor aprendizaje de esta práctica, más allá de las cláusulas básicas de OpenMP o el funcionamiento del semáforo, es la demostración del uso adecuado de la gestión de recursos compartidos en memoria; y las posibles consecuencias que puede traer el descuido de este aspecto en un programa paralelo. El seguimiento de tales pautas traerá consigo programas paralelos que superen en rendimiento a sus versiones seriales.

## Referencias

- [1] Introduction to critical section with animation. Recuperado de: <https://tuxthink.blogspot.com/2013/07/introduction-to-critical-section-with.html?m=1>. Fecha de consulta: 13/05/2020.
- [2] Java Semaphore. Recuperado de: <https://data-flair.training/blogs/java-semaphore/>. Fecha de consulta: 13/05/2020.
- [3] Java.util.concurrent.Semaphore class in Java. Recuperado de: <https://www.geeksforgeeks.org/java-util-concurrent-semaphore-class-java/>. Fecha de consulta: 13/05/2020.
- [4] Programación Concurrente. Recuperado de: <https://www.fing.edu.uy/tecnoinf/mvd/cursos/so/material/teo/so07-concurrencia.pdf>. Fecha de consulta: 13/05/2020.
- [5] Semaphore in Java. Recuperado de: <https://www.geeksforgeeks.org/semaphore-in-java/>. Fecha de consulta: 13/05/2020.
- [6] Semaphores in Process Synchronization. Recuperado de: <https://www.geeksforgeeks.org/semaphores-in-process-synchronization/>. Fecha de consulta: 13/05/2020.
- [7] Sincronización basada en memoria compartida: Semáforos. Recuperado de: [https://www.ctr.unican.es/asignaturas/procodis\\_3\\_II/Doc/Procodis\\_2\\_03.pdf](https://www.ctr.unican.es/asignaturas/procodis_3_II/Doc/Procodis_2_03.pdf). Fecha de consulta: 13/05/2020.
- [8] Sincronización de hilos. Recuperado de: <https://eaddfsi.wordpress.com/2009/06/13/sincronizacion-de-hilos/>. Fecha de consulta: 13/05/2020.

Los créditos de las fotografías pertenecen a sus respectivos autores. ©