# Secretos de la Programación Concurrente

Stuart Perez, Luis Castro

Autómatas y Compiladores, Escuela de Ciencias de la Computación e Informática,
San Pedro de Montes de Oca, Costa Rica
jstuartp@gmail.com ; luiscastrob@gmail.com

**Abstract.** Este articulo refiere a las practicas comúnmente utilizadas para la construcción de programas con procesamiento concurrente, esto es, utilización de varios núcleos para el procesamiento de los datos.

**Keywords:** Programación, multiprocesador, hilos, sincronización

## 1 Introducción

*"Divide et vinces"* La frase celebre del Emperador Julio Cesar, que significa, "Divide y Vencerás" se yergue como una máxima para el mundo de la programación de nuestros días. la multiprogramación, una técnica para entrelazar la ejecución de varios programas, se ha convertido en la solución ideal para reducir los tiempos de ejecución, así como para aprovechar al máximo las cualidades de los procesadores multinucleo de la nueva era.

Esta técnica nació ya hace varios años y tiene sus principales raíces en la construcción de sistemas operativos, con ella, vieron la luz términos como procesos e hilos, sincronización, tiempo compartido de procesador y muchos otros. El presente articulo esta orientado a repasar los principales conceptos de la multiprogramación, así como varias de sus técnicas y "uno que otro" secreto, que nos ayuden a conocer mas este interesante mundo, que se encuentra, hoy en día, en su máximo apogeo.

## 2 Técnicas básicas de programación concurrente

### 2.1 EXCLUSIÓN MUTUA

Una forma en la cual podemos lograr que 2 o más procesos de un programa concurrente se comuniquen es usando variables comunes. Este método aunque sencillo nos puede traer problemas. Se pueden presentar errores en el resultado y ejecución del programa ya que el acceso concurrente podría desembocar en que la acción de un proceso sobre la variable común interfiera en las acciones de otro de una forma no adecuada.

Veamos un ejemplo:
Tenemos 2 procesos (PA, PB) que comparten una variable(X). Los procesos pueden sumar o restar una unidad a la variable. Los procesos comparten la posición de memoria. Supongamos la siguiente sucesión de eventos.
- P1 lee la variable x, la coloca en su registro.
- P2 lee la variable x, la coloca en su registro.
- P2 suma una unidad al registro de x
- P1 suma una unidad al registro de x
- P1 almacena el valor de su registro en la dirección de memoria de x
- P2 almacena el valor de su registro en la dirección de memoria de x

Se ve claramente que en esa sucesión de acciones se pierde un incremento de X. este tipo de errores son muy difíciles de encontrar en un programa pues dependen de temporizaciones especificas. Por esto es necesario sincronizar estos procesos para evitar este tipo de errores.
Lo logramos identificando las regiones de los procesos que comparten variables con otros procesos y ejecutarlas como si fueran una única instrucción. Se denomina Región Crítica a aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma concurrente o, también, que desde otro proceso se ven como si fueran una única instrucción. Esto quiere decir que si un proceso entra a ejecutar una sección crítica en la que se accede a unas variables compartidas, entonces otro proceso no puede entrar a ejecutar una región crítica en la que acceda a variables compartidas con el anterior.
Para lograr esto se deben implementar protocolos de software que no permitan el acceso a una región crítica cuando está siendo utilizada por un proceso.

## 2.2 SEMÁFOROS

Dijkstra creó una solución al problema de la exclusión mutua con el concepto de semáforo binario. Los semáforos permiten resolver los problemas de sincronización entre procesos concurrentes y son muy importantes en el diseño de sistemas operativos.
Un semáforo binario es un indicador (S) de condición que indica si un recurso puede ser usado o no. Un semáforo binario trabaja de forma muy simple, tiene 2 estados: 0 y 1. Si S = 1 entonces el recurso está disponible, y el proceso lo puede tomar. Si S = 0 el recurso no está disponible y el proceso debe esperar. Además tienen tres operaciones: Inicializar (Init), Espera (wait), Señal (signal)
La exclusión mutua se resuelve fácilmente usando semáforos. La operación “Espera” se usa como procedimiento de bloqueo antes de entrar a ejecutar una región crítica y “Señal” como procedimiento de desbloqueo.

## 2.3 SINCRONIZACIÓN

Utilizando semáforos logramos la sincronización entre dos procesos. Las operaciones "Espera" y "Señal" no usan dentro de un solo proceso sino que se dan en dos procesos distintos, el que ejecuta la operación de "Espera" queda bloqueado hasta que el otro proceso ejecuta la operación de "Señal".

## 2.4 MONITORES

Definimos un monitor como un conjunto de procedimientos que proporcionan el acceso garantizando exclusión mutua a un recurso compartidos por un grupo de procesos. Los procedimientos van encapsulados dentro de un módulo que tiene la característica de que sólo un proceso va a estar activo para ejecutar un procedimiento del monitor.
Podemos ver a el monitor como una muralla perimetral del recurso, así que los procesos que van a utilizarlo deben entrar al perímetro del recurso, pero solo se puede según las reglas del monitor. Muchos procesos van a solicitar entrar pero sólo se permite que entre un proceso a la vez, los procesos deberán esperar a que salga del perímetro el proceso que se encuentra del otro lado de la muralla para que otro pueda entrar.
Si comparamos el uso de un monitor contra el uso de los semáforos: encontramos ventaja en los monitores pues la única acción que debe realizar el programador del proceso que quiere solicitar un recurso es llamar una entrada del monitor. Si el monitor se encuentra programado fielmente no podrá ser mal utilizado por un proceso que quiera acceder al recurso. Todo lo contrario ocurre con los semáforos pues el programador debe proveer la secuencia correcta de instrucciones "Espera" y "Señal".
El monitor deberá ser implementado usando manejo de prioridades, para evitar que un proceso en espera de usar el recurso se vea bloqueado indefinidamente por otros procesos.

## 2.5 MENSAJES

Los mensajes se pueden usar como una solución al problema de la concurrencia de procesos que brinda tanto sincronización como comunicación siendo útil para sistemas centralizados y distribuidos. Así es como hoy los vemos en los sistemas operativos soportando las comunicaciones del sistema de un computador o de varios.
Se presenta un proceso emisor, un receptor y la información que compone el mensaje.
Las operaciones básicas para mensajes son: "enviar" (mensaje) y" recibir" (mensaje).
Las acciones de transmisión de información y de sincronización se ven como actividades inseparables.
El proceso de denominación de las tareas para la comunicación por mensajes se puede realizar de dos formas distintas: directa e indirectamente.

Para la comunicación directa los dos procesos nombran de manera explícita al proceso con el que se están comunicando.
Las operaciones de enviar y recibir toman la forma:
Enviar (Q, mensaje): envía un mensaje al proceso Q
Recibir (P, mensaje): recibe un mensaje del proceso P
Este sistema brinda mucha seguridad dado que tanto el receptor como el emisor deben conocer con quien se van a comunicar pero a la vez esto se convierte en una visible desventaja.
En la comunicación indirecta los mensajes se envían y reciben a través de un intermediario que se llama normalmente buzón o puerto. Un buzón es un objeto en el que los procesos "depositan" mensajes y donde otros procesos los pueden recoger. Es menos seguro pero ofrece una mayor versatilidad que el nombramiento directo. Esto porque es posible la comunicación 1 a 1, 1 a N, N a 1, y N a M.
Cada buzón que exista tendrá una identificación para reconocerlo. Las operaciones básicas de comunicación serán así:
Enviar (buzónA, mensaje): envía un mensaje al buzón A
Recibir (buzónA, mensaje): recibe un mensaje del buzón A.
Las diferencias en los modelos usados para la sincronización de los procesos se deben a las distintas formas que puede tomar la operación de envío del mensaje.
a) Síncrona. El proceso que envía sólo prosigue su tarea cuando el mensaje ha sido recibido.
b) Asíncrona. El proceso que envía un mensaje sigue su ejecución sin preocuparse de si el mensaje se recibe o no.
c) Invocación remota. El proceso que envía el mensaje sólo prosigue su ejecución cuando ha recibido una respuesta del receptor.

### 2.6 INTERBLOQUEO

El interbloqueo es el error más serio que puede ocurrir en programas concurrentes, este consiste en que dos o más procesos entran en un estado que imposibilita a cualquiera de ellos salir del estado en que se encuentra. A dicha situación se llega porque cada proceso adquiere algún recurso necesario para su operación a la vez que espera a que se liberen otros recursos que retienen otros procesos, llegándose a una situación que hace imposible que ninguno de ellos pueda continuar.

## 2 Aplicaciones Prácticas

Aprender a programar de forma paralela no es un simple paseo por el bosque, sin embargo, como toda técnica de programación, es posible llegar a dominarla con el tiempo y sobre todo, con mucha dedicación.

Inicialmente el mundo de la programación no estaba diseñado para trabajar con la concurrencia; durante sus primeros años los esquemas de trabajo secuencial gozaron de mucho éxito, y pronto, todo el mundo de la programación continuo con esta

tendencia, depurando y perfeccionando el comportamiento de bloque. Distintos paradigmas se pusieron en práctica, bajo la constante idea de la programación secuencial.

Los primeros en dar el salto resultaron ser los sistemas operativos, los cuales, en su afán por mejorar los tiempos de respuesta y aprovechar los momentos de "ocio" en los procesadores, introdujeron técnicas de "multiprocesamiento", este procesamiento inicialmente era una simple simulación de paralelismo, ya que, al contar con un único procesador físico, los procesos o "hilos" debían echar mano al "tiempo compartido" para hacer uso del CPU.

Con el advenimiento de los tiempos modernos, el peligro de alcanzar el límite cuántico, y las limitaciones en los materiales utilizados para la fabricación de procesadores, nace la tendencia de los múltiples núcleos, esto es, computadores con más de dos procesadores, los cuales comparten chaches y memoria principal. Esta nueva concepción dio un una nueva luz de esperanza a la programación en paralelo, ya que resultaba posible (y útil) programar hilos del sistema que tomaran diferentes núcleos para su ejecución.

Pero con estas nuevas aplicaciones, llegaron nuevos retos, la multiprogramación se encontró con los conflictos de concurrencia y con la necesidad de asegurar la integridad de los datos, nuevas técnicas de programación debieron ser concebidas, y hasta la actualidad, aun se trabaja en mejorar los métodos de la programación concurrente.

Dicho esto, echemos un vistazo breve a las técnicas más populares aplicadas en la multiprogramación.

Supongamos que en su primer día de trabajo, su nuevo jefe le solicita construir un programa que le devuelva todos los números primos entre 1 y $10^{10}$ (no ahondemos en el porqué), usando una maquina con procesamiento paralelo y diez hilos de capacidad. Esta computadora está siendo alquilada al minuto, y entre más se tarde su algoritmo en funcionar, más costoso resultara para la empresa. ¿Cómo lo resolvería?

En un principio, es fácil pensar en dividir el trabajo en partes iguales, darle a cada hilo la misma cantidad de entrada, este intento fallaría, ya que los números primos no aparecen de manera uniforme, por ejemplo, hay muchos números primos entre 1 y $10^9$ pero muy pocos entre $9.10^9$ y $10^{10}$, además, el tiempo de procesamiento de cada primo no es el mismo, se tarda más analizando un numero grande que un número pequeño, en otras palabras, no hay motivo para pensar que el trabajo puede ser dividido en partes iguales y tampoco resulta claro cuánto trabajo deberá tener cada hilo.

Una manera más prometedora de realizar el trabajo, es asignar un entero a cada hilo a la vez, para esto, resulta necesario utilizar una variable contadora independiente del código de cada hilo.

Sin embargo, el uso de esta variable nos puede crear un conflicto mayor e inesperado, supongamos que el hilo A acceda el segmento de código, se le asigna el contador 1 y comienza su ejecución, mientras que en el mismo instante, el hilo B tiene acceso al mismo contador y registra un 1 como su número de inicio. O peor aún, supongamos

que el hilo A lee el 1 y antes de aumentar el contador a 2, otro hilo C ha realizado varias iteraciones colocando el contador en 4; cuando el hilo A realiza la escritura, la variable contador estaría siendo disminuida en dos unidades. La raíz de todo este problema resulta ser que, sobre la variable contador es necesario hacer dos distintas operaciones, lectura y escritura, resolver esto consta de tomar una simple decisión, quien va primero y quien va después.

En los aspectos prácticos de la programación concurrente, existen muchos acercamientos, muchas teorías y muchas diferentes maneras de hacer las cosas (como vimos en los párrafos anteriores) sin embargo, pocas nos llevan a un resultado optimo.

Los diferentes lenguajes de programación han adoptado, con el paso del tiempo, estrategias para el manejo de concurrencia, es así como C++ establece las primeras bibliotecas para el uso de hilos. En este lenguaje, el programador puede desencadenar la creación de procesos hijos de un proceso padre, con la instrucción fork, estos hilos son completamente dependientes de su padre y terminan su proceso cuando este así se los indique. Usualmente, se escribe un segmento de código que será compartido por los diferentes procesos hijos, dentro del mismo, las variables serán comunes a todos y cada uno de ellos tendrá una copia de ellas. Los mecanismos de sincronización utilizados son los mismos observados en la primera parte de este documento, esto con el fin de evitar las posibles colisiones y demás conflictos antes mencionados.

```
1 class Counter {
2     private int value;
3     public Counter(int c) { // constructor
4            value = c;
5           }
6     // increment and return prior value
7     public int getAndIncrement() {
8           int temp = value; // start of danger zone
9           value = temp + 1; // end of danger zone
10          return temp;
11    }
12 }
```

**Figura 1**

En el ejemplo anterior, vemos la declaración de la clase Counter (en java), esta clase seria funcional para la utilización de un solo proceso, sin embargo, al utilizar más de dos hilos sobre este segmento de código, esta implementación se vuelve inestable, ya que ambos procesos podrían acceder indiscriminadamente a la sección marcada con los comentarios `// start of danger zone.`
En el siguiente segmento de código, se utilizará una implementación que soluciona este problema

```
1 public interface Lock {
2     public void lock(); // before entering critical
3                            section
```

```
4     public void unlock(); // before leaving critical
5                                section
6 }
1 public class Counter {
2     private long value;
3     private Lock lock; // to protect critical section
4
5     public long getAndIncrement() {
6           lock.lock(); // enter critical section
7           try {
8                 long temp = value; // in critical section
9                 value = temp + 1; // in critical section
10              } finally {
11                lock.unlock(); // leave critical section
12                       }
13    return temp;
14    }
15 }
```

**Figura 2**

El uso de los métodos lock y unlock, eliminan el problema sincronización presentado en el segmento anterior, dentro de las regiones criticas, solo un hilo podrá trabajar a la vez, cuando este finaliza su ejecución, sale de la región critica y esta queda abierta, para que otro proceso haga uso de ella.

Sin embargo, para que este sistema tenga efecto, es necesario que cada uno de los hilos presentes tenga cierto formato, para ello, cada hilo debe obedecer a las siguientes reglas:

1. Cada sección critica debe estar asociada únicamente a un método lock()
2. El hilo debe invocar al método lock() cuando está tratando de ingresar a la zona critica y
3. El hilo debe llamar al método unlock() cuando abandone la zona critica.

Las regiones condicionales nos aseguran éxito en los tres requisitos de la sincronización, ya que aseguran la exclusión mutua, están libres de “deadlocks” y no causan inanición.

En las siguientes figuras, podremos apreciar un ejemplo del uso de semáforos para controlar el flujo de ejecución de 3 hilos.

Lo que se requiere es que el hilo 1 y el 3 siempre ejecuten primero que los hilos 2 y 4. Sin el uso de una estructura de control como los semáforos, asegurar este comportamiento resultaría imposible, ya que el procesador repartiría los tiempos de ejecución a los hilos de manera aleatoria.

```
1 import java.util.concurrent.Semaphore;
2   public class p1 extends Thread  {
3      protected Semaphore oFinP1;
4      public p1(Semaphore oFinP1) {
5            this.oFinP1 = oFinP1;
6      }
7      public void run()  {
8            try {
9                  sleep((int) Math.round(500 * Math.random() -
10                 0.5));
11                 }
12                 catch (InterruptedException e) {
13                       e.printStackTrace();
14                 }
15     System.out.println("P1");
16     this.oFinP1.release(2);
17     }
18 }
```

**Figura 3: Segmento de Código para el hilo p1**

Podemos apreciar en la figura 3 el uso de la variable oFinP1 de tipo Semaphore, que se establece con la propiedad protected, además, en la línea 16 vemos el llamado al método release(2), el cual permite liberar el semáforo una vez que la ejecución del hilo halla finalizado

```
1 import java.util.concurrent.Semaphore;
2 public class p2 extends Thread  {
3      protected Semaphore oFinP1;
4      protected Semaphore oFinP3;
5      public p2(Semaphore oFinP1,Semaphore oFinP3) {
6            this.oFinP3 = oFinP3;
7            this.oFinP1 = oFinP1;
8      }
9      public void run()  {
10           try {
11           this.oFinP1.acquire();
12           this.oFinP3.acquire();
13           } catch(Exception e) {
14                 e.printStackTrace();
15           }
16           try {
17                 sleep((int) Math.round(500 * Math.random() –
18                 0.5));
19           } catch (InterruptedException e) {
20                 e.printStackTrace();
21           }
22           System.out.println("P2");
23     }
24}
```

**Figura 4: Segmento de Código para el hilo p2**

Para el segmento de código correspondiente al hilo 2, resulta necesario implementar más variables de semáforo, ya que se debe controlar si tanto el proceso 1 como el proceso 3 ya fueron ejecutados, para así poder hacer uso del procesador. En el método run(), el programador hace llamados al método acquire() sobre los semáforos de los hilos 1 y 3, si este método tiene éxito, el proceso continuara su ejecución, sino, se detendrá a esperar que estos semáforos se marquen como libres.

```
1 import java.util.concurrent.Semaphore;
2 public class p3 extends Thread  {
3      protected Semaphore oFinP3;
4      public p3(Semaphore oFinP3) {
5            this.oFinP3 = oFinP3;
6      }
7      public void run()  {
8            try {
9                  sleep((int) Math.round(500 * Math.random() –
10                 0.5));
11           } catch (InterruptedException e) {
12                 e.printStackTrace();
13           }
14           System.out.println("P3");
15           this.oFinP3.release(2);
16     }
17}
```

**Figura 5: Segmento de Código para el hilo p3**

Los segmentos de código para el hilo 3 y 4 son semejantes a los respectivos 1 y 2.

```
1 import java.util.concurrent.Semaphore;
2 Public class p4 extends Thread  {
3      protected Semaphore oFinP1;
4      protected Semaphore oFinP3;
5      public p4(Semaphore oFinP1,Semaphore oFinP3) {
6            this.oFinP3 = oFinP3;
7            this.oFinP1 = oFinP1;
8      }
9      public void run()  {
10           try {
11                 this.oFinP1.acquire();
12                 this.oFinP3.acquire();
13                 } catch(Exception e) {
14                       e.printStackTrace();
15                 }
16           try {
17                 sleep((int) Math.round(500 * Math.random() –
18                 0.5));
19           } catch (InterruptedException e) {
20                 e.printStackTrace();
21           }
22           System.out.println("P4");
```

```
23     }
24}
```

**Figura 6: Segmento de Código para el hilo p4**

```
1 import java.util.concurrent.Semaphore;
2 public class UsoSemaforos {
3      protected static Semaphore oFinP1,oFinP3;
4      public static void main(String[] args) {
5              oFinP1 = new Semaphore(0,true);
6              oFinP3 = new Semaphore(0,true);
7              (new Thread(new p1(oFinP1))).start();
8              (new Thread(new p2(oFinP1,oFinP3))).start();
9              (new Thread(new p3(oFinP3))).start();
10             (new Thread(new p4(oFinP1,oFinP3))).start();
11     }
12}
```

**Figura 7: Segmento de Código que controla la ejecución de los hilos**

Para el caso de la figura 7, el método que inicia el proceso de todos los hilos pone en true los semáforos 1 y 3 y activa cada uno de los métodos, con los correspondientes semáforos y usando el método start().