

PROGRAMACIÓN DE SERVICIOS Y PROCESOS
TÉCNICO EN DESARROLLO DE APLICACIONES
MULTIPLATAFORMA

Sincronización de varios hilos

ÍNDICE

/ 1. Introducción y contextualización práctica	3
/ 2. Problemas asociados a la sincronización	4
/ 3. Herramientas para sincronización: monitores	4
/ 4. Caso práctico 1: “Sincronización en otros lenguajes de programación”	5
/ 5. Problema del productor consumidor: definición	6
/ 6. Problema del productor consumidor: monitores	6
/ 7. Semáforos	7
/ 8. Caso práctico 2: “¿Qué método de sincronización utilizar?”	8
/ 9. Problemas clásicos de sincronización	9
/ 10. Resumen y resolución del caso práctico de la unidad	9
/ 11. Bibliografía	10

OBJETIVOS



Conocer los problemas derivados de la sincronización de hilos.

Practicar con herramientas para sincronizar hilos.

Conocer el problema del productor/consumidor.

Utilizar monitores y semáforos.



/ 1. Introducción y contextualización práctica

En esta unidad, trataremos los problemas más comunes derivados de la sincronización de hilos en programas paralelos o concurrentes.

Veremos qué herramientas podemos utilizar para resolver estos problemas en las llamadas herramientas de sincronización. Las herramientas que vamos a ver son los monitores y los semáforos.

También analizaremos uno de los problemas más comunes en este tipo de programas, el problema del productor/consumidor.

Por último, aprenderemos a sincronizar métodos de forma nativa en Java.

Todos estos conceptos los pondremos en práctica a medida que los vayamos estudiando, así que no te preocupes, porque su asimilación será fácil.

Escucha el siguiente audio, en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.



Fig. 1. Semáforo.



Audio Intro. "Programas congelados"

<https://bit.ly/31MBnB>



/ 2. Problemas asociados a la sincronización

Cuando estamos realizando programas que utilizan la programación multitarea o concurrente, nos vamos a encontrar, inevitablemente, con el problema de la sincronización de hilos. Este problema ocurre cuando varios hilos intentan acceder al mismo tiempo a los mismos recursos compartidos. Una forma muy sencilla de comprender este problema es el siguiente: imagina que una pareja intenta sacar dinero, a la vez, de su cuenta corriente conjunta, en cajeros distintos, ¿qué crees que pasará si quisieran sacar 100€ cada uno, pero su saldo actual fuese de 150€?

Cuando los hilos intentan acceder a los recursos que comparten, pueden darse los siguientes problemas:

- **Condición de carrera:** En este caso, el resultado de ejecutar el programa dependerá del orden en que se realicen los accesos a los recursos. Por ejemplo: tenemos 2 hilos que pueden acceder a una variable llamada 'cuenta', la cual, inicialmente, tiene un valor de 10; iniciamos los dos hilos simultáneamente y uno realizará una suma y otro una resta, por lo tanto, el valor de 'cuenta' dependerá de la ejecución del hilo que se haya iniciado primero.
- **Inconsistencia de memoria:** Tenemos diferentes hilos que tienen una visión diferente de un mismo dato.
- **Inanición:** Este problema se producirá cuando a un hilo se le haya denegado continuamente el acceso a un recurso compartido al que quiere tener acceso, porque otros hilos toman el control antes que él. Es muy complicado de detectar.
- **Interbloqueo o Deadlock:** Este problema se va a dar cuando dos o más hilos estén esperando que suceda un evento que solo puede generar un hilo que se encuentra bloqueado.
- **Bloqueo activo:** El bloqueo activo se va a dar cuando tenemos dos hilos que están cambiando continuamente de estado y terminan por bloquearse mutuamente. Es un tipo de inanición, porque un proceso no deja avanzar al otro, y viceversa.



Fig. 2. Problemas de bloqueo.



Audio 1. "La exclusión mutua"
<https://bit.ly/2Fin3EH>



/ 3. Herramientas para sincronización: monitores

Para evitar gran parte de los problemas en la sincronización, podemos elegir entre varias opciones. Una de estas son los **monitores**, los cuales se encargan de gestionar el acceso a los recursos compartidos o críticos. Al fragmento de código que engloba a estos recursos, lo llamamos **sección crítica o zona de exclusión mutua**. Es una región de código a la que se accede de forma ordenada a los recursos compartidos.

Esta sección crítica tiene que ser forzosamente excluyente para los hilos, es decir: si un hilo se encuentra ejecutando su fragmento de código de sección crítica en un momento dado, ningún otro hilo podrá entrar a esa zona hasta que este no finalice. El lenguaje de programación Java nos proporciona el modificador `synchronized`, que, cuando lo aplicamos a un método, nos va a garantizar que este se va a ejecutar de forma excluyente.

A una clase que tiene un método con el modificador `synchronized`, la llamaremos monitor, debido a que, dentro de este método, se va a estar monitoreando algún recurso crítico.



Tenemos que tener muy claro que los monitores son los encargados de implementar las secciones críticas. La implementación de monitores debe tener forzosamente mecanismos que nos permitan llevar a cabo la sincronización y que actúen antes y después de entrar en la sección crítica. Estos fragmentos de código son los que vamos a tener dentro de la palabra reservada `synchronized`.

Para poder programar los monitores en Java vamos a utilizar, además de `synchronized`, los métodos:

- **`wait()`**: Con este método, si no se cumple la condición de exclusión, tendremos que esperar.
- **`notify()`**: Con el que conseguiremos que, cuando se concluya la tarea que se encuentra en la sección crítica, podamos salir de ella avisando al hilo que se encuentre esperando para que pueda entrar.
- **`notifyAll()`**: Este método tiene el mismo comportamiento que `notify`, pero la diferencia reside en que se notificará a todos los hilos que están esperando.

```
public class ContadorSincronizado {  
  
    private int valor = 0;  
  
    public synchronized void incrementar() {  
        valor++;  
    }  
  
    public synchronized void decrementar() {  
        valor--;  
    }  
  
    public synchronized int getValor() {  
        return valor;  
    }  
}
```

Código 1. Clase monitor.

/ 4. Caso práctico 1: “Sincronización en otros lenguajes de programación”

Planteamiento: Pilar y José están estudiando cuáles son los problemas derivados de la sincronización de varios hilos en la programación paralela. Todos estos problemas les están desmotivando con respecto al uso de la sincronización, ya que les parecen muy complicados de resolver, aunque ya haya herramientas disponibles para su resolución.

«Estas son las herramientas que vamos a poder usar en Java, pero, ¿qué hay de los otros lenguajes de programación?», le pregunta José a Pilar.

Nudo: ¿Crees que, en los demás lenguajes de programación, también existen herramientas para la sincronización de varios hilos? ¿O piensas que solo existen en Java?

Desenlace: La programación paralela la podremos encontrar en prácticamente todos los lenguajes de programación, para que no tengamos la opción de crear y ejecutar la concurrencia de hilos, tendríamos que estar hablando de algo muy antiguo.

Entonces, cualquier lenguaje de programación que ofrezca la posibilidad de crear hilos deberá ofrecer mecanismos para poder sincronizarlos. Sí va a cambiar la implementación de este tipo de mecanismos. No obstante, la idea será la misma: bloquear una cantidad de hilos para que vayan accediendo, de uno en uno, a las zonas críticas, por lo que habrá que tener en consideración cómo se utilizarán estos mecanismos, no cuáles son.

Ejemplos de lenguajes de programación que no son Java y que ofrecen mecanismos de sincronización de hilos pueden ser C, C++ y Python.

/ 5. Problema del productor consumidor: definición

El problema del productor/consumidor es el problema por excelencia en la **sincronización de hilos**. Este problema puede llegar a producir condiciones de carrera e inconsistencia de memoria. El problema del productor/consumidor consiste en **tener dos «agentes» que comparten un almacén o buffer con un tamaño limitado**. Uno de estos agentes, al que llamaremos productor, se encargará de colocar o de producir información en el buffer, mientras que el otro, al que llamaremos consumidor, se encargará de extraerla para realizar una operación con ella. Si el productor se dispone a colocar un nuevo elemento, pero el almacén se encuentra lleno, deberá «dormirse». El consumidor «despertará» al productor cuando haya extraído algún elemento del almacén, ya que será entonces cuando el productor pueda producir otro elemento y colocarlo en este. De igual forma, si el almacén se encuentra vacío y el consumidor necesita extraer un elemento, este debe «dormirse» hasta que el productor coloque elementos en el almacén, «despertando» en ese momento al consumidor.

Para **solucionar el problema** del productor/consumidor, necesitaremos tener dos hilos, que cubrirán el papel de agentes: uno que será el productor, encargado de generar elementos y de guardarlos, y otro que será el consumidor, encargado de extraerlos. En este problema pueden ocurrir variedad de situaciones:

- Tanto el productor como el consumidor quieren acceder a la vez al almacén.
- El productor genera a distinta velocidad que el consumidor consume.
- El productor ha llenado el almacén y no puede producir más hasta que el consumidor consuma algo.
- El consumidor no puede consumir nada porque el almacén está vacío.

DATOS

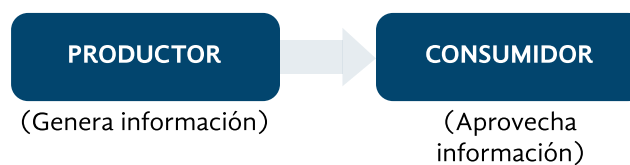


Fig. 3. Productor/Consumidor.

Este problema se puede extrapolar a tener un productor y varios consumidores, y varios productores y consumidores.



Vídeo 1. “Problema del productor/
consumidor sin monitores”
<https://bit.ly/3kEXJIY>



Puedes acceder al **código completo** del programa expuesto en el vídeo en el anexo del tema.

/ 6. Problema del productor consumidor: monitores

La forma más eficiente y sencilla de resolver el problema del productor es mediante monitores. Para ello, necesitaremos las siguientes clases:

- **Productor:** Esta será la clase que se encargará de producir los elementos que se guardarán en el almacén y que el consumidor obtendrá.




- **Consumidor:** Esta consumirá los elementos que se guardarán en el almacén y que el productor creará.
- **Buffer:** Será la encargada de almacenar los elementos que producirá el productor y que, más adelante, consumirá el consumidor. Dentro de esta clase, vamos a tener dos métodos:
 - **Método put:** Este método será el encargado de introducir un elemento dentro del buffer. Deberá estar sincronizado para que tanto productor como consumidor no accedan al mismo tiempo.
 - **Método get:** Este método será el encargado de obtener un elemento que se encuentre dentro del buffer. Igualmente, deberá estar sincronizado para que tanto productor como consumidor no accedan al mismo tiempo.


El funcionamiento de este problema será muy sencillo:

- Se lanzarán tanto productor como consumidor.
- Si el buffer está vacío, el consumidor no podrá obtener ningún valor hasta el que el productor cree y almacene uno. En este caso, será cuando el productor se active y coloque un elemento dentro del almacén.
- Si el buffer está lleno, el productor no podrá introducir ningún valor hasta que el consumidor obtenga y consuma el valor que hay dentro del buffer. En este caso, será cuando el consumidor se active y obtenga el valor que se encuentre dentro del almacén.

Al igual que en el caso anterior, esta versión del problema la podremos extrapolar a tener un productor y varios consumidores, y varios productores y consumidores.



Vídeo 2. "Problema del productor/
consumidor con monitores"
<https://bit.ly/3kBPj58>



Puedes acceder al **código completo** del programa expuesto en el vídeo en el anexo del tema.

/ 7. Semáforos

Otro de los mecanismos que nos va a proporcionar el lenguaje de programación Java para solucionar problemas de sincronización de varios hilos son los semáforos. Podemos considerar un semáforo binario como un indicador de condición de entrada que gestiona si un recurso de la sección crítica de nuestro código está disponible o no. Se dice que es binario porque va a tener dos posibles valores, 'disponible' o 'no disponible'.

En Java, podremos utilizar la clase `java.util.concurrent.Semaphore`, la cual ya está integrada en la JDK de Java y es totalmente funcional. Su especificación la podemos encontrar en el siguiente enlace:

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>

Como ya sabemos, en todos nuestros problemas tendremos una sección crítica que necesitamos proteger de accesos indebidos, la cual puede consistir en una o varias variables compartidas que los hilos van a necesitar. Mediante los semáforos podremos sincronizar todos esos hilos para que no se produzca inanición ni accesos indebidos.



Los semáforos, básicamente, tienen dos métodos:

- **Método `acquire()`:** Este método es el que nos va a permitir cerrar la sección crítica y que ningún otro hilo pueda acceder a ella.
- **Método `release()`:** Este método es el que nos va a permitir abrir la sección crítica y que otro hilo pueda acceder a ella.

Cuando creamos un semáforo, podremos indicarle el número de hilos que podrán entrar a la vez en la sección crítica. En caso de no indicar nada, será un semáforo binario y únicamente podrá acceder a la sección crítica un hilo.

```
public static void main(String[] args) {  
    Semaphore semaphore = new Semaphore(1, true);  
  
    try {  
        // Protejo la sección crítica  
        semaphore.acquire();  
        // Funcionalidad de la sección crítica  
        semaphore.release();  
    } catch (InterruptedException ex) {  
        System.out.println("Error -> " + ex.toString());  
    }  
}
```

Código 2. Ejemplo de semáforo en Java.

/ 8. Caso práctico 2: “¿Qué método de sincronización utilizar?”

Planteamiento: Pilar y José acaban de recibir su nueva tarea que consiste en realizar un programa que pueda lanzar varios hilos. Estos deberán poder acceder a una variable aleatoria que les indicará qué acción de las disponibles deberán realizar. Como es lógico, deberán sincronizar todos esos hilos.

«No pensé que la sincronización de hilos fuera un problema tan complicado de resolver, no me gusta nada», le comenta Pilar a José, a lo que él le responde que está totalmente de acuerdo, pero que, de alguna forma habrá que hacerlo, aunque no tiene muy claro cuál es la mejor.

Nudo: ¿Qué piensas al respecto? ¿Crees que hay algún método de sincronización más sencillo que prevalece sobre los demás?

Desenlace: Aunque nuestro trato con los métodos de sincronización no ha sido muy amplio, podemos ver claramente que estos problemas no son fáciles de entender ni de diseñar o programar. Como hemos visto, todos los métodos de sincronización se basan en los mismos conceptos: hacer que las zonas de memoria críticas solo sean accedidas por un hilo a la vez, bloqueando a los demás hasta que sea necesario. Visto esto, el problema que tienen que realizar nuestros amigos podría resolverse con cualquiera de los métodos que hemos estudiado.

Veamos un ejemplo básico:

```
// Método sincronizado para realizar la acción  
public synchronized realizarAccion()  
{  
    accion = getAccion();  
  
    switch(accion)  
    {  
        case 1:  
            // código  
            break;  
        case 2:  
            // código  
            break;  
    }  
}
```

Código 3. Código para resolver el problema.



/ 9. Problemas clásicos de sincronización

Además del problema del productor/consumidor, en el mundo de la programación paralela nos vamos a encontrar con otros problemas que podemos representar con las siguientes metáforas:

- **Problema de los fumadores:** Este problema consiste en un sistema con tres procesos, que serán los «fumadores», y un proceso, al que podremos considerar el «estanquero», que dará material a los fumadores. Cada fumador está continuamente deseando fumar un cigarrillo. Sin embargo, para fumar, necesita tres ingredientes: tabaco, papel y fósforos. Cada uno de los tres fumadores tendrá un material de los necesarios, por lo que necesitan que el estanquero produzca los dos que les faltan. Cuando un fumador consigue los ingredientes que le faltan, fuma, avisando al productor cuando termina, quien coloca otros dos de los tres ingredientes, repitiéndose el ciclo.
- **Los filósofos:** En este problema, tenemos a cinco filósofos que se pasan la vida pensando y comiendo. Los filósofos están sentados en una mesa circular. Delante de cada filósofo hay un plato, y hay cinco platos en total. Cuando un filósofo está pensando, no necesita comer, y cuando come, no piensa. Cuando a un filósofo le da hambre y quiere comer, necesitará 2 cubiertos, por tanto, tratará de coger los dos cubiertos más cercanos a él, pero solo podrá usar cada cubierto si no lo tiene el filósofo contiguo. Cada cubierto, por tanto, es compartido por 2 filósofos. Cuando termina de comer, vuelve a dejar sus dos cubiertos en la mesa y comienza a pensar de nuevo.
- **El barbero:** Este problema consiste en una peluquería que regenta un barbero, la cual tiene una silla de peluquero y X sillas para que se sienten los clientes que están en espera. Si no hay clientes, el barbero se sentará en su silla y dormirá, pero, cuando llega un cliente a la peluquería, el barbero se despierta y lo atiende. En el caso de que lleguen más clientes y el barbero esté ocupado, estos deberán esperar sentados, siempre que haya sillas disponibles, o salirse de la peluquería, si no las hay.

/ 10. Resumen y resolución del caso práctico de la unidad

A lo largo de esta unidad, hemos estudiado qué problemas podemos tener a consecuencia de sincronizar nuestros programas, que utilizan varios hilos en la programación paralela o concurrente.

Hemos visto qué herramientas nos ofrece el lenguaje de programación Java para solventar estos problemas. De estas herramientas, las que hemos estudiado son los monitores y los semáforos.

También hemos conocido uno de los problemas más comunes en programación paralela: el problema del productor/consumidor, donde tendremos un hilo que irá produciendo recursos y uno o varios hilos que irán consumiendo dichos recursos.

Por último, hemos aprendido cómo podemos sincronizar código de forma nativa en Java.

Resolución del caso práctico de la unidad

Uno de los mayores problemas a los que nos vamos a enfrentar cuando tengamos que realizar programas que utilicen programación multitarea o concurrente es que estos se pueden quedar congelados, quedando totalmente inservibles y teniendo que cerrarlos.

Para solucionar el ejercicio de Papá Noel, deberemos crear una variable para representar los caramelos, y una clase que representará a un niño, pudiendo crear tantos como queramos (ya que habrá muchos). La forma de acceder a los caramelos la podremos resolver de varias formas, mediante un método sincronizado, mediante monitores o mediante semáforos, ya que la variable caramelo sería nuestra zona crítica.



/ 11. Bibliografía

Exclusión mutua | *Sistemas Operativos* | *Consultoría Informática*. (s. f.). webprogramacion. Recuperado el 14 de julio de 2020 de: <https://webprogramacion.com/44/sistemas-operativos/exclusion-mutua.aspx#:~:text=Concepto%20de%20exclusi%C3%B3n%20mutua..garantice%20la%20integridad%20del%20sistema.>

Gómez, O. (2016). *Programación de Servicios y Procesos* Versión 2.1. https://github.com/OscarMaestre/ServiciosProcesos/blob/master/_build/latex/ServiciosProcesos.pdf

La cena de los Filósofos - *Aplicación de Interbloqueo e Inanición* en C#. (14 julio de 2020). [Entrada de Blog]. Disponible en: <http://ingenieroprendiz4ever.blogspot.com/2012/06/la-cena-de-los-filosofos-aplicacion-de.html>

Problema de la cena de los filósofos en Wikipedia, la enciclopedia libre. Disponible en: https://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_fil%C3%B3sofos

Problema del Productor_Consumidor. (s. f.). Recuperado el 14 de julio de 2020 de:

<https://www2.infor.uva.es/~cllamas/concurr/pract97/rsantos/index.html>

Sánchez, J. M. y Campos, A. S. (2014). *Programación de servicios y procesos*. Madrid: Alianza Editorial.

Semaphore in Java. (10 diciembre de 2018). Disponible en: <https://www.geeksforgeeks.org/semaphore-in-java/>