



# UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Estructura de Datos y Algoritmos II

*Grupo: 04 - Semestre: 2020-2*

## **Proyecto 2: Programación Paralela**

FECHA DE ENTREGA: 01/06/2020

### **Alumnos:**

López Lara Marco Antonio

Olivera Martínez Jenyffer Brigitte

Téllez González Jorge Luis



# Índice

<b>1. Objetivo</b>	<b>2</b>
<b>2. Introducción</b>	<b>2</b>
<b>3. Antecedentes</b>	<b>3</b>
3.1. Concurrencia vs Paralelismo . . . . .	4
<b>4. Problemas de la concurrencia</b>	<b>6</b>
4.1. Secciones críticas . . . . .	7
4.1.1. Condiciones de carrera . . . . .	8
4.1.2. Exclusión mutua . . . . .	9
4.1.3. Deadlocks . . . . .	10
4.1.4. Starvation . . . . .	12
4.1.5. Sincronización . . . . .	13
<b>5. El problema de los filósofos</b>	<b>15</b>
5.1. Descripción del problema . . . . .	15
5.2. La concurrencia en la resolución del problema . . . . .	18
5.3. Consideraciones del problema . . . . .	18
<b>6. Implementación en Java</b>	<b>19</b>
6.1. Monitores . . . . .	19
6.2. Descripción general . . . . .	20
6.3. Análisis . . . . .	21
6.3.1. Modelado del monitor <i>portero</i> . . . . .	21
6.3.2. Modelado del Filósofo . . . . .	22
6.3.3. Modelado de los tenedores . . . . .	24
6.3.4. Clase Principal . . . . .	25
6.4. Pruebas de validez y rendimiento . . . . .	27
<b>7. Conclusiones</b>	<b>34</b>

## 1. Objetivo

- Que el alumno ponga en práctica los conceptos de la programación paralela a través de la implementación de un algoritmo paralelo y así mismo desarrolle su capacidad para responder preguntas acerca de un concepto analizado a profundidad.

## 2. Introducción

El término de *conurrencia* en la programación puede ser observado desde múltiples perspectivas. De manera informal, un programa concurrente puede definirse como aquel que realiza *más de una tarea* en ejecución. Por ejemplo, un navegador web puede estar realizando de forma simultánea una petición a un servidor para conectarse a una página, reproducir un video y solicitar al usuario que realice una acción. Sin embargo, tales acciones *simultáneas* pueden llegar a ser meramente una simple ilusión.



Figura 1: *En un navegador web pueden ejecutarse diversas actividades 'al mismo tiempo'.*

En algunos sistemas estas actividades podrían ser realizadas por diferentes unidades de procesamiento, sin embargo, en otros sistemas estas actividades podrían ser realizadas por un único CPU que cambia entre las diferentes actividades lo suficientemente rápido al punto de que estas aparenten ser simultáneas al observador humano. Sin embargo, lo anterior no implica necesariamente que estos procesos se encuentren relacionados entre sí o que su ejecución sea al mismo tiempo. El paradigma de la programación *paralela* y la programación *concurrente* poseen una notable relación mutua, sin embargo, no aluden a los mismo conceptos.

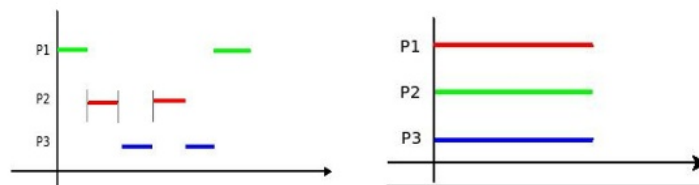


Figura 2: *Conurrencia vs. Paralelismo.*

**Edsger Dijkstra** fue un famoso científico de la computación pionero en el estudio de la programación concurrente a partir de su artículo *Procesos secuenciales de cooperación*, publicado en 1967. En este artículo, Edsger planteó los problemas más comunes de la programación concurrente como las *secciones críticas* e introdujo el uso del *semáforo* para su solución.

Hoy en día, la programación concurrente se compone de un conjunto de técnicas para expresar la concurrencia entre tareas y solucionar los problemas de comunicación y sincronización entre procesos. En el siguiente trabajo escrito se ilustrarán las semejanzas y diferencias entre los conceptos de *concurrency* y *paralelismo*, se abordarán los problemas más comunes derivados de la comunicación entre *hilos* y se abordará mediante el lenguaje de programación **Java** uno de los problemas más famosos que **Dijkstra** introdujo en el estudio de la concurrencia y su comunicación: **El problema de los filósofos**.

### 3. Antecedentes

La filosofía de poder atender más de un proceso en tiempo de ejecución es uno de los puntos clave de la programación *concurrente*, la cual consiste en la ejecución de múltiples tareas de forma interactiva. Tales tareas pueden estar formadas por un conjunto de *procesos* o *hilos* de ejecución de un único programa.

- Un hilo de ejecución se define como una secuencia de instrucciones *atómicas* (que no pueden dividirse en más tareas) en ejecución.
- Un proceso representa cualquier **secuencia de operaciones** ejecutándose en memoria activa, el cual realiza una o varias acciones sobre un cierto conjunto de datos. Un proceso forma parte de un programa y representa una entidad más compleja que un *hilo*. A su vez, un proceso puede estar conformado por uno o más hilos.

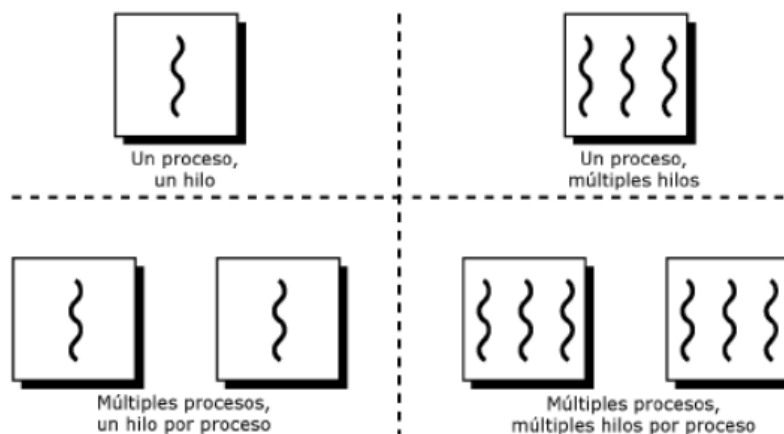


Figura 3: *Relación entre hilos y procesos.*

Cada hilo que pertenece a un proceso comparten secciones de código, datos, entre otros recursos. El *multithreading* consiste en la capacidad de proporcionar múltiples hilos de ejecución al mismo tiempo. Por otra parte, se dice que los hilos de ejecución que comparten recursos dan como resultado a un proceso.

### 3.1. Concurrency vs Parallelismo

Las semejanzas y diferencias entre estos conceptos han sido extensamente discutidas entre diversos autores; sin embargo, no se suele tener un consenso general para definirlos claramente. En general, un programa puramente concurrente es diferenciable de un programa paralelo atendiendo a lo siguiente:

- Un programa es concurrente si es capaz de soportar dos o más acciones **en progreso**.
- Un programa es paralelo si es capaz de soportar dos o más acciones ejecutándose de forma **simultánea**.

Es decir, la concurrencia se presenta cuando un programa puede manejar diversas tareas en tiempo de ejecución en un mismo CPU. Y además, será paralelo sí y solo si se encuentra diseñado para realizar sus operaciones internas por medio de hardware paralelo como *procesadores multinúcleo*, *GPU's*, *etc.* con el fin de trabajar operaciones **al mismo tiempo** y acelerar su ejecución.

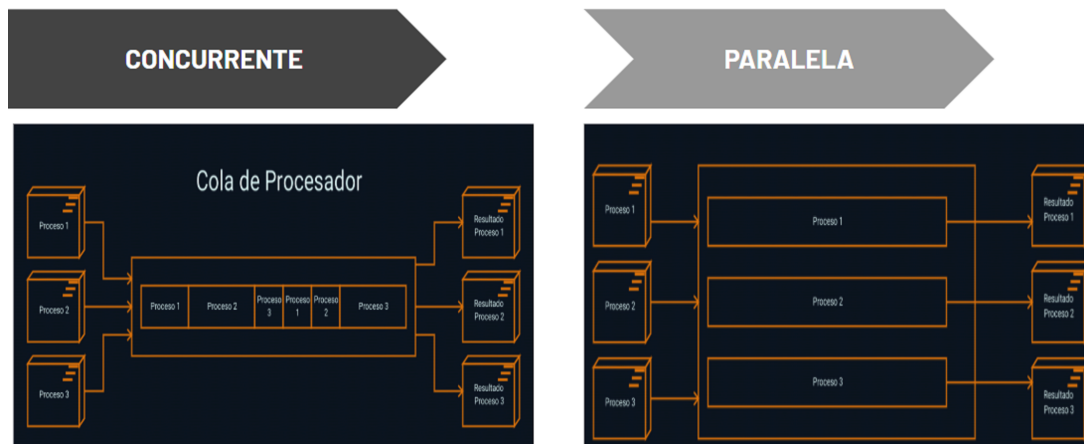


Figura 4: Comparativa de procesos concurrentes y paralelos.

La concurrencia, por tanto, se caracteriza por la ejecución de actividades que no necesariamente están involucradas entre sí. Por ejemplo, un gestor de descargas actúa como un programa *concurrente* debido a que cada descarga es independiente de las otras y no influye en el estado de las otras; más allá de los recursos de red utilizados para cada descarga.

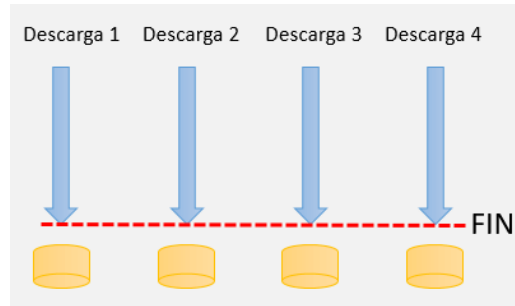


Figura 5: Operación de un programa concurrente.

Por otra parte, el caso de un hipotético buscador de ofertas sigue otro enfoque. Si desearamos realizar una búsqueda de las mejores ofertas en productos de computación en varias tiendas, buscar secuencialmente las ofertas en cada tienda resultaría prohibitivamente costoso en tiempo. En cambio, una ejecución paralela consistente en buscar **al mismo tiempo** las ofertas y unir los resultados en una sola búsqueda permitiría un ahorro drástico en el tiempo de búsqueda total.

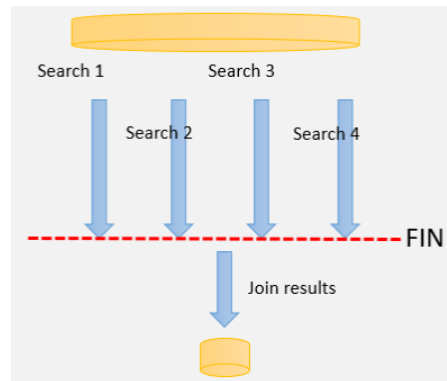


Figura 6: Operación de un programa paralelo.

Ambos conceptos, si bien poseen semejanzas notables, cuentan con diferencias que resultan importantes de destacar. Los programas concurrentes pueden aprovechar el paralelismo para realizar tareas de forma mucho más eficiente. Además, la construcción del paralelismo requiere hasta cierto punto una expresión *concurrente* o incluso *secuencial* para ser construidos: la creación de un algoritmo 100 % paralelo es una tarea prácticamente imposible. El paralelismo, por tanto, consiste en una forma de ejecutar programas concurrentes.

Lenguajes como *Java* implementan una expresión de *paralelismo* por medio de la clase **Thread**; sin embargo, de forma estricta lo anterior resulta en *conurrencia*. En diversas situaciones, ambos paradigmas pueden tener diferencias mínimas, sin embargo, resulta importante marcar las diferencias entre ambos conceptos así como sus semejanzas y los conceptos que comparten. El *paralelismo*, a su vez, comparte problemas heredados de la *conurrencia* que pueden presentarse a lo largo del desarrollo de un algoritmo

paralelo. Por ello, es importante abordar más a detalle los *inconvenientes* de la concurrencia que pueden impactar directamente en el paralelismo.

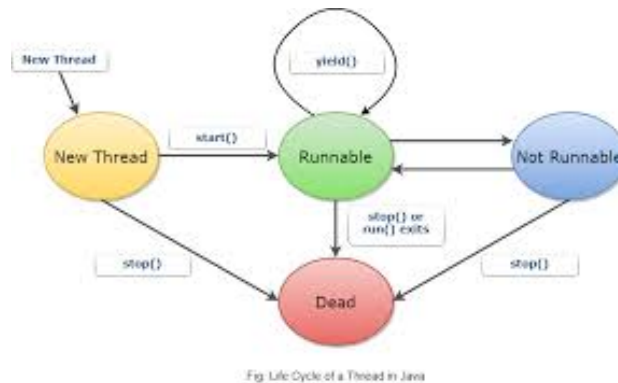


Figura 7: Java implementa paralelismo por medio de concurrencia.

## 4. Problemas de la concurrencia

El paradigma concurrente es una herramienta sustancialmente poderosa para resolver diversos problemas y necesidades computacionales; sin embargo, no está exenta de problemas críticos que puedan afectar su implementación.

Uno de los conceptos más importantes de la programación concurrente es que se basa en la *ejecución de instrucciones atómicas desde dos o más hilos*, y además, su ejecución no es **determinista**. Por lo anterior, no es posible predecir la secuencia exacta de ejecución de las instrucciones atómicas en los múltiples hilos durante cada ejecución.

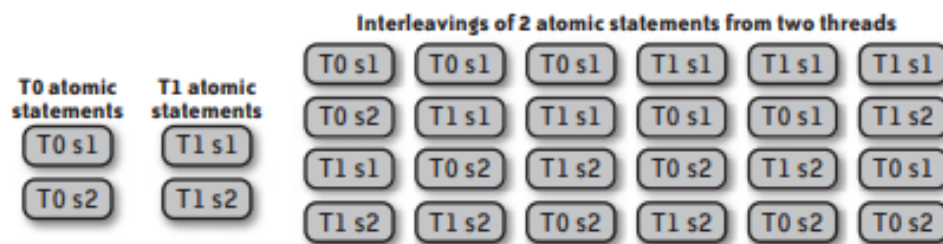


Figura 8: El orden de ejecución puede tener diversas formas, de acuerdo al número de hilos e instrucciones.

El proceso de abstracción involucrado en la escritura de programas concurrentes puede llevar a múltiples situaciones indeseables como consecuencia. La correctitud de un programa *concurrente* y, en múltiples instancias, *paralelo*, depende de que esta clase de situaciones no se presenten en la ejecución del programa.

## 4.1. Secciones críticas

Una *sección crítica* consiste en una porción de código presente en un algoritmo *concurrente/paralelo* donde existen variables *compartidas* en memoria. Los recursos de los *sistemas operativos* modernos son compartidos por múltiples procesos concurrentes.

La existencia de estas secciones resulta ser el principal motivo de discordia en la programación concurrente, debido a las **problemáticas** generadas por el acceso de diversos *hilos* a un mismo recurso durante un intervalo de tiempo determinado. Por lo anterior, constituye el problema base de la programación concurrente.

La situación presentada por las secciones críticas puede imaginarse de la siguiente forma: véase el caso de dos personas tratando de modificar un mismo documento de texto simultáneamente y sin considerar las acciones de la otra persona. El resultado final de tal situación probablemente será un documento **confuso y con múltiples errores**.

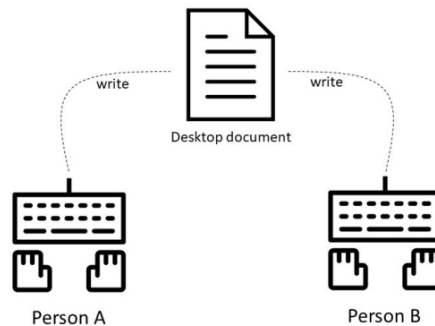


Figura 9: Acceso simultáneo a un recurso compartido.

Otra forma de ver las *secciones críticas* es observarlas como aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma completamente *concurrente*. Lo anterior implica que si un proceso inicia su ejecución haciendo uso de una sección crítica de variables compartidas, otro proceso no puede ejecutarse en tal sección crítica.

Las secciones críticas son el primer gran obstáculo que un programa *concurrente/paralelo* pueden enfrentar en su desarrollo, y su existencia trae consigo múltiples situaciones que pueden presentarse y afectar a su desarrollo.

El científico de la computación **Edsger Dijkstra** fue uno de los más importantes aportadores a la investigación de los problemas generados por la existencia de secciones críticas. A raíz de su trabajo desarrollado, se han logrado identificar con claridad las diversas situaciones que pueden presentarse en un programa *concurrente/paralelo*. Identificarlas resulta vital para obtener resultados correctos al momento de diseñar algoritmos tanto concurrentes como paralelos.



#### 4.1.1. Condiciones de carrera

Retomando la situación anterior, consideremos a ambas personas como *hilos* que se encuentran *compitiendo* por el acceso a un determinado recurso, que en este caso particular está representado por el documento a modificar. ¿Quién logrará modificar primero el documento y a qué costo? Los resultados de esta *competición* llevarán inevitablemente a un desenlace que no será el más agradable para ambas personas, y especialmente, para el receptor de tal documento.

Una *race condition* o condición de carrera es un comportamiento del software en el cual la salida depende de un orden de ejecución de eventos que **no se encuentran bajo control** y que potencialmente pueden provocar resultados incorrectos. Esta condición se presenta cuando diversos *hilos* se encuentran *compitiendo* por acceder a un recurso compartido en memoria.

- Las situaciones en las que dos o más hilos o procesos leen o escriben en un área de memoria compartida producirán una *condición de carrera*.
- El resultado final que se obtendrá dependerá de los instantes de ejecución de cada uno.
- Las condiciones de carrera a menudo generan resultados inesperados o erróneos en la ejecución de un programa.

Uno podría preguntarse: ¿Qué clase de implicaciones podría tener una *race condition* en situaciones reales? La respuesta es: **consecuencias muy serias**. Un ejemplo de esto es la máquina de radioterapia *Therac-25* que contenía en el interior de su software una condición de carrera sin resolver, además de múltiples problemas en su diseño. ¿El resultado? la muerte de 6 personas por sobredosis de radiación entre los años 1985 y 1987.

El diseño de programas concurrentes y, por ende, paralelos, deben de estar apropiadamente probado para evitar a toda costa que esta clase de situaciones se presenten en el desarrollo de programas *críticos*.



Figura 10: Las condiciones de carrera son el ingrediente perfecto para el desastre.

Cuando se tienen diversos hilos en ejecución existirán múltiples casos como el descrito anteriormente donde **no se desea** que se ejecuten al mismo tiempo; especialmente en presencia de un recurso

*compartido*. El mecanismo para controlar su aparición consiste en generar **sincronización** en su ejecución, de tal forma que la salida obtenida sea la deseada.

Por otra parte, el diseño de los programas concurrentes que brinden una solución a esta clase de problema deben de considerar algunas situaciones, las cuales serán presentadas a continuación:

#### 4.1.2. Exclusión mutua

El primer requerimiento básico de un programa *concurrente/paralelo* es que presente **exclusión mutua**, es decir, solo permita que un único *hilo* se encuentre presente en una sección crítica del programa y forzará a que cualquier otro hilo que requiera el uso de la *sección crítica* tenga que esperar hasta que el recurso sea liberado. Cualquier técnica de sincronización válida requiere implementar una *exclusión mutua* de tal forma que no se presente una *condición de carrera*.

La exclusión mutua puede ejemplificarse de forma sencilla: pensemos en el caso de 2 personas que desean probarse ropa en la sección de prendas de un supermercado. Hay una gran cantidad de personas de tal forma que únicamente queda 1 único probador disponible.

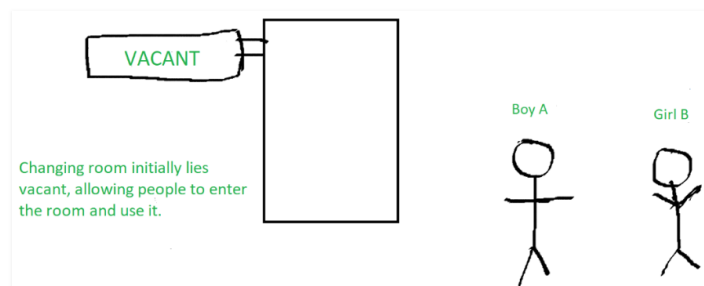


Figura 11: Probador con 2 personas en espera.

Cuando una persona entre en primer lugar al probador, este cambiará su estado de *disponible* a *ocupado*, lo que obligará a la otra persona a esperar para poder acceder al probador de ropa.

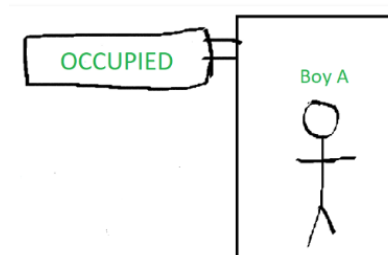


Figura 12: Acceso de una única persona al probador.

Finalmente, cuando una de las personas salga del probador, el estado volverá a cambiar a *disponible*; permitiendo a la otra persona acceder al mismo y volver a modificar su estado a *ocupado*.

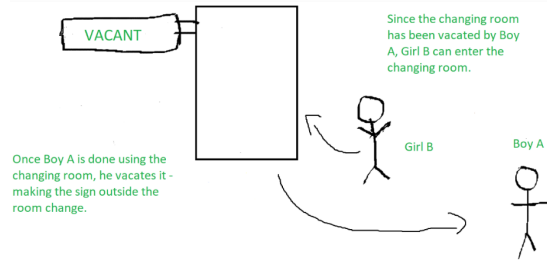


Figura 13: *El probador vuelve a estar disponible para que la otra persona lo utilice.*

En este caso particular, el probador de ropa mostrado representa nada más que la **sección crítica** del sistema, mientras que las 2 personas (El chico A y la chica B) representan dos hilos diferentes. Finalmente, el estado del probador representa el mecanismo de **sincronización** utilizado para evitar la *condición de carrera* que podría presentarse si ambas personas acceden al probador al mismo tiempo.

Una solución a este problema debe de cumplir con las siguientes condiciones esenciales:

- Debe de **garantizarse** la exclusión mutua entre los diferentes hilos al momento de acceder a la sección crítica.
- Ningún proceso que esté fuera de la sección crítica debe de interrumpir a otro para acceder a la sección crítica en cuestión.
- Cuando más de un proceso desee entrar en la sección crítica, el acceso debe de concederse en tiempo *finito*. Jamás se le tendrá esperando en un bucle sin fin.

#### 4.1.3. Deadlocks

La solución de la *exclusión mutua* podría parecer perfecta para deshacerse de la mayoría de problemas que podrían presentarse al escribir un programa *concurrente/paralelo*, sin embargo, esto no es así.

En los ejemplos anteriores se han visto casos donde los diferentes hilos involucrados idealmente se ven obligados a esperar hasta que un recurso compartido esté disponible para su uso. En todos ellos, su estado queda en *espera* hasta que el recurso esté disponible; de allí que se dice que el hilo que se encuentra actualmente en la sección crítica contiene el candado o el **lock** necesario para hacer uso del mismo.

Veamos la siguiente situación: ¿Qué pasaría si diferentes hilos o procesos desean acceder a recursos compartidos de forma simultánea? Lo ideal podría ser que, mientras uno esté utilizando el recurso, el resto espere ordenadamente a obtener su respectivo acceso. ¿Y si sucediera que estos hilos en cuestión tuviesen un recurso compartido al que los otros hilos quisieran acceder tras liberar el **lock** del recurso que se encuentran ocupando?

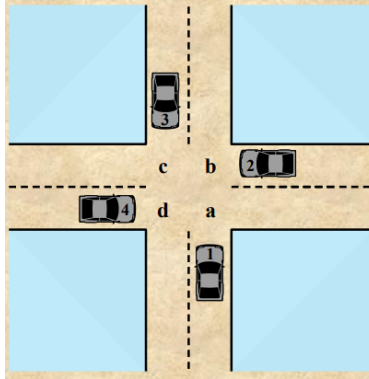


Figura 14: Una situación de interbloqueo inminente.

En la imagen anterior podemos ver cuatro automóviles que tratan de circular al mismo tiempo por *a*, *b*, *c* y *d*. Por ejemplo, para que el carro 2 pueda cruzar, deberá de pasar en primer lugar por *b* para finalmente pasar por *c* y seguir su trayecto. De igual forma, 3 pasará por *c* y *d* para continuar su camino. Véase que para que 2 pueda pasar de *b* a *c*, el lugar *c* debe de estar desocupado, pero 3 estará pasando en ese mismo instante por *c*, y para que 3 continúe su camino, *d* deberá estar disponible, ¡Pero 4 se encontrará allí!

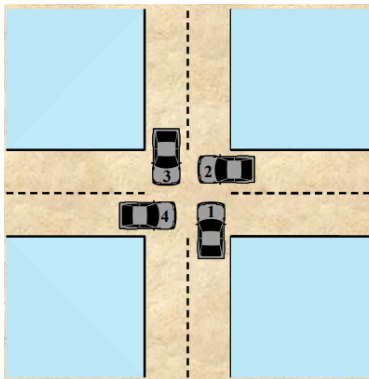


Figura 15: Interbloqueo o deadlock entre los automóviles.

Al final de todo esto, los 4 autos estarán a la espera de que los otros liberen el lugar que ocupan para proseguir su camino, sin embargo, para que puedan liberar su lugar tienen que tener el espacio disponible que solicitan para circular, pero todos los lugares están ocupados y, por tanto, no pueden circular y liberar el lugar que ocupan. *Pase usted, no, pase usted, para nada pase usted primero...*

Un **deadlock** describe una situación en la que existen dos o más hilos o procesos bloqueados que se encuentran esperándose entre sí de forma indefinida. Un *deadlock* implica que un hilo o proceso se quedará esperando un evento que nunca ocurrirá.

Para que los *interbloqueos* se presenten en un programa *concurrente/paralelo* deben de cumplirse las siguientes condiciones:



- **Exclusión mutua:** Los hilos deben de reclamar un acceso único y exclusivo a los recursos compartidos.
- **Hold and wait:** Los hilos que deseen acceder a un recurso retendrán los recursos que tengan en su posesión.
- **No Preemption:** Una vez que los hilos retengan un recurso, este únicamente podrá estar disponible cuando el hilo en cuestión libere voluntariamente el recurso compartido.
- **Espera circular:** Se presenta una cadena circular de hilos, donde cada hilo retiene uno o más recursos que el siguiente hilo de la cadena necesita.

Para prevenir la aparición de un interbloqueo, al menos una de las cuatro condiciones presentadas anteriormente tienen que negarse en el diseño del programa. Solucionar esta clase de situación puede resultar en un problema complicado, y no existe una solución que sea *óptima* para todos los programas que presenten esta situación.

#### 4.1.4. Starvation

Otro de los problemas que pueden surgir a raíz de la implementación de la *exclusión mutua* para resolver la condición de carrera reside en la aparición de otra desagradable situación: la *inanición* o *starvation*.

La *inanición* resulta ser una situación similar al *interbloqueo*, sin embargo, sus causas son diferentes. Mientras que en el interbloqueo, dos o más hilos de ejecución llegan a un punto muerto donde cada uno de ellos necesita un recurso que está siendo ocupado por otro hilo, en la inanición, sucede que uno o más procesos están esperando recursos ocupados por otros procesos que no necesariamente se encuentran en un punto muerto.

En esta situación, a un hilo de ejecución se le deniega siempre el acceso a un recurso compartido, lo cual provoca que la tarea en cuestión jamás pueda ser completada y hilo quede en espera *eterna*. Si estuviésemos en una lista de espera para la consulta del médico, y siempre que se solicita una cita esta resulta múltiples veces reprogramada por dar prioridad a otras personas, nos encontraríamos en una situación de *inanición*.

Las causas por la que puede presentarse esta situación en lenguajes como **Java** son:

- Los hilos con mayor prioridad consumen todo el tiempo de procesamiento de los hilos con menor prioridad.
- Los hilos quedan bloqueados indefinidamente esperando entrar a un bloque sincronizado.
- Los hilos que esperan un objeto indefinidamente debido a que otros hilos se encuentran activos y ocupando el objeto.

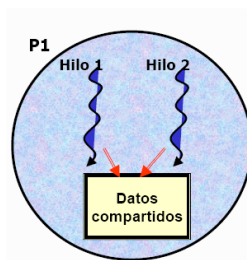
Figura 16: *La fila es eterna...*

Uno de los objetivos principales que un diseño *concurrente* debe de considerar es la **justicia** o *fairness* en el progreso *parejo* de las tareas concurrentes. Por medio de esto, debe asegurarse que *si un proceso puede ser ejecutado, será ejecutado inevitablemente*. Una situación de **injusticia** provocada por un diseño que no garantice lo anterior tarde o temprano podrá presentar un *aplazamiento infinito* en uno de sus hilos o procesos de ejecución.

#### 4.1.5. Sincronización

La *sincronización* en esencia se refiere a los mecanismos utilizados para tener control del orden de ejecución de tareas en un programa *concurrente/paralelo*. 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 un forma habitual de comunicación y sincronización entre ellos. Un programa bien diseñado deberá de garantizar que *un mensaje enviado por un hilo o proceso será siempre recibido por otro* y, así mismo, deberá de evitar la aparición de *race conditions* sorteando los problemas descritos anteriormente.

Figura 17: *La sincronización es vital para obtener resultados correctos en múltiples instancias.*

*Edsger Dijkstra* diseñó el concepto de *semáforo* como un mecanismo de sincronización y comunicación entre hilos. Los semáforos pueden definirse como componentes pasivos de bajo nivel de abstracción

utilizados para arbitrar el acceso a un recurso compartido de memoria. 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.

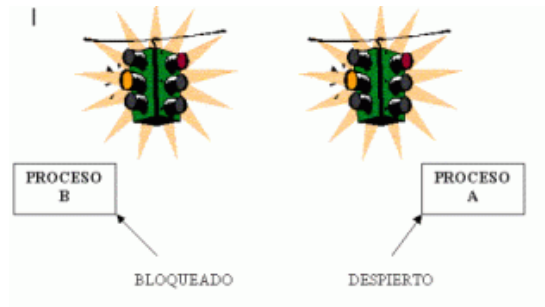


Figura 18: ¡Qué útil hubiera sido un semáforo en la calle de los interbloqueos!

El uso de un semáforo puede traer diversas ventajas:

- Implementa uso de **locks** o *permisos* para conceder acceso de un hilo a su sección crítica y gestionar su acceso.
- Garantiza exclusión mutua.
- Sincroniza dos o más hilos, de modo que su ejecución se realice de forma ordenada y sin conflicto entre ellos.

La *sincronización* no es un tema exclusivo de la concurrencia. La programación *paralela* debe de tomar en cuenta la *sincronización* al momento de utilizar variables compartidas durante su ejecución y evitar que todos accedan *al mismo tiempo*. La API de **OpenMP** implementa una *sincronización de alto nivel* con el fin de evitar que todos los *threads* representados por cada procesador de la máquina paralela accedan a realizar una operación *crítica*.

```
#include <omp.h>
main()
{
    int x;
    x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x + 1;
    } /* end of parallel section */
}
```

Figura 19: Directiva *shared* y *critical* para delimitar un bloque de hilos sincronizado.

A lo largo de las anteriores secciones se han abordado los múltiples conflictos que puede acarrear la programación *concurrente*, de allí que existen múltiples formas de presenciar la aparición de cada una de estas situaciones en la ejecución de un programa, pero veamos más allá. ¿De qué forma podemos ejemplificar cada una de estas situaciones en un solo problema?

Sentemonos a pensar y reflexionar, incluso, puede que sea buena idea parar un momento, sentarse en una mesa, y empezar a *filosofar*...



Figura 20: ¿Qué hace un filósofo cenando aquí?

## 5. El problema de los filósofos

El problema de los *filósofos comensales* (*dining philosophers problem*) fue *propuesto* por primera vez por **Edsger Dijkstra** en 1965 para representar el problema de la *sincronización* de los procesos en los sistemas operativos, así como las posibles técnicas adecuadas para implementarla. La versión actual fue formulada **Tony Hoare** en 1985 como una reformulación del concepto inicial de Dijkstra: un ejercicio propuesto de examen presentado en términos de computadoras compitiendo por el acceso a unidades de cinta.

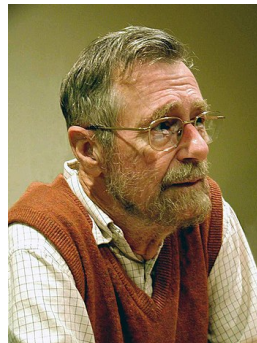


Figura 21: *Edsger Dijkstra*.

### 5.1. Descripción del problema

El problema de los filósofos comensales enuncia la siguiente situación:



”Se tiene un  $n$  número de filósofos sentados en una mesa, que pasan toda su vida cenando y pensando. Cada uno de los filósofos tiene un plato de espaguetis delante de él y entre cada uno de ellos hay un tenedor. Para que puedan comer del plato es necesario que tengan dos tenedores (el de su izquierda y el de su derecha). Si un filósofo desea comer agarra un tenedor, y si el otro no está disponible, se quedará con el tenedor en la mano esperando hasta que se desocupe el segundo, para finalmente disfrutar su plato hasta que quede saciado, deje los tenedores en la mesa y proceda a seguir pensando. El filósofo no podrá comer hasta que obtenga sus dos tenedores respectivos”.

- Cada filósofo tiene 3 posibles estados: **piensa, tiene hambre y come**.
- El objetivo del problema es crear un algoritmo, o disciplina de comportamiento, que asegure que los filósofos podrán alternar continuamente entre los estados de *comer* y *pensar* sin que uno de los filósofos espere de forma indefinida tomar un tenedor y, por tanto, muera de hambre.
- Se asume que los filósofos son silenciosos, y por tanto, la comunicación entre ellos es despreciable. De allí que el algoritmo tiene que implementar un método de sincronización efectivo que asegure **justicia** y que cada filósofo pueda desarrollar sus actividades con normalidad.

El concepto de **justicia** o el *fairness* se refiere a que un programa que tenga como propiedad la ausencia de *inanición* en sus hilos de ejecución; lo que asegura que cada hilo que solicite un **lock** para acceder al recurso compartido eventualmente podrá acceder a la sección crítica. Por otra parte, lo anterior no realiza ninguna garantía de cuanto tiempo puede tomar en acceder. Un requerimiento importante en este punto es que el tiempo de espera de un hilo para acceder a la sección crítica debe de estar acotado en un determinado intervalo y ser estrictamente *finito*.

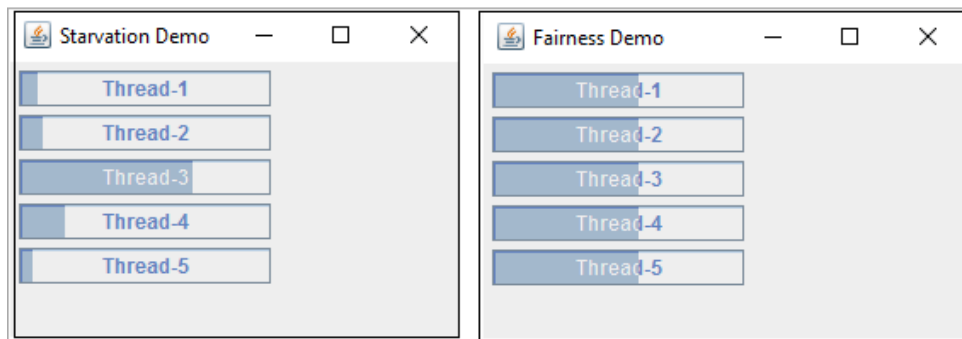


Figura 22: Ejemplo de hilos en inanición y en justicia.

En el ejemplo anterior, es posible observar que el *Thread 3* se lleva completamente al resto de hilos en cuanto al acceso a una determinada región crítica; en detrimento del resto. Una situación de justicia *ideal* estará presente cuando todos los hilos puedan acceder con la misma cantidad de oportunidad a la sección crítica. Lo anterior resulta una situación ideal; su cumplimiento puede estar afectado por otras variables fuera de control para el programador.

Cuando dos filósofos continuos intentan tomar el mismo tenedor, entrarán en una condición de carrera; la cual resultará en uno de ellos agarrando el tenedor y el otro esperando hasta que lo libere. Si todos los filósofos toman el tenedor a su derecha, cada uno de ellos se quedarían esperando *eternamente* a que se desocupe el tenedor a su izquierda y morirían de hambre. Es decir, se presentaría un **intebloqueo** entre ellos.



Figura 23: *Mesa de los filósofos que piensan, comen y tienen hambre.*

La pregunta para este problema es **¿Cómo hacer para que los filósofos no se mueran de hambre?** Este problema fue propuesto hace cinco décadas para poder representar el problema de sincronización al momento de compartir recursos en sistemas multiprogramados, por ello para este problema existen varias soluciones con distintos grados de efectividad.

En un *sistema operativo multiprogramado*, la sincronización se refiere al mecanismo que permite a dos o mas procesos realizar acciones al mismo tiempo: por ejemplo, que un filósofo coma mientras los demás están pensando o esperan un tenedor. En este tipo de sistemas, cada proceso se ejecuta *asíncronamente* y, en algunos instantes, los procesos deben sincronizar sus actividades cuando la continuación de uno depende de alguna tarea que debe de realizar otro o un recurso que está siendo **ocupado** por el mismo.

Los procesos *compiten* entre ellos por el acceso a los recursos o *cooperan* para comunicar información. Ambas situaciones las maneja el sistema operativo mediante sincronización que permite el acceso exclusivo de forma coordinada a los recursos; en este caso particular, que cada filósofo pueda comer.

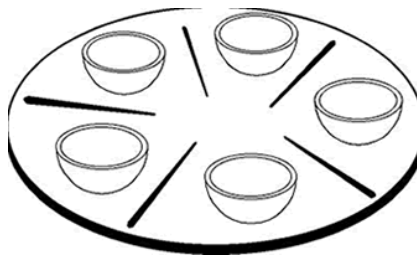


Figura 24: *La sincronización entre los filósofos permitirá que cada uno siga con sus actividades.*

## 5.2. La concurrencia en la resolución del problema

Cuando el recurso compartido no se encuentra disponible (tenedores o palillos, si fueran fideos), se hace necesaria la comunicación entre recursos que se ejecutan en paralelo. En el caso particular de este problema tratar de implementar una solución puramente *secuencial* resulta imposible, ya que implicaría que solo un filósofo pueda comer mientras el resto está pensando o muriendo de hambre.

Dado que dos filósofos no puede utilizar el mismo tenedor a la vez, es necesario implementar sincronización a la hora de utilizarlos. En la realidad, un filósofo representa a un *hilo de ejecución* y un tenedor representa a un *recurso compartido* de uso exclusivo.

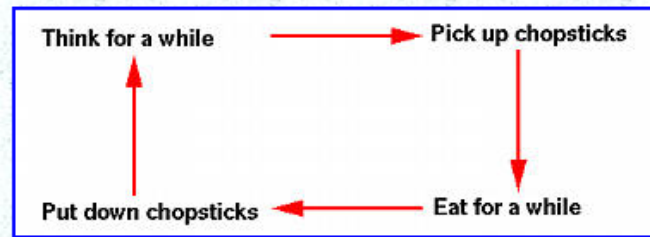


Figura 25: Cada filósofo realiza su propio ciclo de actividades independiente del resto.

## 5.3. Consideraciones del problema

Para evitar un bloqueo indefinido entre los filósofos una de las siguientes reglas no se ha de cumplir:

- Exclusión mutua: Por lo menos un recurso debe retenerse en modo no compartido; es decir, sólo un proceso a la vez puede usar el recurso. Si otro proceso solicita el recurso, deberá esperar hasta que se haya liberado.
- Retención y espera: Debe haber un proceso que retenga por lo menos un recurso y espere adquirir otros recursos retenidos por otros procesos.
- No apropiación: Los recursos no se pueden quitar; es decir, un recurso solo puede ser liberado voluntariamente por el proceso que lo retiene, después de que haya cumplido su tarea.
- Espera circular: Debe haber un conjunto  $P_0, P_1, \dots, P_n$  de procesos en espera tales que  $P_0$  espera un recurso retenido por  $P_1$ ,  $P_1$  espera un recurso retenido por  $P_2$ , sucesivamente.

Siempre se debe tener en cuenta que:

- Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.
- Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se

quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces, los filósofos se morirán de hambre.

## 6. Implementación en Java

La solución al problema implementada en **Java** que será presentada y analizada es una modificación al código autoría del programador *José Francisco Sánchez Portillo* con el fin de ejecutarse únicamente por medio de consola. La versión *gráfica* de este programa es analizada y observada a detalle en el video correspondiente del proyecto.

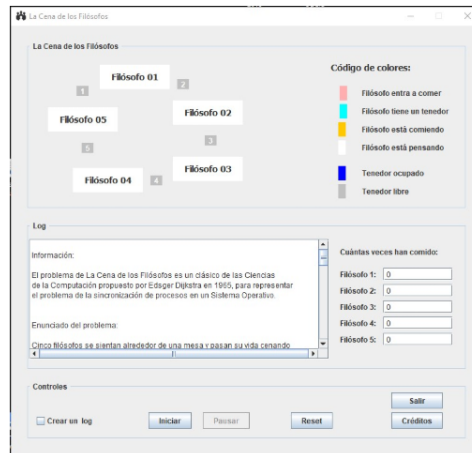


Figura 26: Programa en su versión gráfica.

### 6.1. Monitores

Un monitor representa una solución que implementa una región crítica condicional, de forma que este puede considerarse como una construcción con un cuarto especial; tal que ese cuarto solo puede ser ocupado por un solo *hilo* en un tiempo determinado. Los monitores implementan el concepto de **locks** o bloqueos.

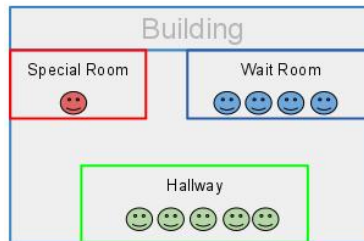


Figura 27: Abstracción del monitor.

Si uno de los *hilos* desea ocupar el cuarto especial, deberá entrar al pasillo para esperar. El planificador eligirá a un hilo bajo un determinado criterio (FIFO, por ejemplo). Si su ejecución se suspende por algún motivo, será enviado a la sala de espera, y será reprogramado para entrar a la sala especial más tarde. En resumen, un monitor permite acceso a los hilos a la *región crítica*, asegurándose de que solo un único hilo acceda a la misma.

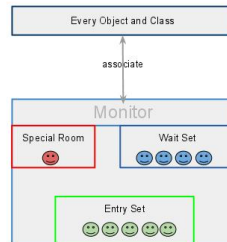


Figura 28: Asociación entre monitores y clases u objetos.

En la máquina virtual de **Java**, cada objeto y clase se encuentra *asociada lógicamente* a un monitor. Para implementar la *mútua exclusión*, todos los métodos de la clase deben de ser **synchronized**. Los métodos más relevantes son:

- **wait()**: Si una condición no se cumple, se esperará una cantidad finita de tiempo.
- **notify()**: Cuando un hilo entra a la región crítica, y si se ha realizado cierta acción, se notifica a un proceso o hilo que está esperando para entrar si se cumple su condición de entrada, es decir, si despierta del **wait()**.
- **notifyAll()**: Se realiza la misma acción pero se notifica a todos los hilos que se encuentran esperando.

## 6.2. Descripción general

El programa que implementa el algoritmo o disciplina de comportamiento para los filósofos implementa el uso de *monitores* con el fin de evitar la aparición de interbloqueos. La ejecución del programa no tiene límite con el fin de demostrar que el programa no presenta *interbloqueos* ni *inanición* a largo plazo. El programa permite modificar el número de filósofos presenten en la mesa como *comensales*. Finalmente, se imprimen las actividades de cada filósofo, junto con el tiempo que tardó en realizar alguna acción como *comer* o *esperar*.

El programa se encuentra compuesto por clases: *Filósofo*, *Tenedor*, *PorteroDelComedor* y la clase *Principal*. Un punto importante a destacar es que, con el fin de implementar una disciplina de comportamiento entre los filósofos se ha optado por el uso de *monitores*, donde a cada filósofo que desea alimentarse se le asignará un lugar como *comensal* en la mesa; si este se encuentra disponible.

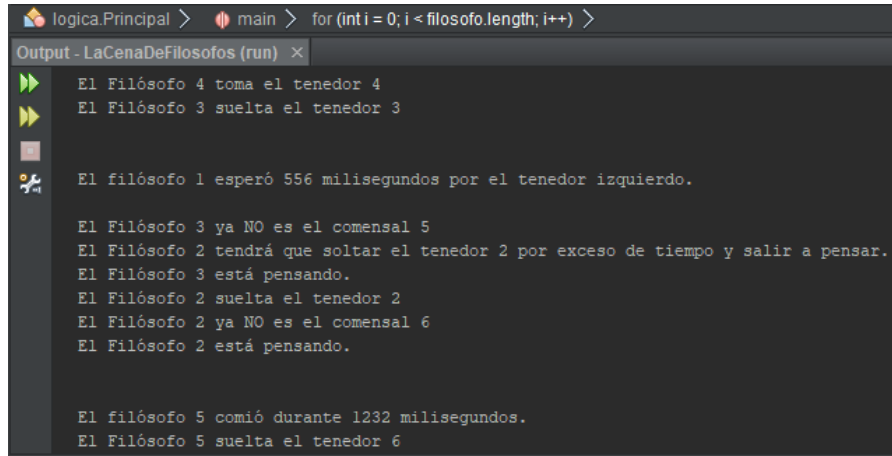


Figura 29: Vista previa del programa en ejecución.

## 6.3. Análisis

### 6.3.1. Modelado del monitor *portero*

El monitor empleado para implementar el algoritmo de comportamiento está contenido en la clase *Portero\_del\_Comedor*, la cual tiene como único atributo privado el entero *comensales*, que representa el número de filósofos que comerán en la mesa en total *menos una unidad*. A raíz de esto, el constructor del *portero del comedor*, será igual a  $\text{filosofos} - 1$ .

```
/**
 * Constructor de la clase Portero_del_Comedor
 *
 * @param comensales
 */
public Portero_del_Comedor(int comensales){
    this.comensales = (comensales - 1);
}
```

Figura 30: Inicialización del portero del comedor de los filósofos.

Este monitor implementa dos métodos para la asignación de lugares en el comedor de los filósofos:

- **public synchronized void tomarComensal(int id\_f):** Este método en primer lugar verifica si existe un lugar disponible en la mesa de comensales (denotado con el atributo *comensales*); en caso de que no existan lugares disponibles (*comensales* = 0), el hilo o filósofo que invoque al método se verá obligado a esperar por medio del método **wait**.

Si en cambio, se encuentra un lugar disponible denotado por un valor *comensales* > 0, se imprimirá un mensaje de estado indicando que el filósofo con el identificador recibido como parámetro será

el comensal  $n$ ; valor basado en el contador interno de lugares disponibles que es decrementado al ocuparse un lugar.

```
public synchronized void tomarComensal(int id_f) throws InterruptedException{
    while(comensales==0){ // Si no hay comensales libres toca esperar
        this.wait();
    }
    System.out.println("El Filósofo " + (id_f+1) + " es el comensal " + comensales);
    comensales--; // Conteo de comensales
```

Figura 31: Inicialización del portero del comedor de los filósofos.

- **public synchronized void soltarComensal(int id\_f):** A través de este método se libera un *lugar* en la mesa de los filósofos *comensales*, incrementado *comensales++* e imprimiendo un mensaje de estado indicando que un filósofo identificado con su id único ya no ocupa un lugar en la mesa. Finalmente, se invoca el método **notify** para avisar a los otros *filósofos* que se encuentra un lugar disponible en la mesa para comer.

```
System.out.println("El Filósofo " + (id_f+1) + " ya NO es un comensal y se retira de la mesa.");
comensales++; // Conteo de comensales
this.notify(); // Notificación al siguiente de que hay comensal disponible.
```

Figura 32: Inicialización del portero del comedor de los filósofos.

### 6.3.2. Modelado del Filósofo

La clase *Filósofo* es la encargada de modelar el comportamiento de los filósofos en el problema planteado. Esta clase hereda de *Thread* y tiene los siguientes atributos:

- Una inicialización de un objeto de tipo *Random*.
- **private int rand:** Utilizado para almacenar un número aleatorio, utilizado para modelar el tiempo que cada filósofo pasa pensando o alimentándose.
- **int id:** Identificador de cada *Thread* filósofo.
- **private Tenedor, izqda, dcha:** Los tenedores que cada filósofo utilizará. Los tenedores son modelados por medio de la clase *Tenedor*.
- **Portero.del.Comedor comensal:** Declarado para implementar un monitor para los filósofos.

El constructor de la clase cuenta con cuatro parámetros de inicialización, los cuáles representan los atributos de cada *filósofo* en ejecución:

- **id:** Identificador único de cada filósofo.
- **izqda:** Tenedor izquierdo asignado al filósofo.

- **dcha**: Tenedor derecho asignado al filósofo.
- **comensal**: Lugar asignado al filósofo en la mesa para comer.

La clase cuenta con una implementación del método **run**. Este método se ejecuta de forma *infinita* con el fin de verificar el correcto comportamiento del algoritmo o disciplina implementada en los filósofos con el objetivo de que no se *interbloqueen* o alguno muera de hambre. De forma similar al problema del *productor-consumidor* abordado en la *Práctica 13* del laboratorio de EDA II, este proceso se ejecuta continuamente por cada hilo. Su procedimiento en tiempo de vida es el siguiente:

1. En primer lugar, se introduce un bloque **try-catch** donde, por medio del atributo *comensal* de cada hilo, se ejecutarán los métodos **tomarComensal** y **tomarTenedor**, enviando como parámetro su **id** para identificarlo en su proceso de petición y acceso al recurso compartido. Un punto importante en la petición es que el filósofo únicamente tomará el tenedor ubicado a su derecha utilizando el atributo **dcha**.

```
// Obtener el número de comensal para poder comer.
comensal.tomarComensal(id);

// Obtener el Tenedor Derecho.
dcha.tomarTenedor(id);
```

Figura 33: Asignación de lugar en la mesa y acceso al tenedor derecho compartido.

2. A continuación, se establece una condicional **if** tal que, si el método **tomarTenedorIzqdo** invocado por medio de **izqda** retorna *false* (es decir, no tiene un tenedor izquierdo disponible para tomar), el hilo *filósofo* se verá obligado a soltar el tenedor derecho que tomó, dejar el lugar que ocupó en la mesa, y salir a pensar al *balcón*.

```
// Si no se consigue el izquierdo: el filósofo tendrá que volver a casilla de salida y volver a obtener el número de comensal:
System.out.println("El Filósofo " + (id+1) + " tendrá que soltar el tenedor " + (id+1) + " por exceso de tiempo y salir a pensar.");

// Como no ha conseguido el Tenedor izquierdo suelta el derecho
dcha.soltarTenedor(id);
```

Figura 34: Qué mal servicio tienen en este restaurante...

3. Posteriormente, se introduce otro bloque **try-catch** anidado. El tiempo que el filósofo estará pensando será obtenido por medio del método **nextInt** haciendo uso del atributo **random** declarado previamente; este valor se encontrará entre un intervalo de 100 a 1000ms. Durante ese intervalo, el hilo *filósofo* será enviado a dormir utilizando el método **sleep**, enviando como atributo el número aleatorio obtenido previamente. Finalmente, cuando el hilo despierte de su estado durmiente, se imprimirá un mensaje de estado indicando el tiempo total que el filósofo *pensó* en milisegundos.



```
rand = random.nextInt(1000) + 100;  
// El tiempo que tarda el filósofo en pensar, entre 100 y 1000 milisegundos:  
Filosofo.sleep(rand);
```

Figura 35: *Hora de filosofar.*

4. Utilizando la sentencia **continue**, y cumpliéndose la condición presentada por el **if**, el filósofo tratará de nuevo de obtener un lugar y sus respectivos tenedores para poder comenzar a alimentarse. Si tal proceso es exitoso, procederá a consumir sus espaguetis una cantidad de tiempo aleatorio (entre 0.5 y 1 segundo) y, posteriormente, el hilo será enviado a *dormir* la cantidad aleatoria obtenida con **random.nextInt**. Finalmente, se imprime un mensaje de estado indicando que el filósofo consumió su plato en una determinada cantidad de milisegundos. Todo esto en su respectivo bloque **try-catch**.

```
try {  
    rand = random.nextInt(1000) + 500;  
    sleep(rand);  
    System.out.println("\n\nEl filósofo " + (id+1) + " comió durante " + rand + " milisegundos.");  
}
```

Figura 36: *Hora de comer!*

5. Una vez que el filósofo ha terminado de comer, soltará ambos tenedores haciendo uso del método **soltarTenedor** en ambos atributos (**izqda** y **dcha**) y dejará el lugar que ocupa en la mesa para proceder a ir al 'balcón' a pensar de forma idéntica a la explicada en el paso 3.

### 6.3.3. Modelado de los tenedores

La clase *Tenedor* es la encargada de modelar a los tenedores que cada filósofo deberá de compartir para comer sus respectivos espaguetis. Como sus principales atributos contiene:

- Una inicialización a un objeto de tipo *Random*.
- **private int rand**: Utilizado para almacenar un número aleatorio, utilizado para modelar el tiempo que cada filósofo pasa pensando.
- **private int id**: Identificador de cada tenedor disponible.
- **private boolean libre**: Determina si un tenedor se encuentra disponible: *true* disponible o *false* en caso contrario. Por defecto se establece en *true*.

A continuación se tiene el método constructor de la clase que asigna un entero recibido como el **id** o el identificador único de cada tenedor. Y, posteriormente, se implementan tres métodos *sincronizados* para implementar el manejo de los tenedores por cada filósofo:

- **tomarTenedor(int id f)**: Este método es utilizado como monitor para que un filósofo tome un tenedor derecho. Recibe el ID del filósofo que desea tomarlo y, mientras el tenedor en cuestión tenga como

valor *libre* = *false*, el hilo filósofo que desea acceder al mismo será puesto en espera por medio del método **wait**; esto contenido en un ciclo **while** que se ejecutará mientras *libre* mantenga el valor *false*. En caso de que esté disponible, se imprimirá un mensaje en pantalla indicando que el filósofo ha tomado un determinado tenedor derecho y cambiando el valor de *libre* a *false*.

```
System.out.println("El Filósofo " + (id_f+1) + " toma el tenedor derecho:" + (id+1));  
libre = false;
```

Figura 37: Impresión del mensaje en pantalla de confirmación.

- **tomarTenedorIzqdo(int id\_f)**: Este método también implementa un monitor con el fin de tomar los tenedores izquierdos. A diferencia del método anterior, dentro del ciclo **while** con la condicional **!libre**, el filósofo que intentó tomar el tenedor izquierdo ocupado se verá obligado a esperar una cantidad aleatoria de tiempo (entre 0.5 y 1 segundo) por medio del método **random.nextInt**, recuperando el valor en la variable *rand* y utilizandola como parámetro en el método **wait**. Posteriormente, se imprime un mensaje en pantalla indicando que el filósofo esperó una determinada cantidad de tiempo para poder tomar el tenedor izquierdo y el método devolverá el valor booleano *false*.

```
this.wait(rand); // Sólo espera aleatoriamente entre 0.5 y 1 seg y si no, retorna false  
System.out.println("\nEl filósofo " + id_f + " esperó " + rand + " milisegundos por el tenedor izquierdo.\n");  
return false;
```

Figura 38: Espera del filósofo para acceder al tenedor izquierdo.

En caso de que el tenedor izquierdo se encuentre disponible, se imprimirá un mensaje en pantalla indicando que el filósofo ha tomado un tenedor izquierdo, y el atributo *libre* asociado a tal tenedor será establecido en *false* y, finalmente, el método devolverá *true*.

- **soltarTenedor(int id\_f)**: El último método al ser invocado modificará el valor del atributo *libre* de un cierto tenedor en *true*, se imprimirá un mensaje en pantalla indicando que un hilo filósofo ha soltado un determinado tenedor y se enviará una notificación por medio de **notify**; desbloqueando los hilos que se encuentran a la espera de un tenedor.

```
System.out.println("El Filósofo " + (id_f+1) + " suelta el tenedor " + (id+1));  
this.notify();
```

Figura 39: Liberación de tenedores por parte de un hilo filósofo.

#### 6.3.4. Clase Principal

La clase contenida en el paquete *lógica* contiene el método o hilo principal de ejecución del programa. En primer lugar, se establece una variable *n* entera usada para delimitar el total de filósofos que se generarán en el programa. El programa está capacitado para trabajar con *n* filósofos en su ejecución, además, por cada filósofo hay un tenedor.

Posteriormente, se crean 2 arreglos: el primer arreglo contiene las  $n$  instancias elegidas para contener los tenedores compartidos y el segundo contendrá todos los *Threads* que modelan a los filósofos; los actores principales del problema. Por último, se crea una única instancia del *Portero del comedor*.

```
// Se crea el Array para contener las n instancias de Tenedores:
Tenedor[] tenedor = new Tenedor[n];

// Se crea el Array para contener las n instancias de Filósofos:
Filosofo[] filosofo = new Filosofo[n];

// Se crea una sola instancia de Portero_del_Comedor:
Portero_del_Comedor comensal = new Portero_del_Comedor(n);
```

Figura 40: Generación de los arreglos contenedores de tenedores y filósofos así como del portero.

Un aspecto importante es que el valor  $n$  es enviado al inicializar el portero por medio del constructor con el fin de que asigne la cantidad máxima de comensales en la mesa, la cuál será de  $n - 1$  con el fin de asegurar que siempre haya un lugar para los comensales comiendo y que, por consecuencia, hayan suficiente tenedores para cada filósofo que desee comer. Luego, se inicializan las instancias de tenedores y filósofos por medio de dos ciclos **for**. Para inicializar a los filósofos se les da un identificador por medio de la variable iteradora  $i$ , se envía un tenedor izquierdo y posteriormente un tenedor **izquierdo**; este se contabiliza con una operación *mod*, ya que cuando se llega a  $n-1$ , el siguiente tenedor resultará en 0 y afectará al flujo del programa.

```
// Se crean las n instancias de Filósofos:
for (int i = 0; i < filosofo.length; i++) {

    filosofo[i] = new Filosofo(i, tenedor[i], tenedor[(i + 1) % n], comensal);
```

Figura 41: Generación de los arreglos contenedores de tenedores y filósofos así como del portero.

Una vez hecho lo anterior, se procede por medio de otro ciclo **for** a iniciar la ejecución de cada *Thread* filósofo por medio del método **start**. Como complemento de la ejecución del programa, se utiliza un objeto de tipo *Timer* con el fin de medir en segundos el tiempo total de ejecución del programa.

```
Timer timer = new Timer();
timer.scheduleAtFixedRate(timerTask, 0, 1000);

}

static TimerTask timerTask = new TimerTask() {
    @Override
    public void run() {
        System.out.println("\n***Ha transcurrido 1 segundo de ejecución!***\n");
```

Figura 42: Timer utilizado para delimitar el tiempo de ejecución del programa en segundos.

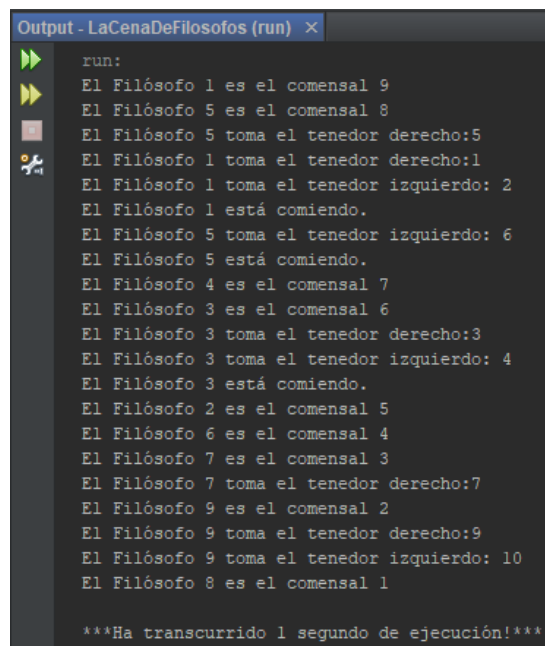
## 6.4. Pruebas de validez y rendimiento

Los objetivos de las pruebas de esta implementación son los siguientes:

- Verificar que el programa no presenta *interbloqueos* entre filósofos.
- No debe de presentarse un caso de *inanición* en alguno de ellos.
- Comprobar que el tiempo de espera para acceder al recurso compartido (específicamente, el tenedor izquierdo) es finito.
- El programa debe de ejecutarse de forma estable y sin presentar errores en su ejecución.

Para lo siguiente, y por conveniencia, se han tomado los primeros 5 segundos de ejecución para verificar el funcionamiento del programa. Para delimitar este lapso, se ha utilizado el *Timer* implementado en la versión del programa en consola.

A continuación, se mostrarán capturas de la ejecución inicial del programa. En este primer punto puede verse la asignación inicial de los filósofos a la mesa donde estarán comiendo sus espaguetis. Como puede verse, el filósofo 1 y el filósofo 5 consiguen sentarse en primer lugar en la mesa y, por tanto, pueden tomar los tenedores izquierdo y derecho sin problemas y comenzar a comer de su plato. Por otra parte, los filósofo 3 y 9 también logran obtener un lugar y sus dos respectivos tenedores, sin embargo, el resto de filósofos no tienen la misma suerte y se verán obligados a esperar por sus tenedores.



```
run:
El Filósofo 1 es el comensal 9
El Filósofo 5 es el comensal 8
El Filósofo 5 toma el tenedor derecho:5
El Filósofo 1 toma el tenedor derecho:1
El Filósofo 1 toma el tenedor izquierdo: 2
El Filósofo 1 está comiendo.
El Filósofo 5 toma el tenedor izquierdo: 6
El Filósofo 5 está comiendo.
El Filósofo 4 es el comensal 7
El Filósofo 3 es el comensal 6
El Filósofo 3 toma el tenedor derecho:3
El Filósofo 3 toma el tenedor izquierdo: 4
El Filósofo 3 está comiendo.
El Filósofo 2 es el comensal 5
El Filósofo 6 es el comensal 4
El Filósofo 7 es el comensal 3
El Filósofo 7 toma el tenedor derecho:7
El Filósofo 9 es el comensal 2
El Filósofo 9 toma el tenedor derecho:9
El Filósofo 9 toma el tenedor izquierdo: 10
El Filósofo 8 es el comensal 1

***Ha transcurrido 1 segundo de ejecución!***
```

Figura 43: Primer segundo de ejecución del programa.

Siguiendo con la ejecución, se puede observar que finalmente el filósofo 7 logra tomar el tenedor izquierdo y, junto con 9, comienzan a comer. Luego, 7 termina de comer en 503ms, suelta sus tenedores y se retira de la mesa para ir a pensar al balcón. Posterior a esto, 8 toma el tenedor a su derecha y el filósofo 10 finalmente ya tiene acceso a la mesa como comensal. 7 termina de pensar en un total de 194ms y, posterior a eso, solicita un lugar para sentarse de nuevo en la mesa; lugar que obtiene de 5 que suelta sus tenedores y se retira de la mesa a pensar. Así mismo, 6 obtiene sus cubiertos y procede a comer.

```
El Filósofo 7 toma el tenedor izquierdo: 8
El Filósofo 9 está comiendo.
El Filósofo 7 está comiendo.

El filósofo 7 comió durante 503 milisegundos.
El Filósofo 7 suelta el tenedor 8
El Filósofo 7 suelta el tenedor 7
El Filósofo 8 toma el tenedor derecho:8
El Filósofo 7 ya NO es un comensal y se retira de la mesa.
El Filósofo 7 está pensando.
El Filósofo 10 es el comensal 1

El filósofo 7 pensó durante 194 milisegundos.

El filósofo 5 comió durante 724 milisegundos.
El Filósofo 5 suelta el tenedor 6
El Filósofo 6 toma el tenedor derecho:6
El Filósofo 6 toma el tenedor izquierdo: 7
El Filósofo 6 está comiendo.
El Filósofo 5 suelta el tenedor 5
El Filósofo 5 ya NO es un comensal y se retira de la mesa.
El Filósofo 7 es el comensal 1
El Filósofo 5 está pensando.

***Ha transcurrido 1 segundo de ejecución!***
```

Figura 44: *Segundo 2 de ejecución del programa.*

El tercer segundo trae consigo resultados interesantes. 1 termina de comer en 1084ms y se retira de la mesa y 2 aprovecha para tomar el tenedor a la derecha del filósofo 1 (recordando que, en esta implementación, los tenedores con contiguos y los tenedores noes son considerados izquierdos y los pares se consideran derechos). Luego, 9 termina de comer, suelta sus cubiertos y se retira a pensar. 10 aprovecha que 9 ha terminado y procede a consumir su plato.

Para la mala fortuna del filósofo 8, este esperó una cantidad demasiado elevada de tiempo (1061ms) por el tenedor izquierdo y se verá obligado a soltar el tenedor derecho que había tomado y retirarse de la mesa a pensar un rato. 9 entra de nuevo a la mesa y toma un tenedor derecho. 3 termina de comer y 4 inicia a comer. Posteriormente, 2 se verá obligado a dejar el tenedor que tomó y deberá salir a pensar al balcón.

```
El filósofo 1 comió durante 1084 milisegundos.
El Filósofo 1 suelta el tenedor 2
El Filósofo 2 toma el tenedor derecho:2
El Filósofo 1 suelta el tenedor 1
El Filósofo 1 ya NO es un comensal y se retira de la mesa.
El Filósofo 1 está pensando.

El filósofo 9 comió durante 1172 milisegundos.
El Filósofo 9 suelta el tenedor 10
El Filósofo 10 toma el tenedor derecho:10
El Filósofo 10 toma el tenedor izquierdo: 1
El Filósofo 10 está comiendo.
El Filósofo 9 suelta el tenedor 9
El Filósofo 9 ya NO es un comensal y se retira de la mesa.
El Filósofo 9 está pensando.

El filósofo 8 esperó 1061 milisegundos por el tenedor izquierdo.

El Filósofo 8 tendrá que soltar el tenedor 8 por exceso de tiempo y salir a pensar.
El Filósofo 8 suelta el tenedor 8
El Filósofo 8 ya NO es un comensal y se retira de la mesa.
El Filósofo 8 está pensando.

El filósofo 9 pensó durante 143 milisegundos.
El Filósofo 9 es el comensal 3
El Filósofo 9 toma el tenedor derecho:9

El filósofo 3 comió durante 1362 milisegundos.
El Filósofo 3 suelta el tenedor 4
El Filósofo 3 suelta el tenedor 3
El Filósofo 4 toma el tenedor derecho:4
El Filósofo 4 toma el tenedor izquierdo: 5
El Filósofo 4 está comiendo.

El filósofo 2 esperó 1468 milisegundos por el tenedor izquierdo.

El Filósofo 2 tendrá que soltar el tenedor 2 por exceso de tiempo y salir a pensar.
El Filósofo 3 ya NO es un comensal y se retira de la mesa.
El Filósofo 3 está pensando.
El Filósofo 2 suelta el tenedor 2
El Filósofo 2 ya NO es un comensal y se retira de la mesa.
El Filósofo 2 está pensando.
```

Figura 45: *Primera parte del tercer segundo de ejecución del programa.*

Continuando con el tercer segundo de ejecución, 5 vuelve a acceder a la mesa, 6 come un total de 850ms y se retira de la mesa; 7 aprovecha y comienza a comer. 1 vuelve a solicitar lugar en la mesa, 10 termina de comer en 666ms y 1 aprovecha para comenzar a comer. 9 tendrá que soltar el tenedor que tomó y se verá obligado a salir de la mesa. Hasta este punto, **1, 3, 5, 6, 7, 9, y 10** han consumido sus espaguetis con éxito al menos 1 vez.

```
El filósofo 5 pensó durante 801 milisegundos.
El Filósofo 5 es el comensal 4

El filósofo 6 comió durante 850 milisegundos.
El Filósofo 6 suelta el tenedor 7
El Filósofo 7 toma el tenedor derecho:7
El Filósofo 7 toma el tenedor izquierdo: 8
El Filósofo 7 está comiendo.
El Filósofo 6 suelta el tenedor 6
El Filósofo 6 ya NO es un comensal y se retira de la mesa.
El Filósofo 6 está pensando.

El filósofo 1 pensó durante 599 milisegundos.
El Filósofo 1 es el comensal 4

El filósofo 10 comió durante 666 milisegundos.
El Filósofo 10 suelta el tenedor 1
El Filósofo 10 suelta el tenedor 10
El Filósofo 1 toma el tenedor derecho:1
El Filósofo 1 toma el tenedor izquierdo: 2

El filósofo 9 esperó 918 milisegundos por el tenedor izquierdo.

El Filósofo 10 ya NO es un comensal y se retira de la mesa.
El Filósofo 9 tendrá que soltar el tenedor 9 por exceso de tiempo y salir a pensar.
El Filósofo 1 está comiendo.
El Filósofo 9 suelta el tenedor 9
El Filósofo 10 está pensando.
El Filósofo 9 ya NO es un comensal y se retira de la mesa.
El Filósofo 9 está pensando.

***Ha transcurrido 1 segundo de ejecución!***
```

Figura 46: Segunda parte del tercer segundo de ejecución del programa.

En este punto (el cuarto segundo), 6, 3, 8, 10 toman un lugar en la mesa. 7 termina de comer en 742ms y se retira de la mesa. 6 es obligado a retirarse de la mesa y 8 comienza a comer su plato. A su vez, 2 regresa a la mesa después de pensar 974ms y 9 de igual forma.

La segundo parte del cuarto segundo tiene al filósofo 4 terminando de consumir su plato y 5 volviendo a comer. 3 se levantará de la mesa por exceso de tiempo esperando un tenedor izquierdo y se levantará. Luego 4 y 3 regresan a la mesa intentando volver a comer (no tienen llenadera...) y 1 termina de consumir su plato y se va de la mesa a pensar. 10 abandonará la mesa por exceso de tiempo esperando por el tenedor izquierdo y 2 tomará un tenedor derecho.

Por último, el filósofo 8 habrá terminado de consumir su plato y se retirará de la mesa; situación que 9 aprovechará para volver a comer otro plato de espaguetis. Igualmente, 7 otra vez podrá tener acceso a repetir otro plato más. Una vez que este cuarto segundo ha concluido, los filósofos **1, 3, 4, 5, 6, 7, 8, 9, y 10**. 2 no ha tenido tanta suerte hasta este punto.

```
El filósofo 6 pensó durante 450 milisegundos.
El Filósofo 6 es el comensal 5
El Filósofo 6 toma el tenedor derecho:6

El filósofo 3 pensó durante 675 milisegundos.
El Filósofo 3 es el comensal 4
El Filósofo 3 toma el tenedor derecho:3

El filósofo 8 pensó durante 1004 milisegundos.
El Filósofo 8 es el comensal 3

El filósofo 10 pensó durante 420 milisegundos.
El Filósofo 10 es el comensal 2
El Filósofo 10 toma el tenedor derecho:10

El filósofo 7 comió durante 742 milisegundos.
El Filósofo 7 suelta el tenedor 8
El Filósofo 7 suelta el tenedor 7
El Filósofo 8 toma el tenedor derecho:8
El Filósofo 7 ya NO es un comensal y se retira de la mesa.
El Filósofo 7 está pensando.

El filósofo 6 esperó 1106 milisegundos por el tenedor izquierdo.

El Filósofo 8 toma el tenedor izquierdo: 9
El Filósofo 8 está comiendo.
El Filósofo 6 tendrá que soltar el tenedor 6 por exceso de tiempo y salir a pensar.
El Filósofo 6 suelta el tenedor 6
El Filósofo 6 ya NO es un comensal y se retira de la mesa.
El Filósofo 6 está pensando.

El filósofo 2 pensó durante 974 milisegundos.
El Filósofo 2 es el comensal 3

El filósofo 9 pensó durante 582 milisegundos.
El Filósofo 9 es el comensal 2
```

Figura 47: *Primera parte del cuarto segundo de ejecución del programa.*

Como se pudo observar, la mayoría de los filósofos ha podido acceder a su platilla al menos en una ocasión. Sin embargo, 2 no ha tenido tanta suerte como el resto. ¿Habrá caído en *starvation*? Para esto, se han verificado los siguientes 6 segundos de ejecución para comprobar que 2 no haya *muerto* de hambre. Para la gran fortuna del filósofo 2, finalmente tuvo oportunidad de poder consumir sus preciados espaguetis en el segundo 10; un número relativamente grande pero que nos permite comprobar que no cayó en inanición ni quedó interbloqueado con alguno de los filósofos contiguos al mismo.

```
El filósofo 2 comió durante 780 milisegundos.
El Filósofo 2 suelta el tenedor 3
El Filósofo 2 suelta el tenedor 2
El Filósofo 3 toma el tenedor derecho:3
El Filósofo 2 ya NO es un comensal y se retira de la mesa.
```

Figura 48: *Filósofo 2 alimentándose después de un largo rato.*



```
El filósofo 4 comió durante 1160 milisegundos.
El Filósofo 4 suelta el tenedor 5
El Filósofo 4 suelta el tenedor 4
El Filósofo 5 toma el tenedor derecho:5
El Filósofo 5 toma el tenedor izquierdo: 6
El Filósofo 5 está comiendo.

El filósofo 3 esperó 601 milisegundos por el tenedor izquierdo.

El Filósofo 4 ya NO es un comensal y se retira de la mesa.
El Filósofo 4 está pensando.
El Filósofo 3 tendrá que soltar el tenedor 3 por exceso de tiempo y salir a pensar.
El Filósofo 3 suelta el tenedor 3
El Filósofo 3 ya NO es un comensal y se retira de la mesa.
El Filósofo 3 está pensando.

El filósofo 4 pensó durante 244 milisegundos.
El Filósofo 4 es el comensal 3
El Filósofo 4 toma el tenedor derecho:4

El filósofo 3 pensó durante 313 milisegundos.
El Filósofo 3 es el comensal 2
El Filósofo 3 toma el tenedor derecho:3

El filósofo 1 comió durante 1052 milisegundos.
El Filósofo 1 suelta el tenedor 2
El Filósofo 1 suelta el tenedor 1

El filósofo 10 esperó 1289 milisegundos por el tenedor izquierdo.

El Filósofo 10 tendrá que soltar el tenedor 10 por exceso de tiempo y salir a pensar.
El Filósofo 10 suelta el tenedor 10
El Filósofo 10 ya NO es un comensal y se retira de la mesa.
El Filósofo 2 toma el tenedor derecho:2
El Filósofo 10 está pensando.
El Filósofo 1 ya NO es un comensal y se retira de la mesa.
El Filósofo 1 está pensando.

El filósofo 8 comió durante 638 milisegundos.
El Filósofo 8 suelta el tenedor 9
El Filósofo 9 toma el tenedor derecho:9
El Filósofo 8 suelta el tenedor 8
El Filósofo 8 ya NO es un comensal y se retira de la mesa.
El filósofo 8 comió durante 638 milisegundos.
El Filósofo 8 suelta el tenedor 9
El Filósofo 9 toma el tenedor derecho:9
El Filósofo 8 suelta el tenedor 8
El Filósofo 8 ya NO es un comensal y se retira de la mesa.
El Filósofo 8 está pensando.
El Filósofo 9 toma el tenedor izquierdo: 10
El Filósofo 9 está comiendo.

El filósofo 7 pensó durante 646 milisegundos.
El Filósofo 7 es el comensal 4
El Filósofo 7 toma el tenedor derecho:7
El Filósofo 7 toma el tenedor izquierdo: 8
El Filósofo 7 está comiendo.

***Ha transcurrido 1 segundo de ejecución!***
```

Figura 49: Segunda parte del cuarto segundo de ejecución del programa.

A raíz de la ejecución presentada, podemos realizar las siguientes observaciones con respecto al desempeño del programa:

- Todos los filósofos pudieron alimentarse en, al menos, una ocasión. Lo que permite descartar que el programa presente un interbloqueo temprano o un caso de inanición en los filósofos.

- La *justicia* en esta implementación parece cumplirse *a secas*, ya que el filósofo 2 tuvo que esperar una cantidad considerable de tiempo (10 segundos, específicamente) para poder acceder a su plato en la mesa. Por otra parte, fue eficaz en el sentido de que en, diversas ocasiones, los filósofos que trataban de repetir plato fueron obligados a salir a pensar por exceso de tiempo de espera.
- El programa no presentó inestabilidad durante su ejecución. Para la prueba realizada el programa fue ejecutado un total de 4 minutos y 9 segundos sin presentar alguna situación anormal.

Posterior a la prueba de ejecución, se ha realizado un análisis estadístico de la cantidad de platos que cada filósofo consumió en 68 *seg.* de la ejecución completa del programa. El *log* de salida de la prueba presentada se encuentra anexo al documento escrito del presente proyecto.

Los resultados obtenidos indican que, con una relativa certeza, no se presentan interbloqueos o inaniciones a largo plazo. Así mismo, puede observarse una relativa *justicia*, ya que no se presenta algún dato anormal que sugiera que un filósofo se encuentra consumiendo muchos menos platos de lo que debería. Sin embargo, resulta apreciable que los filósofos **5, 7 y 10** son quienes mayor platos de espaguetis consumieron durante su tiempo de vida. Por otra parte, el filósofo **2** pudo consumir una mayor cantidad de platos y no se quedó relegado frente a los otros filósofos; como la prueba inicial sugería.

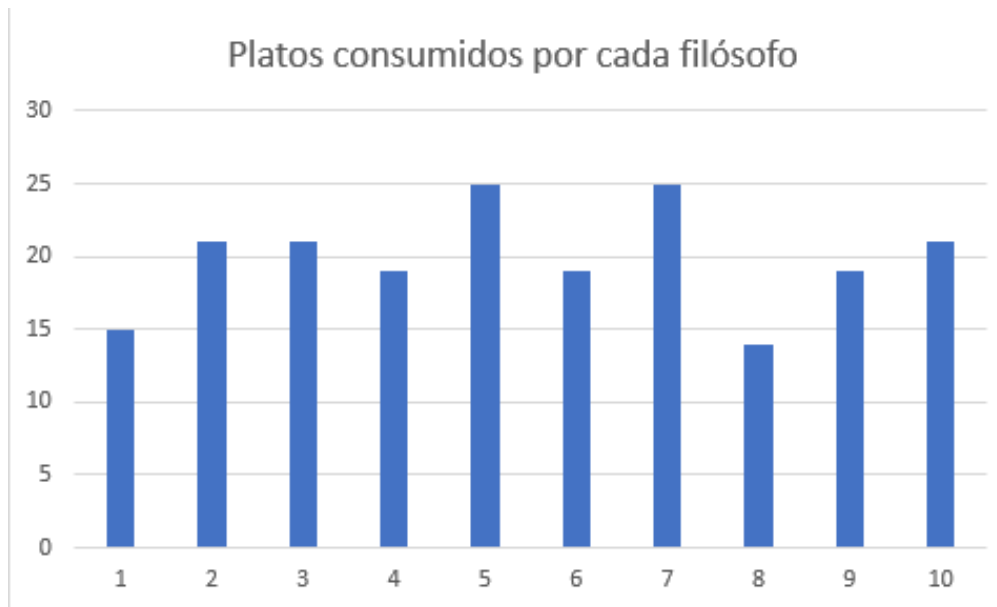


Figura 50: Gráfica de los platos consumidos por cada filósofo en 68 segundos.

A raíz de esto, es posible concluir que el programa presentado y la disciplina de comportamiento implementada cumplen con las especificaciones solicitadas por el problema y, por tanto, el problema es debidamente solucionado por medio de monitores de *sincronización* en **Java**.

## 7. Conclusiones

### ■ López Lara Marco Antonio

Gracias a este proyecto pude entender de una mejor manera la teoría del paralelismo y concurrencia así como un poco de su historia. Me parece importante destacar la importancia de la programación concurrente/paralela ya que este programa se realizó con ello, gracias a esta programación se logró que nuestro programa fuera rápido, se mostrara el tiempo en que cada filósofo tiene los dos palillos chinos o tenedores así como se muestra que cada filósofo tiene sus acciones individuales gracias a la sincronización.

La importancia de la historia de este problema es de gran importancia a mi parecer para el paralelismo y la programación concurrente ya que marca diversos temas de ellos para que la resolución de este problema sea correcta además de que en este problema existen diversas soluciones por la manera en que uno puede hacer que los tenedores pase por los filósofos, a mí pensar nuestro programa o solución esta aun nivel medio alto ya que en las diversas pruebas no se encontraron deadlocks, todos los filósofos en algún momento comieron, pensaron y tuvieron hambre (en todo momento se muestra el ciclo de actividades independientes de cada filósofo).

Pensar desde el inicio que un filósofo representa un hilo en ejecución y el tenedor/palillo representa un recurso compartido de uso exclusivo nos ayuda a buscar una solución correcta, la teoría que habíamos visto en la clase nos ayudo para poder agregarle más a nuestra solución así como ver y esperar la correcta sincronización necesaria para que dos filósofos no tuvieran el mismo tenedor o se causara una espera infinita de uno, por ello con la ayuda de semaphore se lograron cubrir ciertos aspectos importantes.

Por último puedo concluir gracias a todo lo anterior que el paralelismo es de gran importancia al momento de buscar una solución siempre teniendo en cuenta la importancia de ciertos aspectos teóricos, además de que el paralelismo hace que la ejecución de un programa sea más rápido, al usar paralelismo o hacer un programa concurrente será de gran ayuda pero si no se tiene en mente desde un principio el problema así como cosas específicas de el paralelismo y la concurrencia puede ser difícil o hasta perjudicial.

### ■ Olivera Martínez Jenyffer Brigitte

A lo largo del desarrollo de este proyecto fue posible ver distintos aspectos de la programación paralela/concurrente, es relevante que el paralelismo, es un paradigma distinto a la programación estructurada y la programación orientada a objetos, ello no significa que no pueda emplearse el paradigma orientado a objetos en el paradigma paralelo, son paradigmas que se complementan o por lo menos esto fue lo que pude observar con este proyecto.

Uno pensaría que el paralelismo/concurrencia están totalmente separados de la secuencialidad que

tiene la programación no paralela, sin embargo es otro ingrediente importante para implementar el paralelismo/concurrencia, pues no es posible hacer programas totalmente paralelos o concurrentes.

Visualizando este paradigma y sus virtudes en un contexto real, juega un papel de progreso y de desarrollo de software que con los años se ha podido implementar en distintas áreas, me doy cuenta de la exigencia de las situaciones actuales hacia el desarrollo computacional. Vivimos una época donde la tecnología avanza cada vez más en distintos campos y de manera muy acelerada.

Definitivamente las creaciones que se han desarrollado y los descubrimientos que se han hecho en el área de la computación son una señal del potencial humano y de su hambre de creación, que si bien, muchas veces el software está diseñado para satisfacer necesidades humanas, se prestan a otros objetivos que pueden afectar de manera no tan positiva o totalmente negativa a terceros. De ahí lo valioso que es tener un conocimiento amplio que nos permita desarrollar con el principal objetivo de beneficiar necesidades de manera conciente.

En el proyecto también se abordarán algunos problemas, que si bien tienen forma de solucionarse, son de gran importancia estudiarlas, pues si no se conocen los problemas, no se pueden implementar soluciones y por ende no se puede seguir desarrollando nuevas implementaciones y mejoras.

Una vez más se demuestra que el trabajo en equipo rinde frutos más grandes que el trabajo individual en esta área de desarrollo de proyectos, pues cada integrante tiene distintas virtudes que en conjunto dan resultados realmente satisfactorios, lo que recalca la importancia de formar sociedades y el valor del papel que jugamos en ellas, además del impacto que tiene el trabajo en conjunto.

Para terminar, doy reconocimiento a el proceso de creación de proyectos, en nuestra formación como ingenieros y el valor que agrega a los proyectos formarnos también como personas, ser capaces de asumir responsabilidades y contribuir con lo mejor que tenemos a este tipo de actividades tanto como estudiantes, como individuos y en un futuro como profesionales.

#### ■ Téllez González Jorge Luis

El trabajo desarrollado junto con mis compañeros me ha traído grandes aprendizajes en este primer acercamiento al paradigma de la programación paralela, y de forma más específica, a la programación concurrente. El proyecto que presentamos tuvo como principal enfoque mostrar los problemas que pueden surgir a raíz del uso de *concurrencia* y su relación que estos pueden tener en la programación de algoritmos *paralelos*. Resulta importante mencionar los siguientes aspectos:

1. El estudio del paralelismo requiere a su vez de la concurrencia, ya que ambos campos se encuentran en profunda relación y su estudio requiere tener conocimiento del otro.
2. Los problemas clásicos de la concurrencia pueden impactar severamente en el diseño de un algoritmo paralelo si no son debidamente tratados o estudiados.

3. La sincronización puede resultar un método efectivo para evitar errores asociados a la concurrencia en secciones específicas de un algoritmo paralelo.
4. Problemas como el abordado en el presente proyecto, u otros como el problema del *productor-consumidor*, tienen un papel fundamental en el diseño de los sistemas operativos actuales.
5. El problema de los filósofos ilustra los posibles conflictos que pueden presentarse en el diseño tanto de algoritmos paralelos como de sistemas multiprogramados que aprovechen un mayor número de CPU para potenciar su desempeño operacional.

La experiencia obtenida ha resultado ser muy importante para iniciar un estudio de mayor profundidad de la programación *concurrente* y de las implicaciones directas que puede tener en el *paralelismo*; recordando que su principal objetivo es acelerar la ejecución de programas secuenciales o aprovechar entornos multinúcleo para realizar sus operaciones atómicas con efectividad. Una vez más, y con base en la investigación, es afirmable que un estudio correcto del paralelismo requiere necesariamente un estudio de la concurrencia.

La implementación expuesta por el equipo logró cumplir sus objetivos propuestos, sin la aparición de los problemas expuestos en la investigación teórica. Esta implementación estuvo basada en una aplicación gráfica, la cual fue modificada para ser ejecutada únicamente en consola y se hicieron diversas modificaciones adicionales para adaptarse a los objetivos principales del proyecto. Los resultados fueron satisfactorios y nos han enseñado grandes lecciones respecto a la sincronización de procesos en **Java**.

Finalmente, considero que el objetivo del proyecto ha sido cumplido con total éxito; ya que incluso fuimos más allá de implementar un algoritmo paralelo y nos adentramos a las cuestiones teóricas de diseño que pueden impactar en todos ellos. Incluso, este primer acercamiento nos ha brindado bases iniciales para introducirnos al estudio de la programación de sistemas operativos, lo cual es completamente invaluable para nuestro desarrollo profesional como Ingenieros en Computación.

## Referencias

- [1] Chapter 6 Concurrency: Deadlock and Starvation. Recuperado de: <https://www.unf.edu/public/cop4610/ree/Notes/PPT/PPT8E/CH%2006%20-0S8e.pdf>. Fecha de consulta: 16/05/2020.
- [2] Condiciones de Carrera. Recuperado de: [http://babel.upm.es/teaching/concurrencia/material/slides/groman/CC\\_CondCarrera.pdf](http://babel.upm.es/teaching/concurrencia/material/slides/groman/CC_CondCarrera.pdf). Fecha de consulta: 16/05/2020.
- [3] Exclusión mutua. Recuperado de: <https://webprogramacion.com/44/sistemas-operativos/exclusion-mutua.aspx>. Fecha de consulta: 16/05/2020.
- [4] Java Thread Starvation. Recuperado de: <https://avalides.com/java-thread-starvation-livelock-with-examples/>. Fecha de consulta: 16/05/2020.



- [5] La Cena de los Filósofos. Recuperado de: <https://pacoportillo.es/informatica-avanzada/programacion-multiproceso/la-cena-de-los-filosofos/>. Fecha de consulta: 16/05/2020.
- [6] Monitors – The Basic Idea of Java Synchronization. Recuperado de: <https://www.programcreek.com/2011/12/monitors-java-synchronization-mechanism/>. Fecha de consulta: 16/05/2020.
- [7] Mutual Exclusion in Synchronization. Recuperado de: <https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/>. Fecha de consulta: 16/05/2020.
- [8] Programación concurrente. Recuperado de: [http://ferestrepoca.github.io/paradigmas-de-programacion/progconcurrente/concurrente\\_teoría/index.html](http://ferestrepoca.github.io/paradigmas-de-programacion/progconcurrente/concurrente_teoría/index.html). Fecha de consulta: 16/05/2020.
- [9] Starvation and Fairness. Recuperado de: <http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html>. Fecha de consulta: 16/05/2020.
- [10] Therac-25. Recuperado de: <https://ethicsunwrapped.utexas.edu/case-study/therac-25>. Fecha de consulta: 16/05/2020.
- [11] Breshears, C. (2009). *The Art of Concurrency*. O'Reilly Media, 1st edition.
- [12] Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 1st edition.

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