



POLITÉCNICA



UNIVERSIDAD POLITÉCNICA DE MADRID
ETS DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN
DPTO. DE INGENIERÍA TELEMÁTICA Y ELECTRÓNICA

Sistemas Operativos

Curso 2017/2018

Práctica 3

Hilos y concurrencia

Versión r3, 5/dic/2017

Tabla de contenido

1	Objetivos.....	3
2	Presentación	3
3	Evaluación.....	4
4	Enunciado	5
5	Desarrollo de la práctica	7
5.1	Fase 1.....	7
	Entrenamientos fase 1.....	7
	Caso práctico fase 1	10
5.2	Fase 2.....	12
	Entrenamientos fase 2.....	12
	Caso práctico fase 2	15
5.3	Fase 3.....	19
	Entrenamientos fase 3.....	19
	Caso práctico fase 3	22
5.4	Fase 4.....	24
	Entrenamiento fase 4	24
	Caso práctico fase 4	25

1 Objetivos

- Desarrollar una aplicación multihilo, utilizando para ello las herramientas del entorno de programación Java.
- Utilizar las herramientas de comunicación y sincronización de hilos que ofrece el entorno de programación Java: cerrojos, semáforos, colas, etc.



Se recomienda el uso del entorno de desarrollo Eclipse.

2 Presentación

Para la realización de esta práctica se han programado cuatro sesiones de laboratorio. **No será necesario realizar ninguna entrega de código, puesto que el código desarrollado no será evaluado.** Sin embargo, es muy importante realizar la práctica y conocer todos los detalles de su codificación para poder tener éxito en la prueba de evaluación.

Para facilitar su desarrollo, la práctica está dividida en **4 fases**. Cada fase está dividida en dos apartados:

- **Entrenamiento:** Introduce uno o varios conceptos relacionados con los problemas de concurrencia y propone unos ejercicios o programas para solucionarlos usando las herramientas de concurrencia disponibles en Java.
- **Caso práctico.** Propone el desarrollo de elementos concretos del sistema.

A lo largo de la práctica se va a codificar en Java un sistema cuyas características generales, a modo de **avance**, se resumen en:

El proyecto de búsqueda de vida extraterrestre (Seti) usa radiotelescopios para localizar señales de origen no natural. Como este análisis es un proceso intensivo en cálculo, se recurre a voluntarios externos que colaboran con este proyecto. Un voluntario es una persona que pone a disposición del proyecto Seti un recurso informático para que haga las operaciones de cálculo. De esta forma el proyecto Seti realiza una computación distribuida.¹

Esta práctica pretende implementar una simulación sencilla del proyecto Seti. Para ello, se dispone un sistema que trocea la información que entregan los radiotelescopios en un conjunto de tareas. Cada tarea se envía a través de Internet al ordenador de un voluntario para su proceso. Cada voluntario obtiene, a su ritmo, su resultado, y lo remite al sistema. Cada tarea se envía a varios voluntarios distintos y, como algún voluntario puede generar resultados erróneos, se realiza una validación comparando los resultados que devuelven todos voluntarios: si son idénticos, la tarea se considera realizada, y si alguno fuera diferente, se descartan los resultados y se remite la misma tarea a nuevos voluntarios.



La documentación necesaria está disponible en la unidad L: de la asignatura.



Para realizar esta práctica no necesita usar la máquina virtual FreeBSD. Puede desarrollarla en cualquier ordenador que tenga instalado el entorno de desarrollo Eclipse.

¹ Información del proyecto Seti: <http://setiathome.berkeley.edu/>

! Realice una fase cada semana. Cada semana solo necesita revisar el apartado Enunciado común a las 4 fases, y el enunciado de esa fase.

! Los entrenamientos están diseñados para aislar y experimentar por separado un aspecto o concepto concreto que se necesitará en la práctica. La mayoría de su código podrá incorporarlo después a la práctica, ya con la ventaja de conocer en detalle cómo funciona.

! Tras realizar los entrenamientos, deberá aplicar esos conocimientos a un caso práctico. En aspectos de concurrencia se requieren pocas líneas de código pero es crítico el lugar y orden en que se aplican, por lo que la dificultad de esta práctica no reside en la cantidad de código sino en su calidad. Por ello, este proceso de aplicación de conceptos simples ya experimentados en los entrenamientos a un caso práctico más amplio es muy importante. Razone siempre por sí mismo por qué cada línea va en su lugar y no en otro; procure no consultar soluciones, porque una vez puesta la línea en el lugar adecuado, parece obvio, pero no es así antes de hacerlo. De lo contrario, no le servirá para su aprendizaje y, en la próxima ocasión probablemente tendrá dudas similares.

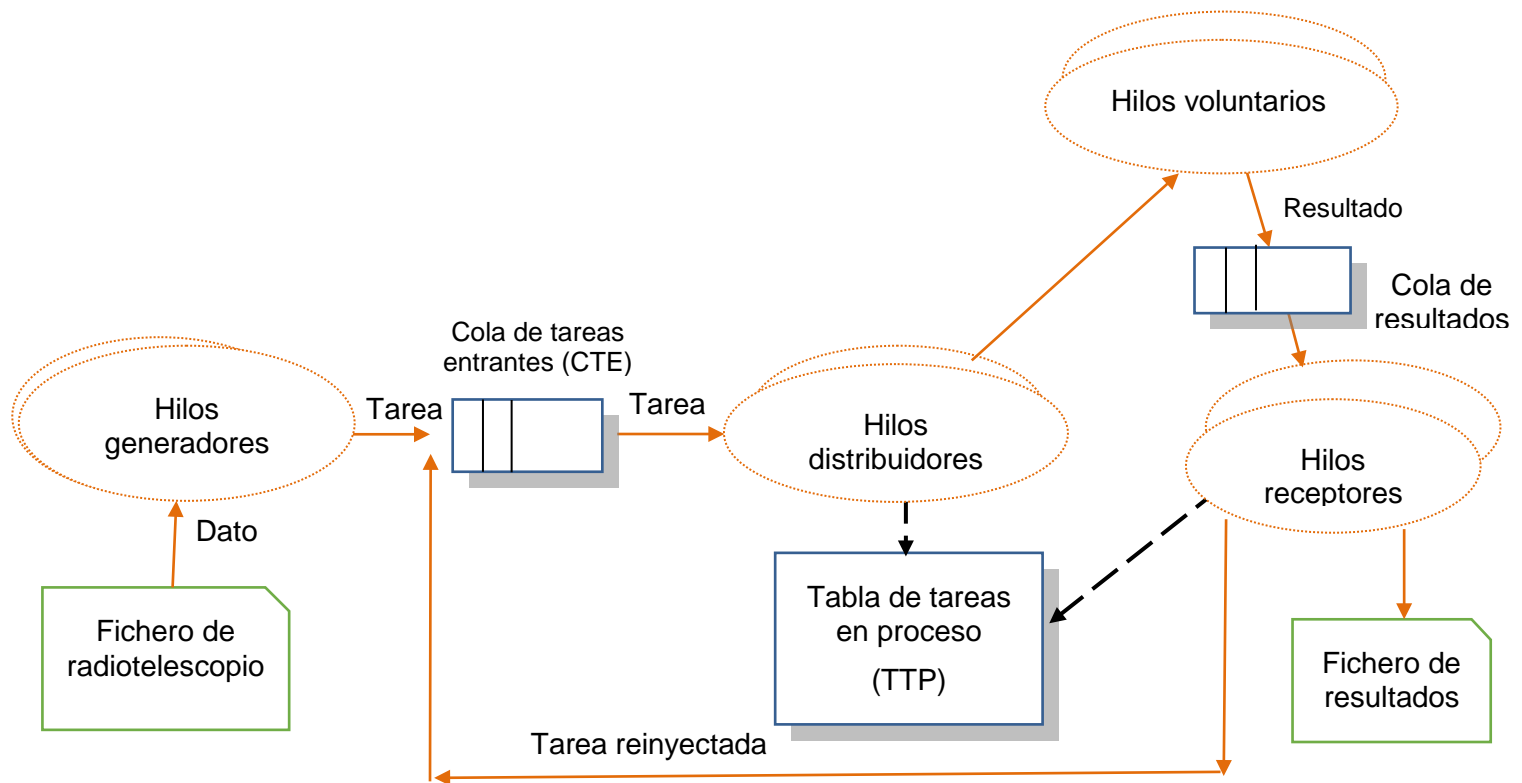
3 Evaluación

Esta práctica **será evaluada únicamente mediante un cuestionario** que se llevará a cabo el día que se realice la evaluación del Bloque II de la asignatura, previsto para el **18 de diciembre**.

El cuestionario de evaluación estará orientado a conocer el grado de dominio tanto de los conceptos generales manejados en la práctica como de las cuestiones relacionadas con su programación. Para ello se mezclarán preguntas conceptuales con preguntas específicas del código necesario para realizar la práctica o con preguntas relacionadas con estrategias de cambio ante alguna modificación que pueda plantearse. Con el objetivo de homogeneizar soluciones, en cada una de las fases se presentan algunos criterios de diseño de obligatorio cumplimiento para la elaboración de la práctica.

4 Enunciado

Se va a codificar una aplicación multihilo que simula el comportamiento del proyecto Seti comentado en el apartado de [presentación](#). El esquema propuesto es el siguiente:



Descripción resumida de los elementos que componen este esquema (consulte la documentación de la fase porque no hace falta comprender en detalle todo este esquema desde un inicio):

- **Fichero de radiotelescopio.** Contiene la información generada por los radiotelescopios que el sistema va a analizar². En esta práctica, el fichero contiene líneas de texto y a cada línea se las referenciará como *Dato*.
- **Hilos generadores.** Producen *tareas*. Para ello, iterativamente, leen una línea del *fichero de radiotelescopio* para construir una tarea y la depositan en la *cola de tareas entrantes*.
- **Tarea.** Agrupa varios campos relacionados con cada tarea: al menos la línea a procesar (el *Dato*) y un *identificador único de tarea* que permita distinguirla de otras tareas.
- **Cola de tareas entrantes (CTE).** Almacena las *tareas* producidas hasta que pueden ser procesadas (*consumidas*) por el sistema.

² En el SETI real, esta información es binaria y procede del muestreo digitalizado de ciertas bandas de frecuencia captadas por conjuntos de radiotelescopios. Como en esta práctica, esa información se trocea y se envía a los ordenadores de voluntarios. Cada voluntario realiza un cálculo, basado en múltiples transformadas rápidas de Fourier, que busca, en resumen, localizar picos de señal o señales distinguibles del ruido.

- **Hilos distribuidores.** Distribuyen cada *tarea* a 3 voluntarios. Iterativamente, extraen una *tarea* de la CTE, crean una *tarea en proceso* y la insertan en una *tabla de tareas en proceso*, y crean e inician 3 hilos voluntarios a los que proporcionan esa tarea.
- **Tarea en proceso.** *Tarea* enriquecida con otros atributos que se consideren necesarios para conocer el estado de proceso de la tarea.
- **Tabla de tareas en proceso (TTP).** Almacena las *tareas en proceso* mientras se procesan.
- **Hilos voluntarios.** Simulan un voluntario³. Reciben una *tarea* y calculan un *resultado*, que depositan en la *cola de resultados*.
- **Resultado.** Contiene, al menos, el valor calculado por un voluntario y el *identificador* de la tarea a la que corresponde.
- **Cola de resultados.** Almacena los resultados generados por los voluntarios hasta que puedan ser procesados (*consumidos*) por los hilos receptores.
- **Hilos receptores.** Deciden si la tarea ha conseguido finalizar con éxito, o bien se ha producido un error de cálculo en algún voluntario. Para ello, procesan los *resultados* calculados por los voluntarios: si los 3 *resultados* son idénticos, la tarea ha finalizado con éxito, y así se refleja en el *fichero de resultados*; si algún *resultado* es diferente, esta tarea ha fallado, así que se debe reintentar obtener un resultado correcto creando una nueva *tarea* (que contenga la misma línea que la que ha fallado), y depositándola en la CTE.
- **Fichero de resultados.** Almacena la información de cada tarea finalizada con éxito.

En el siguiente apartado se detalla el comportamiento concreto de cada uno de los hilos, enunciando los problemas de concurrencia que tiene esta aplicación multihilo y que deben resolverse satisfactoriamente.

³ En el Seti real los distribuidores se comunicarían con los voluntarios a través de una red, pero en esta práctica, por simplicidad, reciben la tarea directamente y depositan el resultado en una cola.

5 Desarrollo de la práctica

5.1 Fase 1

Entrenamientos fase 1



Clase `java.lang.Thread`

F1E1 Aplicación multihilo sin compartición

- Haga una clase java con un programa principal (método main) que cree un número de hilos determinado por una constante. A cada hilo se le debe pasar como parámetro un número entero consecutivo. Los hilos se deben crear con el nombre “Hilo-N”, donde N coincide con el número que se pasa como parámetro. Una vez creados los hilos, debe esperar a que todos ellos terminen. Utilice un array de objetos `Thread` para almacenar las referencias de cada hilo. De esta forma podrá hacer un `join()` de todos los objetos de ese array para esperar a que los hilos finalicen.
- Escriba una clase que implemente los hilos. Cada hilo hace lo siguiente:
 - a. Recibe, en su constructor, un número entero N que identifique al hilo.
 - b. Muestra por pantalla su nombre y su número.
 - c. Instancia un objeto `ArrayList` (como atributo privado del hilo).
 - d. Iterando, realiza 10.000 veces lo siguiente: instancia un nuevo objeto cadena de caracteres (puede contener el mismo texto que los anteriores) y lo añade al objeto `ArrayList`.
 - e. Consulta al objeto `ArrayList` la cantidad de elementos que contiene, y la muestra en pantalla con el formato: “[Hilo-N] E elementos.”



Pruebe con 8 hilos. Deberían aparecer 8 mensajes “Hilo-1: 10000 elementos”, “Hilo-2: 10000 elementos”, etc. ¿Es así? Observe los resultados y, si detecta algún error, corríjalo.

F1E2 Aplicación multihilo con compartición

A partir de lo realizado en F1E1, haga estas modificaciones para que los hilos compartan el objeto `ArrayList`:

- **La clase principal, donde se encuentra el método main(), hace lo siguiente:**
 - a. Instancia un objeto `ArrayList`.
 - b. Comparte la referencia a este objeto con todos los hilos, proporcionándosela a cada hilo como parámetro al instanciarlo.
 - c. Espera a que finalicen todos los hilos.
 - d. Consulta al objeto `ArrayList` cuántos elementos contiene, y muestra ese valor en pantalla: “[Hilo-principal] El ArrayList tiene N elementos”.
- **La clase que implementa los hilos hace lo siguiente:**

- a. Recibe, en su constructor, además del número que identifica al hilo, la referencia del objeto compartido `ArrayList`
- b. Muestra por pantalla su nombre y su número.
- c. Realiza 10.000 iteraciones, de forma que en cada iteración instancia un nuevo objeto cadena de caracteres diferente (puede contener el mismo texto que otros) y lo añade al objeto compartido `ArrayList`.



Pruebe con 8 hilos: debería ver un mensaje “Hilo-principal: El `ArrayList` tiene 80000 elementos”, pero puede no ser así. También es posible que vea mensajes de excepciones durante la ejecución de la aplicación. **Deduzca la causa de estos problemas**, pero NO intente repararlos.

F1E3 Aplicación productor/consumidor comunicados mediante una cola bloqueante

En este entrenamiento se van a crear hilos *productores* e hilos *consumidores* que se comunican mediante un objeto *cola* que implementa la interfaz `java.util.concurrent.BlockingQueue<E>`, como los de la clase `java.util.concurrent.ArrayBlockingQueue`.



Interfaz `java.util.concurrent.BlockingQueue<E>`. Encontrará código ejemplo en las diapositivas de la asignatura y también en:
<http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>

Cree las siguientes clases de acuerdo con esta especificación:

- **En la clase principal, el método `main()`:**
 - a. Instancia un objeto cola (objeto `ArrayBlockingQueue`) que admita un máximo de 3 elementos.
 - b. Crea dos hilos consumidores y dos hilos productores, compartiendo la referencia a la cola con todos los hilos.



Los hilos se deben crear con el nombre “Productor-N” o “Consumidor-N”, donde N será el número que se pasa como parámetro.


- c. En cada iteración de un bucle sin fin, pregunta el número actual de elementos a la cola, lo imprime en pantalla y realiza una espera pasiva de un segundo.




Para programar que este hilo realice una espera pasiva durante cierto tiempo:
`Thread.currentThread().sleep(milisegundos);`

- **Hilo productor:**
 - a. Recibe, en su constructor, la referencia a la cola compartida.
 - b. Muestra por pantalla su nombre.

- c. Iterativamente, produce 10 mensajes (cadenas de caracteres), con el formato “HiloProductor-N: mensaje M”, siendo N el número de hilo (1 o 2) y M el número de mensaje (1...10), y los va añadiendo a la cola.
- **Hilo consumidor:**
 - a. Recibe, en su constructor, la cola compartida.
 - b. Muestra por pantalla su nombre.
 - c. En un bucle sin fin, va extrayendo mensajes de la cola y los va mostrando en pantalla junto con el nombre del hilo.

 Observe que la cola dispone de operaciones `put/take` (bloqueantes si la cola está llena/vacía, respectivamente) y `offer/poll` (no bloqueantes). Pruebe estos juegos de operaciones y observe sus diferencias.

 Observe si los mensajes que aparecen en pantalla son los que esperaba, tanto en su cantidad, como en su orden.

F1E4 Lectura escritura de ficheros


Parta del entrenamiento F1E3 con los siguientes cambios:

- **Clase principal:**
 - a. Instancia un objeto `FileReader` para leer un fichero de texto cuyo nombre debe estar escrito en una constante.
 - b. Instancia un objeto `BufferedReader` para poder leer las líneas del fichero.
 - c. Crea un hilo productor, compartiendo, además de la referencia a la cola (como ya se hizo en F1E3), el objeto `BufferedReader`.
 - d. Crea dos hilos consumidores compartiendo la cola.
 - e. Permanece en un bucle sin fin mostrando por pantalla, cada segundo, el número de elementos que hay en la cola.
- **Hilo productor:** La única diferencia respecto a F1E3 es que ahora los mensajes que mete en la cola son las líneas que lee del fichero de entrada.
- **Hilo consumidor:** La única diferencia respecto a F1E3 es que ahora los mensajes que saca de la cola los debe escribir en un fichero de texto. Cada hilo usa un fichero diferente, cuyo nombre es el nombre del hilo con la extensión `.txt`.

 Lectura y escritura de ficheros de texto. Use las siguientes clases de `java.io`:

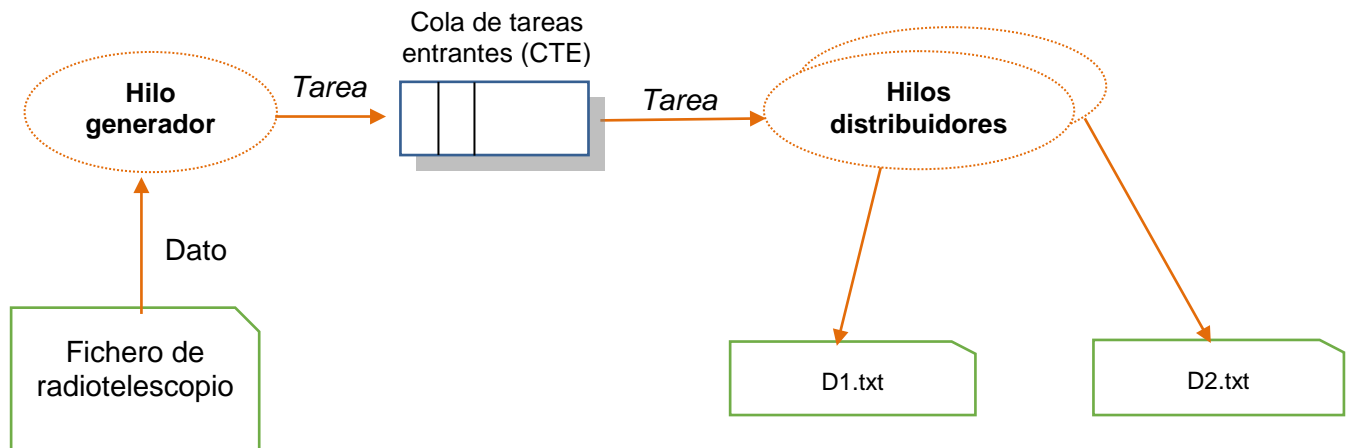
- Para leer: `FileReader` y `BufferedReader`

- Para escribir: `FileOutputStream`, `BufferedOutputStream` y `PrintWriter`

 Podría tardar en volcarse el contenido en los ficheros de texto (hasta que los búferes de salida se llenen). Para observar un refresco inmediato de cada escritura en el fichero, recuerde usar el método `flush()`.

Caso práctico fase 1

La arquitectura del programa, en esta fase, es la siguiente:



El funcionamiento de la aplicación sería:

- El **programa principal** crea un hilo generador y dos hilos distribuidores, tras lo cual permanece en un bucle sin fin mostrando, por pantalla y de forma periódica cada segundo, el número de *tareas* que hay en la *cola de tareas entrantes*.
- El hilo **generador** produce *tareas* y las deposita en la *cola de tareas entrantes*. Cada *tarea* se identifica con un número único (diferente para cada tarea y consecutivo) y contiene una línea que se ha leído del fichero de radiotelescopio. Este hilo permanecerá en un bucle, mientras existan líneas en el fichero.
- Hilos **distribuidores**. Cada hilo, en cada iteración de un bucle sin fin, extrae un objeto *tarea* de la *cola de tareas entrantes* y lo procesa. En esta fase el procesamiento de una tarea consiste tan solo en escribir la información de la tarea al final de un fichero. Cada hilo distribuidor escribe en un fichero diferente, cuyo nombre se construye concatenando la extensión “.txt” al nombre del hilo. La información de la tarea la constituyen su *identificador de tarea* y el texto procedente del fichero de radiotelescopio, y se escribe con formato “*identificador de tarea : dato de la tarea*”, usando fin de línea como separador.



En la unidad L: dispone de un ejemplo de aplicación JAVA que lee un fichero de texto, línea a línea, y copia su contenido en un fichero destino.



Implemente el código de forma que se cree un cierto número (constante) de hilos generadores y distribuidores (y, en este caso, defina las constantes de forma que se creen 1 generador y 2 distribuidores).




Al crear los hilos se recomienda asignarles un nombre, pues esto facilitará la depuración. Por ejemplo, el nombre del hilo generador puede ser “G1” y el de los distribuidores “D1” y “D2”




Para facilitar las pruebas, use como origen un fichero con al menos 10 líneas de texto.



El nombre del *fichero de radiotelescopio* debe estar definido en una constante en la clase del programa principal.

 **DEBE** usar un objeto de la clase `java.util.concurrent.ArrayBlockingQueue<E>` para implementar la **cola de tareas entrantes**. No se admiten otras posibles soluciones.

 Consideraciones referentes a **secciones críticas** que pueden aparecer en esta fase 1. Debe valorar y aportar, en su caso, una solución que garantice la consistencia en la información:

- 1) Analice la existencia o no de una sección crítica en la *cola de tareas entrantes* si acceden de forma concurrente el hilo generador y los distribuidores.
 - 2) Analice la existencia o no de una sección crítica en la *cola de tareas entrantes* si acceden de forma concurrente hilo principal y generador, o hilo principal y algún distribuidor.
-


5.2 Fase 2


Entrenamientos fase 2

F2E1 Región crítica implementada con cerrojos

A partir de lo realizado en **F1E4**, realice la siguiente modificación:

- **Clase principal:**
 - Instancia un objeto `PrintWriter` para poder escribir sobre un fichero llamado "FicheroDeResultados.txt".
 - Al crear los hilos consumidores les pasa como parámetro el objeto `PrintWriter` para que puedan compartirlo.
- **Hilos consumidores:** ambos hilos escribirán en el fichero "FicheroDeResultados.txt". La escritura debe implementarse en exclusión mutua mediante un cerrojo (*mutex*), de forma que, en cada instante, sólo uno de los hilos pueda estar escribiendo en el fichero.

 En java sirve de cerrojo cualquier objeto que implemente la interfaz `java.util.concurrent.locks.Lock`, como es el caso de los objetos de clase `java.util.concurrent.locks.ReentrantLock`.

 Observe que los hilos deben compartir (páselo como parámetro en el constructor) el **mismo** objeto cerrojo. Si cada hilo actuase sobre un cerrojo distinto no estaría implementando correctamente una Región Crítica.

F2E2 Región crítica implementada con `synchronized`

Repita el entrenamiento anterior (F2E1) pero en esta ocasión utilice `synchronized` (objeto) `{...}` en lugar de un cerrojo para implementar la región crítica en la escritura del fichero.

F2E3 Uso de un monitor.

A partir del ejercicio **F1E3** realice las siguientes modificaciones:

- Cree una clase, llamada **Contador**, con las siguientes características:
 - Tendrá un atributo privado llamado contador de tipo entero, inicializado a 0.
 - Tendrá los siguientes métodos públicos:
 - `void elementoIntroducido()`: Incrementa el atributo contador
 - `void elementoConsumido()`: Decrementa el atributo contador
 - `int cuantosElementos()`: Devuelve el valor del contador
- **Hilo productor:**

Cada vez que un hilo *productor* ponga elementos en la cola debe invocar al método `elementoIntroducido()` de la clase Contador.
- **Hilo consumidor:**

- a. Cada vez que un hilo *consumidor* extraiga elementos de la cola debe invocar al método `elementoConsumido()` de la clase `Contador`.
- b. Invocar al método `cuantosElementos()` de la clase `Contador` para mostrar por pantalla el valor del contador.

Observe el funcionamiento del programa. ¿Es el esperado? Puesto que varios hilos simultáneamente están manejando un objeto compartido, deberían producirse problemas de concurrencia y el valor del contador podría no ser el esperado. Para solucionar estos problemas de concurrencia la clase `Contador` debería ser implementada como `Monitor`. Realice las modificaciones necesarias y observe que se han solucionado los problemas de concurrencia.

F2E4 Uso de un mapa

A partir del ejercicio **F1E3** realice las siguientes modificaciones:

- Cree una clase nueva llamada `Mensaje` que contenga: un atributo público entero, llamado `identificador`, y un atributo público de tipo `String`, llamado `texto`.
- Cree una nueva clase `Paquete` que contenga: un atributo público entero llamado `identificador` y dos atributos públicos de tipo `String`, llamados `texto1` y `texto2`.
- Modifique la clase principal para que cree un único hilo *productor* y dos hilos *consumidores*. La clase principal deberá instanciar previamente un objeto de la clase `java.util.concurrent.ConcurrentHashMap` que compartan todos los hilos. Las claves del mapa serán enteros que se corresponderán con los identificadores de la nueva clase `Paquete`, y los valores serán objetos de la clase `Paquete`.
- El hilo productor generará 20 `String` diferentes en grupos de 2. Cada vez que genera un grupo hará lo siguiente:
 1. Generar un identificador, consecutivo al identificador anterior. Por ejemplo, empezar con el identificador 1, y los siguientes serían el 2, el 3, etc.
 2. Crear un objeto de la clase `Paquete`, donde almacenar el identificador (en el atributo `identificador`) y los dos `String` (en los atributos `texto1` y `texto2`).
 3. Introducir el objeto `Paquete` en el mapa.
 4. Generar dos objetos `Mensaje`, que tendrán el mismo identificador pero distinto texto.
 5. Introducir en la cola los dos objetos `Mensaje`.
- d. Cada uno de los hilos *consumidores* permanecerán en un bucle sin fin realizando esta tarea:
 1. Extraer un `Mensaje` de la cola.
 2. A partir del identificador del mensaje, buscar un `Paquete` en el mapa con el mismo identificador.
 3. Localizar el texto del `Mensaje` en los campos `texto1` y `texto2` del `Paquete`, y donde lo encuentre ponerlo a `NULL`.
 4. Si el `Mensaje` recibido es el segundo del grupo, y por tanto `texto1` y `texto2` ya han quedado a `NULL`, borrar el `Paquete` del mapa y mostrar el siguiente mensaje por pantalla: “[Nombre del consumidor]: grupo N completado”, donde N sería el identificador del grupo.

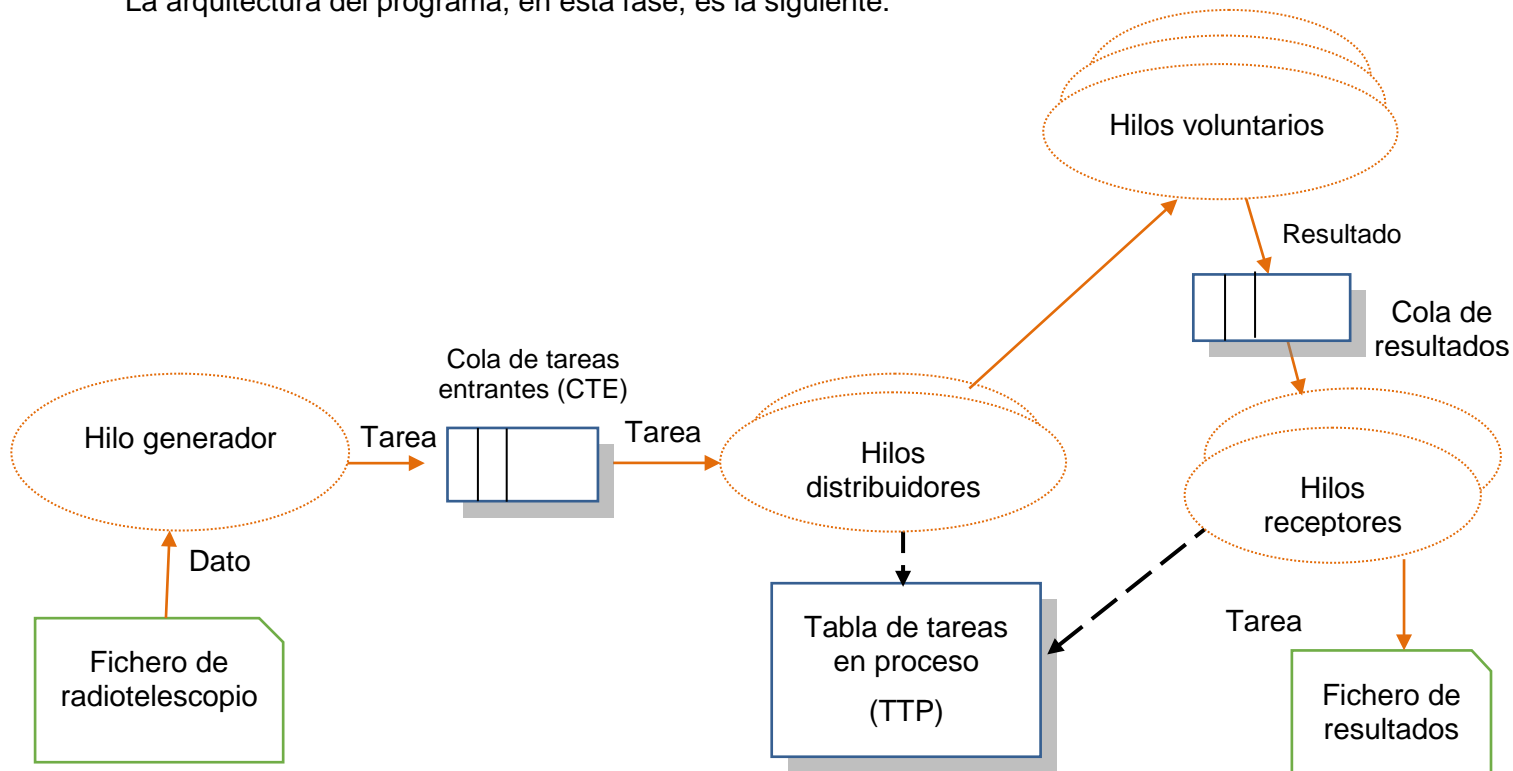


Analice los posibles problemas de concurrencia:

-
- a) ¿Existen problemas de concurrencia por el hecho de que un hilo productor y varios hilos consumidores están accediendo al mismo objeto del mapa, uno para introducir, otros para buscar y modificar e incluso borrar?
 - b) ¿Existen problemas de concurrencia cuando dos consumidores están accediendo al mismo objeto Paquete del mapa para modificarlo e incluso borrarlo? Si es así, establezca la sección crítica mínima y solucione el problema mediante una región crítica con cerrojos o synchronized. También podría solucionarlo añadiendo un método a la clase Paquete que sirva para realizar las operaciones descritas en el consumidor y haciendo que esa clase se comporte como monitor.
-

Caso práctico fase 2

La arquitectura del programa, en esta fase, es la siguiente:



Se añaden o modifican los siguientes elementos respecto a la fase 1:

- Se modifica el **hilo distribuidor**. Ahora, para cada objeto *tarea* extraído de la *cola de tareas entrantes*:
 - **Guarda** en la **tabla de tareas en proceso** aquella información que necesite el receptor para su funcionamiento.
 - **Crea tres hilos voluntarios** pasándoles la tarea.



Defina una constante en la clase principal, con valor 3, para que, en el futuro, se pueda elegir el número de hilos voluntarios cambiando su valor.

- Se añaden los **hilos voluntarios**. Un hilo voluntario simula el comportamiento que tendría esta aplicación enviando una tarea a un ordenador que forme parte del sistema Seti para realizar el procesamiento distribuido, y recibir el resultado de su procesamiento. En concreto:
 - Recibe, como parámetro en su constructor, la tarea a procesar. Esta tarea contiene la información mínima necesaria para su procesamiento (sin incluir información que no incumbe a los voluntarios), esto es, identificador y línea procedente del fichero de radiotelescopio.
 - Procesa la tarea. Este procesamiento se simula mediante:

- Una espera pasiva de un tiempo proporcional al número de caracteres de la línea que contiene la tarea.



Defina una constante pública RETARDO e invoque al método sleep() de la clase Thread pasándole como parámetro la longitud de la cadena multiplicada por el RETARDO. Así, ajustando el valor de RETARDO, los voluntarios tardarán más o menos en devolver su resultado y se ralentizará el funcionamiento para que sea más fácilmente observable.

- Calcula un resultado. Por simplicidad, consiste en el número de caracteres que incluye, procedentes del fichero de radiotelescopio.
 - Devuelve el resultado (asociado, al menos, al *id de tarea* para que sea identificable), y para ello lo encola en la *Cola de resultados*.
- Se añaden **dos hilos receptores**. Estos hilos validan cada resultado que se recibe de los voluntarios, para deducir si es o no correcto. Se valida por unanimidad: se pidió procesar cada tarea a tres voluntarios, por lo que se deberían recibir tres resultados idénticos (unanimidad) para considerarlo correcto. Si algún voluntario falla, se puede detectar porque su resultado será diferente de los otros⁴ resultados. **En esta práctica, si los 3 resultados son idénticos, se considera que la tarea obtiene un resultado correcto. Si alguno de los resultados difiere de los otros, se considera que la tarea obtiene un resultado incorrecto.**

Cada hilo receptor, en cada iteración de un bucle sin fin:

- **Obtiene** un *Resultado*⁵ extrayéndolo de la *Cola de resultados*.
- **Busca**, en la *tabla de tareas en proceso*, la información disponible de la *tarea en proceso* a la que corresponda el *Resultado* que acaba de extraer de la cola.
- **Valida cada resultado**:
 - Si la tarea obtiene un **resultado correcto**:
 - Elimina esta *tarea en proceso* de la *tabla de tareas en proceso*.
 - Escribe al final de un único *fichero de resultados* (compartido por todos los receptores) información de esta tarea finalizada: una línea de texto con el formato "Id tarea : línea de radiotelescopio : resultado obtenido".
 - **En cuanto se pueda deducir** que la tarea obtiene un **resultado incorrecto** (por ejemplo, al recibir un segundo resultado para la tarea ya puede suceder que difiera del primero), elimina esta *tarea en proceso* de la *tabla de tareas en proceso*.

⁴ Aunque en esta práctica se pide por unanimidad (todos los resultados iguales), observe que al obtener tres resultados de distintos voluntarios se podría recuperar un error en un voluntario, pues se puede obtener una mayoría (2) de resultados iguales. Ello sigue la fórmula $\text{Voluntarios} = 2N + 1$, siendo N el número de errores (de tipo bizantino) que se desea poder recuperar. Es decir, necesitamos enviar a 3 voluntarios para poder recuperar 1 error (por mayoría de 2 resultados iguales), a 5 voluntarios para recuperar 2 errores (por mayoría de 3 resultados iguales), ...

⁵ Objeto de la clase que usted considere conveniente que, en su solución, se encole en la cola de resultados



En esta fase, este caso nunca se produce pues los voluntarios siempre devuelven valores correctos. Se desarrollará en la fase siguiente.

- Si aún **no se puede deducir** que el resultado sea correcto o incorrecto, continúa.
- Se modifica el **hilo principal** en los siguientes aspectos:
 - **Crea** el nº de hilos receptores que indique una constante, de valor 2.
 - Continúa mostrando **por pantalla**, cada segundo, el número de tareas que hay en la cola de tareas entrantes, pero ahora además mostrará con la misma periodicidad (cada segundo) el **contenido** de la *tabla de tareas en proceso*.



Se debe mostrar una *tarea en proceso* por línea, con los valores: identificador de tarea, dato de la tarea y, en su caso, la información adicional que considere conveniente.

- Se crea la clase que modela la **tarea en proceso**:
 - Esta clase debe tener la responsabilidad de **decidir** si la *tarea en proceso* ha finalizado o no, y si ha sido o no correctamente. Para ello, debe ofrecer un método público `validarResultado(int resultado)` que reciba cada resultado y, en base al contenido actual de la *tarea en proceso*, devuelva esta decisión. Los hilos receptores invocan este método para actuar en consecuencia, como se ha indicado anteriormente.
 - Además, se recomienda que la *tarea en proceso* también ofrezca, si no lo tiene ya, algún método que genere una cadena de caracteres construida a partir de sus campos significativos. Este método puede ser utilizado, por ejemplo, desde el programa principal que escribir en pantalla un detalle de las tareas en proceso existentes.



Como solución para los posibles problemas de concurrencia que puedan darse, se **debe** configurar esta clase (que modela la *tarea en proceso*) para que sea un **monitor**.



En este caso se **DEBE** usar un objeto de la clase `java.util.concurrent.ConcurrentHashMap<K,V>` como *tabla de tareas en proceso*, donde la clave debe ser un Integer (almacenará el identificador de la tarea) y el valor un objeto *Tarea en proceso*. No se admiten otras posibles soluciones.



El programa en ningún momento debe realizar esperas activas.



El nombre del *fichero de resultados* debe estar definido en una constante en la clase del programa principal.



Consideraciones referentes a **secciones críticas** que pueden aparecer en esta fase 2. Debe valorar y aportar, en su caso, una solución que garantice la consistencia en la información, utilizando los mecanismos propuestos en esta fase:


- 1) Analice la existencia o no de una sección crítica en la *tabla de tareas en proceso* si varios hilos distribuidores añadiesen objetos *tarea en proceso* de forma concurrente y también varios hilos receptores accedieran a ella para consultarlos.
-

-
- 2) Analice la existencia o no de una sección crítica cuando varios hilos receptores manipulan el mismo objeto *tarea en proceso* (para validar un resultado) de la *tabla de tareas en proceso* de forma concurrente.
 - 3) Analice la existencia o no de una sección crítica cuando varios hilos receptores escriben en el *fichero de resultados*. En caso de que exista esa sección crítica, **DEBE** resolverla usando una región crítica implementada por **cerrojos**.
 - 4) Analice la existencia o no de una sección crítica si el hilo principal consultase la *tabla de tareas en proceso* y uno o más hilos distribuidores o receptores pudiesen estar modificando esa tabla o su contenido de forma concurrente.
-


5.3 Fase 3


Entrenamientos fase 3


F3E1 Región crítica implementada con semáforos

 Este entrenador NO se requiere en esta fase, pero puede serle útil si quiere experimentar con este uso de los semáforos estudiado en la teoría de la asignatura.

Cuando realizó el ejercicio **F1E2** observó que existían problemas de concurrencia con el objeto compartido `ArrayList` que era manipulado concurrentemente por varios hilos. En este entrenamiento los hilos deben manipular el objeto compartido dentro de una Región Crítica implementada con un semáforo binario. Compruebe que ahora no se producen errores durante su ejecución.

 En java un semáforo se implementa con la clase `java.util.concurrent.Semaphore`

 Un semáforo implementa una Región Crítica cuando se inicializa a 1. Esta implementación es similar a las que se usaron para implementar Regiones Críticas en la fase 2 mediante `synchronized` o cerrojo.

 Los hilos deben compartir el objeto de clase `java.util.concurrent.Semaphore` que utilice. Este objeto debe ser por tanto instanciado en la clase principal y pasado como argumento del constructor al crear los hilos.

F3E2 Semáforo como control de paso

En este apartado se trabajará con un semáforo, pero no para implementar regiones críticas, sino como control de paso, es decir, para bloquear la entrada de uno o varios hilos en una sección de código hasta que otro hilo les dé paso (siempre y cuando se encuentren más hilos ejecutando esa sección de código que el máximo configurado, de lo contrario podrán acceder directamente).

Escriba un programa que crea 10 hilos, cada uno de los cuales realizará una espera de 5 segundos. Sin embargo, no puede haber más de 2 hilos en ejecución al mismo tiempo (sin contar el propio programa principal). El control del número de hilos en ejecución debe realizarse mediante un semáforo. El hilo principal deberá invocar al método `acquire()` del semáforo antes de poner en ejecución a cada hilo, y a su vez éstos deberán invocar al método `release()` del semáforo cuando terminen.

Una vez realizado el programa y comprobado su correcto funcionamiento responda a la siguiente pregunta:

¿Qué cambios hay que llevar a cabo en el código para modificar el número máximo de hilos en ejecución simultánea?

F3E3 AtomicInteger para id tarea.

El objetivo de los ejercicios que se proponen en este entrenador es que los mensajes que se transmiten entre productores y consumidores tengan un identificador, además del texto, y que este identificador sea único.

A partir del ejercicio **F1E3** realice las siguientes modificaciones:

- Cree una clase nueva, llamada Mensaje, que contenga los siguientes atributos públicos: un entero, llamado identificador, y un String, llamado texto. Puede utilizar la clase que hizo en el entrenador F2E4.
- En la clase principal, declare una variable de tipo int llamada ultimoidentificador, con valor inicial 0. Cree 5 hilos productores compartiendo esa variable con ellos (además de la cola, que en este ejercicio almacenará objetos Mensaje). Cree un hilo consumidor.
- Modifique el hilo productor de forma que introduzca en la cola objetos Mensaje. Para ello creará objetos mensaje con identificadores únicos haciendo uso de la variable compartida ultimoidentificador, para lo cual tendrá que incrementarla con cada nuevo objeto Mensaje.
- Modifique el hilo consumidor de forma que tras sacar un Mensaje de la cola muestre por pantalla el identificador del mensaje y el texto del mensaje.

Cuando ejecute el programa observará que el consumidor muestra mensajes con el mismo identificador. Esto es así porque la variable ultimoidentificador, que supuestamente están compartiendo los hilos productores, en realidad no es un recurso compartido. El motivo es que en Java las variables de tipo primitivo, como es int, se pasan por valor, con lo cual cada consumidor tiene una copia de la variable y por tanto no comparten nada.

Para solucionar este problema se van a usar dos estrategias.

Estrategia 1a

La primera de ella consiste en crear una clase que almacene el último identificador usado y se comparta con los productores. Para ello:

- Cree una clase, llamada **Identificador**, con las siguientes características:
 - Un atributo privado llamado ultimoidentificador de tipo entero (int), con valor inicial 0.
 - El método público:


```
int nuevoIdentificador(): Incrementa el atributo ultimoidentificador y devuelve su valor.
```
- Modifique la clase principal para que, al crear los hilos productores, se comparta con ellos un objeto de la clase Identificador que se debe haber instanciado previamente.
- Modifique el hilo productor de forma que el valor que asigne al atributo identificador de cada objeto Mensaje sea el que le devuelva el método nuevoIdentificador() de la clase Identificador.

Cuando ejecute este programa observará que el consumidor ya mostrará mensajes con identificadores distintos. ¿Pero lo hace siempre? Si aumenta el número de productores o el número de mensajes que mete cada productor, es posible que llegue a ver que en algunos casos hay identificadores iguales o se pierden identificadores. La causa de estos errores es que puede haber problemas de concurrencia por el hecho de estar varios hilos productores accediendo al mismo objeto compartido, es decir, a la operación de incremento del atributo ultimoidentificador.

Aunque la instrucción que se ejecuta en nuevoIdentificador() sea algo tan simple como ultimoIdentificador++, esta instrucción se descompone en varias instrucciones en lenguaje máquina. Por ello esta instrucción debería ejecutarse en exclusión mutua usando alguna de las herramientas que ya conoce como es la Región Crítica.

Estrategia 1b

A continuación, como alternativa a la Región Crítica, le vamos a proponer que use la clase AtomicInteger. Y es que esta clase permite ejecutar atómicamente operaciones sencillas de incrementos como la que se necesita hacer con ultimoidentificador. Para ello haga las siguientes

modificaciones en la clase Identificador:

- Declare el atributo `ultimoIdentificador` como `AtomicInteger` e instancie en el constructor un objeto de esa clase de valor 0.
- Modifique el método `nuevoIdentificador()` para realizar el incremento con alguno de los métodos que tiene la clase `AtomicInteger`

Cuando ejecute este programa observará que funciona correctamente y nunca se muestran identificadores repetidos o se pierden identificadores.

Estrategia 2

Como alternativa a compartir el objeto Identificador entre los productores, se proponen estos cambios cuyo objetivo es que sea el propio objeto Mensaje el que genere su identificador único al instanciarse:

- Modifique la clase Mensaje:
 - Añada un atributo privado llamado `ultimoidentificador` de tipo `AtomicInteger`. Realice en la misma línea de la declaración la instanciación de un objeto de esa clase de valor 0. Para que este atributo sea único en todos los objetos de la clase Mensaje, debe declararlo como `static`
 - El constructor de la clase no recibirá un parámetro con el valor que se debe asignar al atributo `identificador`, sino que directamente calculará el siguiente identificador usando el atributo `ultimoidentificador` invocando al método adecuado de `AtomicInteger`.
- Modifique el programa principal para que no comparta con los productores nada más que la cola.
- Modifique el hilo productor de forma que cree los Mensajes sin preocuparse de obtener previamente un nuevo identificador, pues este lo genera internamente cada nuevo objeto Mensaje.

Cuando ejecute este programa observará que también funciona correctamente.

Caso práctico fase 3

La arquitectura del programa es ya la que se mostró en el [apartado 4](#).

Se añaden o modifican los siguientes elementos respecto a la fase 2:

- Se modifica el **hilo voluntario**, de forma que pueda producir resultados erróneos. Para ello defina en la clase principal una constante llamada `PROBABILIDAD_ERROR_VOLUNTARIO` (con valores reales entre 0 y 1), de forma que cuando su valor sea 0 no genere errores y cuando su valor sea 1 genere errores siempre. Puede usar este código para implementarlo:

```
// Genera, con cierta probabilidad, un resultado erróneo
if (ThreadLocalRandom.current().nextDouble(0, 1) <
    Seti.PROBABILIDAD_ERROR_VOLUNTARIO)
    // Resultado erróneo (rango 1000..1999 para que sea identificable)
    resultado = ThreadLocalRandom.current().nextInt(1000, 2000);
else
    // Resultado correcto: como fase anterior
```

- Se modifica el **hilo receptor**, para tratar el caso en el que se detecte que el resultado de la *tarea* es incorrecto. **En cuanto se detecte** que esto sucede:
 - **Elimina** esta *tarea en proceso* de la *tabla de tareas en proceso*
 - **Crea una nueva tarea**, diferente de la que acaba de fallar, por tanto, con un *identificador de tarea* único, que contiene una copia de la línea procedente del fichero radiotelescopio presente en la *tarea en proceso* que falló, y la encola en la *cola de tareas entrantes*. De esta manera, esta nueva tarea llegará a otros 3 voluntarios.



Observe que, si han llegado resultados de dos voluntarios, y son diferentes, la *tarea en proceso* se elimina de la *tabla de tareas en proceso*, por lo que cuando llegue el tercer resultado del último voluntario, el hilo receptor no encontrará la *tarea en proceso* asociada al resultado en la *tabla de tareas en proceso*. Por tanto, si en la fase 2 consideraba esta situación un error, deberá modificarlo.

- Se modifica la constante que se usa en el programa principal para crear hilos generadores, de forma que, a partir de ahora, se puedan crear **varios hilos generadores** en el sistema. Todos los hilos generadores deberán leer del mismo fichero de radiotelescopio.
- Se establece **un límite en el número total de hilos voluntarios que pueden existir** en el sistema en un instante dado. Para controlarlo, se debe usar un contador de hilos voluntarios manejado por los **hilos distribuidores**, y estos deben consultarlo antes de crear un nuevo voluntario para que procese una tarea, esperando cuando el contador ha superado el límite. Así mismo, los **hilos voluntarios**, antes de finalizar, tendrán que decrementar el contador y despertar a los hilos distribuidores que pudieran estar esperando por haberse superado el límite.



Consideraciones referentes a **secciones críticas** que puedan aparecer en esta fase 3. Debe, en su caso, aportar una solución adecuada que garantice la consistencia en la información, utilizando los mecanismos propuestos en esta fase:

-
- 1) El control del máximo de hilos voluntarios que pueden existir en el sistema se DEBE implementar usando un único **semáforo**. No se admiten otras posibles soluciones.
 - 2) Se debe garantizar que el identificador de tarea es único, y en esta fase existen varios hilos generadores creando tareas y varios hilos receptores que también crearán nuevas tareas cuando tengan que reinyectar tareas que no pudieron resolverse en el intento anterior. Se deben por tanto sincronizar los hilos para generar identificadores de tarea únicos. Para ello en este apartado se DEBE usar una solución basada en un único objeto de la clase `java.util.concurrent.AtomicInteger`.
 - 3) Analice la existencia o no de una sección crítica en los generadores a la hora de realizar las lecturas del fichero de radiotelescopio. En caso de existir esta sección crítica, resuélvala con el mecanismo de región crítica de java **synchronized()** que usó en el entrenador F2E2.
 - 4) Analice la existencia o no de una sección crítica en el acceso a la *cola de tareas entrantes* dado que existen dos fuentes de tareas: hilos generadores (tareas desde fichero radiotelescopio) e hilos receptores (tareas que se reinyectan por no haberse podido resolver con éxito).
-

5.4 Fase 4

Entrenamiento fase 4

F4E1 Región crítica con espera condicional

A partir del entrenamiento **F1E3** realice las modificaciones necesarias para cumplir con las siguientes condiciones:

- Los hilos productores NO deben introducir nuevos mensajes en la cola si el sumatorio del número de caracteres de todos los mensajes presentes en la cola más la longitud del nuevo mensaje supera un cierto valor UMBRAL.
- Los hilos productores NO deben hacer esperas activas; por tanto, si no pueden encolar el nuevo mensaje, deben quedarse bloqueados hasta que sea posible encolar nuevos mensajes.
- Los hilos consumidores, cada vez que sacan un elemento de la cola, deben despertar a todos los hilos productores bloqueados para que estos tengan la oportunidad de evaluar si hay sitio para su nuevo mensaje o deben seguir bloqueados.



Las variables que use para sincronizar los productores y consumidores y para anotar el número de caracteres de todos los mensajes de la cola deberán ser accesibles tanto a los productores como a los consumidores.



Recuerde que en JAVA los tipos primitivos se pasan por valor. Por tanto, si necesita que varios hilos compartan un entero, este debe ser una variable global. Recuerde que una variable global es aquella declarada como pública y estática.



Determine cuáles son las secciones críticas de código. Si necesita implementar Regiones Críticas Condicionales (RCC), deberá hacerlo usando exclusivamente cerrojos y variables de condición. No se admite el uso de otros mecanismos para este cometido.



En JAVA, una variable de condición se implementa con la clase `java.util.concurrent.locks.Condition` (recuerde: siempre asociada a un cerrojo).

Tras realizar este entrenamiento, y comprobado que funcione, puede plantearse si funcionaría correctamente si los hilos consumidores, en lugar de despertar a todos los hilos productores bloqueados, despertasen solo a uno de ellos. ¿Cómo se codificaría?

Caso práctico fase 4

La arquitectura del programa es la que ya se mostró en el [apartado 4](#).

Partiendo de la fase 3, se modifica lo siguiente:

- Se establece un **máximo de tareas vivas** en el sistema. Incluye tareas en cualquier estado: pendiente o en proceso.
- Se establece un **máximo de caracteres** procedentes del fichero de radiotelescopio que pueden estar en proceso en el sistema. Recuerde que cada tarea contiene una línea de caracteres procedente del fichero de radiotelescopio.
- Como consecuencia de ambas restricciones, los **hilos generadores** podrían tener que esperar, si se supera alguno de estos límites, antes de añadir una tarea a la *cola de tareas entrantes*. Además, los **hilos receptores** podrían tener que despertar a los hilos generadores, pues cuando eliminan una tarea del sistema podría dejarse de cumplir alguna de estas restricciones.



Utilice un procedimiento similar al expuesto en el entrenamiento de esta fase, declarando las variables globales en la clase principal, para llevar la cuenta del máximo de tareas vivas y el máximo de caracteres.



Resuelva los problemas de concurrencia que encuentre en las **secciones críticas** que aparecen en esta fase al manipular varios hilos las variables globales de forma concurrente usando una región crítica condicional implementada con **cerrojo y variable de condición**.



Razone si podría encontrar alguna forma de sustituir la RCC implementada mediante un cerrojo y variable condición por un semáforo.
