Prácticas Concurrencia y Distribución (18/19)

Arno Formella, Anália García Lourenço, Hugo López Fernández, David Olivieri

29 de marzo de 2019

Las prácticas de este curso están organizadas en actividades semanales para mantener un esfuerzo constante a lo largo del curso. Las entregas (¡semanales!) se realizan mediante la herramienta FaiTIC que estará configurada para permitir la subida de ficheros durante las horas de prácticas en el aula de informática.
Las entregas consisten en el fichero según lo pedido en la actividad que se haya publicado **que se sube a la plataforma durante las clases presenciales de prácticas** de cada semana. Dichas entregas sirven al mismo tiempo como **testigo de asistencia** a clases prácticas. No obstante el profesorado de prácticas puede usar otras medidas adicionales para monitorizar dicha asistencia.
Las entregas consisten en la subida de un **único** fichero simple o de tipo archivo (`.zip`, `.rar`, `.tgz`, etc.). Dichos ficheros **siempre** deberían tener nombres que se forman de la siguiente manera:

`Apellido1_Apellido2_Nombre_Grupo.Extensión`

donde

- el `Apellido1`, `Apellido2`, y `Nombre` se entienden como tales,
- `Grupo` es uno de CDI1 hasta CDI6,
- `Extensión` según tipo de fichero que que puede ser solamente un fichero `.java` con el código fuente, un solo fichero como archivo que contiene la documentación y el código adicional necesario.

*Ficheros con nombres que no cumplen con esta nomenclatura serán simplemente **ignorados***. No existirá la entrega de ficheros fuera de estos plazos.

1. Práctica 1: Introducción a la concurrencia en Java

Objetivos: Adquirir conocimientos básicos sobre la forma como está implementada la concurrencia en Java.

Material adicional: Son de especial interés los siguientes enlaces

- `http://docs.oracle.com/javase/7/docs/`
- `http://docs.oracle.com/javase/8/docs/`
- `http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`
- `http://www.stack.nl/~dimitri/doxygen/index.html`
- `http://en.wikipedia.org/wiki/Markdown`

Requisito general a todos los programas en todos los programas concurrentes es: **siempre termina el programa y todos sus subprocesos/hilos con un mensaje como *Program of exercise X has terminated***, es decir, todos los componentes del programa concurrente terminan correctamente su ejecución.

Las preguntas que aparecen intercaladas en los anunciados tienen como objetivos: animar a la reflexión y al auto-aprendizaje, servir como ejemplos de posibles preguntas en la fase de evaluación (examen), fundamentan la base para los breves informes que se puedan entregan para las actividades.

1. Examina en el manual y con ejemplos las dos formas que provee Java para crear un hilo: la clase `Thread` y la interfaz `Runnable`.

   ¿Hay alguna diferencia de funcionamiento entre ambas formas? ¿A nivel de diseño, cuál te parece preferible, y por qué?

2. Utiliza la forma con `Runnable` para crear un programa que cree y ejecute tantos hilos como se le indica via línea de comando

3. Utiliza tu programa del apartado anterior y aumenta para

   - que cada hilo imprima en pantalla un mensaje como *Hello world, I'm a java thread number X*.
   - y después de uno o varios segundos (puedes usar un segundo argumento via línea de comando), un mensaje como *Bye, this was thread number X*,

   ¿Las salidas del programa reflejan lo que has esperado?

2. Práctica 2: Comportamiento básico de los hilos

Objetivos: Sincronización simple de los hilos al final de ejecución, medición de tiempo de ejecución.

1. Determinación de la terminación del hilo.

   Este ejercicio explora cómo determinar cuando termina un hilo o un grupo de hilos. Para hacer esto, siga estos pasos:

   a) **Configuración del problema:** Duplicar el código de la semana pasada. En particular, escriba una clase `MyThread` que extiende `Thread` (o implementa `Runnable`) e imprima el nombre del hilo en ejecución. (Ten en cuenta que tu versión podría tener una estructura ligeramente diferente y/o con diferentes nombres de clase, pero debes lograr esencialmente el mismo resultado).

   b) **Detalles:** en el método `main` de la clase principal), cree una lista (o una matriz) de hilos e inícielos. Inmediatamente después de iniciar los hilos, desde el hilo principal, imprima a la pantalla un mensaje (algo como: *el programa ha terminado*). Una estructura en Java para crear una lista de hilos podría ser algo como lo siguiente:

   ```
   final int NUMBER_OF_THREADS = 32;

   List<Thread> threadList =
     new ArrayList<Thread>(NUMBER_OF_THREADS);

   for(int i=1; i<=NUMBER_OF_THREADS; ++i) {
     //...
   }
   ```

   ¿Cuál es el resultado de tu código? ¿En qué orden se imprimen los hilos?

c) **Modificar la clase principal:** ahora queremos imprimir un mensaje desde el hilo principal cuando todos los otros hilos han terminado. Con el método `isAlive()` en un bucle de control en la rutina principal, se puede determinar el estado de cada hilo (si todavía está vivo). La estructura del bucle/`isAlive()` para capturar el estado de cada subproceso debería tener una estructura similar la siguiente:

```
while(t.isAlive()) {
  try {
  }
  catch() {
  }
}
```

donde `t` es uno de los hilos.

d) **Modificación con `join`:** Dado que la técnica anterior de bucle-`isAlive()` se utiliza con tanta frecuencia, Java tiene un método especial llamado `join()` que hace principalmente lo mismo. Reemplace el bucle/`isAlive()` de arriba con `join()` ¿Funciona exactamente igual? ¿Afecta la ejecución de los subprocesos en ejecución?

2. Medición del tiempo de ejecución.

   Queremos mapear el ciclo de vida de los hilos en el programa concurrente registrando los tiempos cuando cambian sus estados. Sigue los pasos aquí para investigar el comportamiento de los hilos:

   a) **Configuración del problema:** escriba una clase principal (que contiene `main`) y una clase que extiende `Thread` como en el problema 1 de arriba. En la clase principal, comienza la ejecución de una lista de hilos que van a realizar una operación matemática (que se explica en el siguiente paso).

   b) **Implementación de la clase `MyThread`:** el hilo debe ejecutar una operación de cálculo intensivo. Para esto, una prueba históricamente interesante es la *prueba de remojo PDP-11*, (vea `https://en.wikipedia.org/wiki/PDP-11`) que se basa en la aplicación continua de funciones trascendentes. Para esto, podemos aplicar continuamente el tangente y su inverso, seguido por la raíz cúbica. Una implementación puede ser la siguiente:

```
for(int i=0; i<1000000; ++i) {
  double d=tan(atan(
    tan(atan(
      tan(atan(
        tan(atan(
          tan(atan(123456789.123456789))
        ))
      ))
    ))
  ));
  cbrt(d);
}
```

   c) **Diagrama de ejecución de hilos:** ahora queremos entender lo que hace cada hilo durante su vida. Inserte diagnósticos (utilizando `System.Print.` o `Logger`) en tu código para mapear el ciclo de vida de cada hilo. Debes distinguir entre el momento en que se ejecuta y el momento en que termina. ¿Puedes distinguir entre la creación del hilo, la ejecución y la terminación final? ¿Qué problemas encuentras cuando intentas determinar las diferentes fases? El siguiente segmento de código podría ser útil (el mensaje debe también incluir el tiempo):

```
LOGGER.debug("Soy {}", Thread.currentThread().getName());
```

d) **Análisis. Tiempo de ejecución respecto a número de hilos:** Estudia el comportamiento de tu código a medida que aumentas la cantidad de subprocesos implementados; es decir, comienza con un hilo y aumenta hasta un gran número de hilos, imprimiendo el tiempo total de ejecución. Sería útil ejecutar el código Java en un script bash (o en Windows, utilizando algo similar) que guarde los resultados de medición en un archivo.

e) **Haz un gráfico de tiempo de ejecución:** desde el archivo del paso anterior, utiliza un programa gráfico como por ejemplo *gnuplot*, *matplotlib* o *seaborn* para hacer gráficos del tiempo de ejecución en función del número de subprocesos (es decir, el eje x es el número de subprocesos, mientras que el eje y es el tiempo de ejecución). A partir de estos gráficos, ¿qué conclusiones puedes sacar?

3. Práctica 3: Gestión de hilos y tareas concurrentes independientes

Objetivos: Gestión de hilos y tareas concurrentes independientes

1. **(P4: para entregar en grupo de práctica)**: Variables locales del hilo y interrupciones

   Este problema continúa explorando dos temas importantes en la administración de hilos: *a)* el alcance de las variables locales y *b)* la interrupción de hilos. Para hacer eso, sigue los siguientes pasos:

   a) **Configuración del problema:** crea una clase de hilo vacío (por el momento) (`MiThread`) que *extends* `Thread` (o *implements* `Runnable`). Tambien crea una clase *principal* (por ejemplo, `MiProblema`) donde dentro de su método *main* crea un *array* (o `ArrayList`) del objeto `MiThread`.

   b) **Variables locales del hilo:** En la clase `MiThread`, crea una variable privada de tipo `Integer` (con nombre como `miSuma`). ¿Qué sucede con respecto a este atributo privado cuando hay multiples instancias de hilos ejecutandose? A continuación, sobreescribe el método `run()` para sumar los números de 0 a algún número N y guardarlo dentro de `miSuma`. Una estructura muy esquemática para la clase del hilo sería:

```
class MiThread {
  private Integer miSuma;
    //...
  @Override
  public void run() {
    // for loop {
       // imprime "started", threadID, miSuma
       // dormir
       // incremente miSuma
    // }
    // imprime "finished",  threadID, miSuma
  }
}
```

   Explicar el resultado ¿Se comporta como se esperaba?

   c) **Variables locales en un subproceso:** en el API de Java, `ThreadLocal<>` es un método que se puede usar para mantener variables locales dentro de hilos. Reescribe la clase anterior creando una variable local con el mecanismo `ThreadLocal<Integer>` (consulta la documentación de Java). Demostrar que con este mecanismo los hilos solo suman sus propias variables locales. Ten en cuenta que al utilizar `ThreadLocal<>`, tendrás que usar métodos como `get()` y `set()`.

   d) **Interrumpir un hilo desde Main:** cambia la tarea del hilo de la parte 2 (de arriba) para calcular la constante π (PI). Para ello, hay muchas fórmulas iterativas, pero una muy sencilla de implementar es la siguiente:

```
for(int i=3; i<100000; i+=2) {
  if(negative)
    pi -= 1.0/i;
  else
    pi += 1.0/i;
  negative = !negative;
}
pi+=1.0;
pi*=4.0;
```

A continuación, escribe el código apropiado en el programa principal para que interrumpa los hilos después de que el hilo *main* duerma por un tiempo aleatorio. ¿Qué comportamiento observas? ¿Se interrumpe inmediatamente el hilo? ¿Cómo podrías cuantificar tu respuesta?

2. **(P3: para entregar dentro de una semana)**: Estudio de dividir tareas concurrentes: dividir tareas en una matriz

   Este problema estudia hilos concurrentes independientes. En particular, dada una matriz grande, cada hilo debe realizar un cálculo en una subregión de esta matriz. Un cálculo simple es el de un filtro numérico (típico en el procesamiento de imágenes), que reemplaza cada elemento con su promedio obtenido a partir de los valores de sus elementos vecinos inmediatos. El problema explora el rendimiento en función del número de hilos no interactivos y no sincronizados (menos al final). Para hacer eso, sigue los siguientes pasos:

   a) **Configuración del problema:** crea un clase de hilo vacío (por el momento) (`MiThread`) que *extends* `Thread` (o *implements* `Runnable`). Tambien crea una clase *main* (por ejemplo, `MiProblema`) con el método principal que crea un *array* (o `ArrayList`) del objeto `MiThread`. Finalmente, crea una clase `MiMatriz` que representa un objeto de matriz de 2 dimensiones e implementa una estrategia para asignar bloques de la matriz a diferentes hilos.

   b) **Desarrollar la clase** `MiThread`: la tarea que se realizará se llama filtro mediano. Dada una matriz, el filtro mediano reemplaza cada elemento de la matriz (i, j) con el promedio calculado con los ocho vecinos y el mismo (ten cuidado cuando te encuentras en los bordes de la matriz). La ecuación para este filtro, J, es:

$$J(i,j) = \frac{1}{(2f+1)^2} \sum_{k=-f}^{f} \sum_{l=-f}^{f} M(i+k, j+l)$$

   donde M es la imagen original y f es el tamaño del filtro. Tratamiento de bordes de matriz: si $M(i+k, j+l)$ resulta en un elemento fuera de la matriz hai que tratar estos casos: puedes reflejar las coordenadas en el borde para simplificar el cálculo. Por ejemplo, puedes asumir que $M(-2, -1) = M(2, 1)$, e igual a lo largo de los demás bordes (y esquinas).

   c) **Desarrollar la clase** `MiMatriz`: esta clase representa el objeto matriz. También es responsable de distribuir los hilos de filtro por toda la matriz La clase debe implementar un método con una estrategia particular para distribuir la lista de hilos. Las estrategias deberían incluir la división de matriz por filas, columnas, bloques o cualquier combinación de los mismos.

   d) **Desarrollar la clase principal:** esta clase principal es simplemente responsable de configurar el problema y crear el objeto `MiMatriz` que ejecutará el filtro dentro de un subregión de la imagen. Tu código debería ser capaz de crear matrices de diferentes tamaños con elementos constantes o aleatorios.

e) **Estudio del comportamiento:** ejecuta tu programa para crear diferentes gráficos (plots) de matriz y número de hilos. A partir de los resultados de estos plots, ¿notas una diferencia en el rendimiento entre las diferentes estrategias? Si es así, ¿cuál podría ser la causa?

4. Práctica 4: Sincronización de hilos

Objetivos: Sincronización de hilos

1. **(P4: para entregar en grupo de práctica)**: Contador sincronizado. El objetivo de este problema es escribir un contador concurrente y sincronizar el acceso de un grupo de hilos a un acumulador. Sigue estos pasos:

   a) **Clase Counter:** crea una clase `Counter` que tenga un método `increment()` que incremente el valor del contador en una unidad y un método para obtener el valor actual del contador. El valor inicial del contador debe ser 0.

   b) **Clase MyTask:** crea una clase `MyTask` que extienda `Thread` o implemente `Runnable` y que se pueda construir recibiendo como parámetro un objeto de tipo `Counter`. Esta tarea debe dormir un tiempo aleatorio entre 0 y 100 milisegundos y a continuación invocar el método `increment()` del contador.

   c) **Método principal:** crea un método principal que construya varios objetos de la clase `MyTask` que comparten un objeto de tipo `Counter`, ejecuta cada tarea en un hilo y espera a que todos los hilos hayan finalizado. Cuando esto ocurra, imprime el valor actual del contador. Ejecuta el programa varias veces y con distintos números de hilos (1, 2, 4, 8, 16, 32, ..., 1024). Explica el comportamiento de la salida.

   d) **Bloques sincronizados:** Modifica la clase `MyTask` para que el incremento del contador se haga desde un bloque sincronizado (el bloque con la palabra reservada `synchronized` debe actuar sobre el objeto contador compartido). ¿Cómo cambia esto el comportamiento del programa principal?

   e) **Utilizando Java AtomicInteger/LongAdder:** Haz las modificaciones necesarias en el código anterior para usar un `AtomicInteger` y/o `LongAdder`. Para realizar la operación del incremento, usa un método adecuado del gran conjunto disponible para estos objetos (consulta la documentación de la API de Java). Explica lo que observas.

2. **(P3: para entregar dentro de una semana)**: Mas sobre sincronización.

   El propósito de este problema es estudiar más a fondo la sincronización de hilos utilizando el código del problema 1.

   a) **Java Locks:** En lugar de bloques sincronizados, la API de Java tiene un conjunto de objetos optimizados de alto nivel para implementar una concurrencia eficiente. En particular, aquí vas a utilizar el paquete `java.util.concurrent.locks`, que proporciona un mecanismo de bloqueo similar al método sincronizado. Reemplaza el bloque sincronizado con un `lock`.

   b) **Análisis/Medidas:** Ejecuta el código del contador para todos los métodos de sincronización empleados variando la cantidad de hilos utilizados. ¿Puedes notar una diferencia en el rendimiento? Explica tus resultados.

   c) **Between Atomic Operations:** El código anterior ilustra que las operaciones atómicas están protegidas. Sin embargo, ahora considera la adición de dos contadores, p y q, donde el acumulador es: $q \longleftarrow q + p$. Este problema demuestra que las operaciones atómicas están protegidas, mientras

que las operaciones entre sí no lo están. Modifica tu código para resolver este problema de calcular correctamente q usando la sincronización según corresponda.

5. Práctica 5: Sincronización y Exclusión

1. **(P4: para entregar en grupo de práctica)**:
   Objetos compartidos y acceso sincronizado: Objetivo: este ejercicio pretende reiterar el concepto de compartir objetos entre hilos y utilizar la referencia del objeto compartido para controlar el acceso de subprocesos a las secciones críticas.

   Configuración del problema:

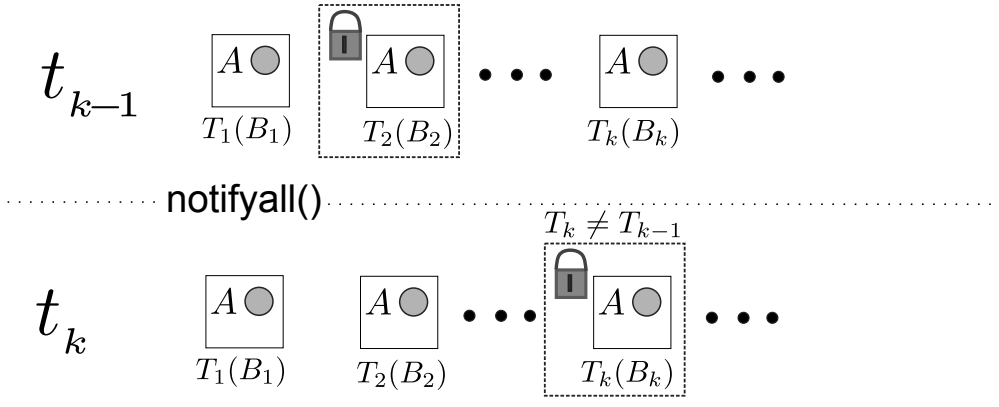
   a) **Class A:** Implementa una clase `ClassA` que contenga un solo método `EnterAndWait()`. Este método debe hacer lo siguiente, en este orden:
      1) imprimir un mensaje indicando cual es el hilo que está comenzando a ejecutarlo;
      2) luego se detendrá durante unos segundos; y
      3) vuelve a imprimir otro mensaje indicando el hilo que está acabando de ejecutar el método.

   b) **Class B:** Implementa una clase `ClassB` que implemente `Runnable` y que se construya recibiendo como parámetro un objeto de la clase `ClassA`. Haz que en el método `run()` simplemente llame al método `enterAndWait()` del objeto con el que ha sido construido.

   c) **Main:** Implementa una clase Main con un método principal en el que se crea un único objeto de la clase `ClassA` y varios objetos de la clase `ClassB` a los que se les pasa a todos como parámetro el objeto de la clase `ClassA`. Después se crearán y ejecutarán el mismo número de hilos (objetos de la clase `Thread`) que de objetos de tipo `ClassB` tengamos pasándoles como parámetros los objetos de la clase `ClassB`, de forma que cada método `run()` de cada objeto de clase `ClassB` se ejecute en un hilo diferente.

   d) **Análisis:** ¿Cuál es el resultado? ¿Cuántos hilos pueden estar simultáneamente ejecutando el método `enterAndWait()`?

   e) **Acceso limitado a la sección crítica:** Utilizando sincronización (el mecanismo que vimos la semana pasada), modifica el código para que solo un hilo pueda estar ejecutando el método `enterAndWait()` en cualquier instante. Explica lo que observas.



2. **(P4: Para empezar en grupo de práctica)**: mecanismo de sincronización `wait()` y `notify()`/`notifyAll()`.

Objetivo: aprender a sincronizar hilos utilizando los mecanismos `wait()` y `notify()`/`notifyAll()`.

Reutiliza el código del problema anterior con las siguientes modificaciones:

a) **Modifica ClassA:** agrega un atributo de tipo entero llamado counter. Este contador se debe de inicializar en el constructor de la clase y cada vez que se invoque al método `enterAndWait()` se debe disminuir en una unidad. Crea también un método `isFinished()` que devuelva verdadero si el valor del counter es 0 y falso en caso contrario. Agrega otro atributo de tipo `Set<Long>`llamado `threadIds` en el que se almacenen los identificadores (`Thread.currentThread().getId()`) de los hilos que han ejecutado el método `enterAndWait()`. Crea también un método que permita recuperar este conjunto.

b) **Modifica ClassB:** Modifica el método `run()` implementar la condición de exclusión condicional con `wait()` y `notify()`/`notifyAll()`. Los hilos deben ejecutarse continuamente hasta que el objeto de la `ClassA` compartido indique que ha finalizado. Básicamente, ahora, cada hilo debe de hacer lo siguiente: llamar al método `wait()` del objeto compartido y esperar a ser despertado para continuar la ejecución. Cuando es despertado, debe comprobar si todavía es posible ejecutar en el `enterAndWait()` y, en ese caso, ejecutarlo. Antes de ponerse a dormir de nuevo, el hilo debe avisar al siguiente hilo con `notify()`/`notifyAll()`.

c) **Modifica el programa principal:** modifica el código del método principal de manera apropiada para que sea posible ejecutar el problema con con `M` hilos y `N` accesos totales a la sección crítica de `enterAndWait()`. Prueba distintas configuraciones (variando `M` y `N`, variando la utilización de `notify()`/`notifyAll()`) y comprueba que funciona como se espera. ¿Cuál es el valor del atributo counter del objeto de tipo ClassA compartido al finalizar la ejecución de todos los hilos?

d) **Comprueba si todos los hilos ejecutan el método `enterAndWait()`:** utilizando la lista de identificadores que han pasado por el método `enterAndWait()` que tiene el objeto de clase `ClassA`, añade una comprobación al final del método principal para saber si todos los hilos creados ejecutan dicho método. Recuerda probar distintas configuraciones (variando `M` y `N`, variando la utilización de `notify()`/`notifyAll()`).

3. **(P3: para entregar dentro de una semana):** Exclusión condicional en el acceso a secciones críticas.

   Objetivo: El propósito de este ejercicio es comprender cómo controlar el acceso de hilos a secciones críticas. En los problemas anteriores todos los hilos compiten continuamente para entrar y es el sistema operativo el que finalmente decide cual es el siguiente hilo. Aquí queremos controlar ese acceso de forma explícita dadas ciertas condiciones adicionales.

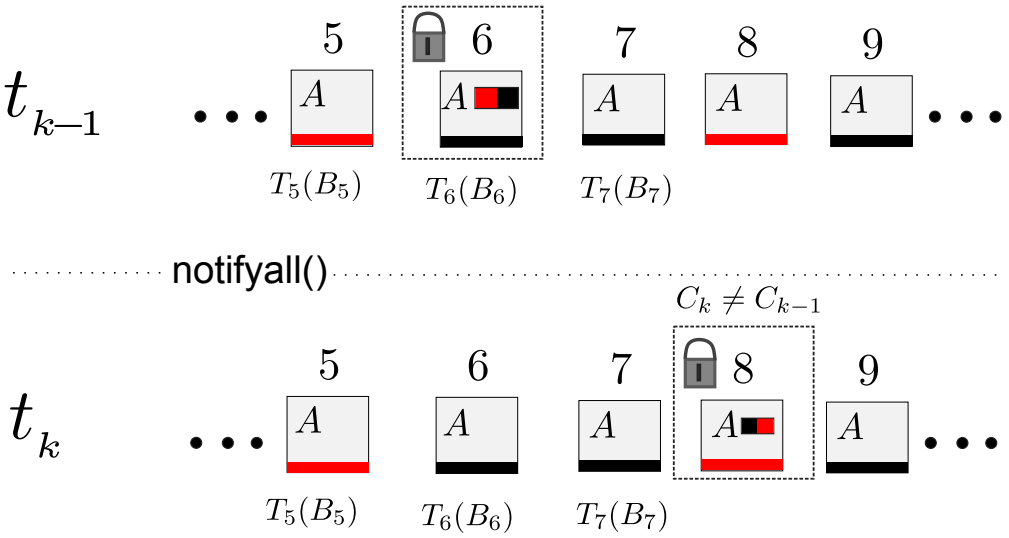
   En este problema, tu programa debería controlar los hilos basándose en una condición adicional: su color. Se tiene que garantizar que dos hilos del mismo color no pueden ejecutar consecutivamente el método `enterAndWait()`.

   Reutiliza el código del problema anterior con las siguientes modificaciones:

   a) **Modifica ClassB:** Haz que los objetos de la clase `ClassB` reciban también en su constructor un objeto de tipo String llamado `color` y que lo guarden en un atributo.

   b) **Modifica el programa principal:** modifica el programa principal para que asigne a los objetos de `ClassB` que crea un color al azar (el color puede tomar los valores “rojo” o “negro”).

c) **Modifica ClassA y ClassB:** finalmente, modifica estas dos clases para que se seleccione al siguiente hilo cumpliendo la regla de que el color del hilo debe alternar entre las iteraciones (por ejemplo, si el hilo anterior fue rojo, el siguiente hilo con acceso a la sección crítica debe ser negro). Una posibilidad para hacer esto es almacenar en el objeto `ClassA` el color del último hilo que ha ejecutado el método `enterAndWait()`. Los hilos podrían obtener este valor para saber si, una vez que estén despiertos, tienen autorización para ejecutar el método.



6. Práctica 6: Sincronización y exclusión II

La semana pasada, se probó como usar el mecanismo de `wait()` y `notify()` / `notifyAll()` para controlar la planificación de hilos dada alguna condición. Ahora, en este laboratorio, queremos llevar esta idea más allá y ordenar los hilos. Esto se hará de dos maneras: 1) utilizando `notifyAll()`, donde cada hilo se despierta y necesita verificar la condición de acceso a la sección crítica (es decir, si es su turno), o 2) directamente usando una referencia al siguiente hilo y despertando solo a dicho hilo mediante `notify()` para que sea el siguiente en acceder.

1. **(P4: para entregar en grupo de práctica)**:
   Objetivo: el objetivo de este ejercicio es que haya dos hilos alternándose en el acceso al método `enterAndWait()` de un objeto de clase `ClassA`.

   Configuración del problema:

   a) **Reutilización de código:** recupera el código de `ClassA` y `ClassB` del apartado 1 de la práctica de la semana pasada (5ª práctica), en que se sincronizaba el acceso de varios hilos a la sección crítica (método `enterAndWait()`).

   b) **Modificación de ClassB:** Ahora, la sincronización (llamadas a `wait()` y `notify()`) debe hacerse empleando las referencias a los objetos `Thread` establecidas. Modifica la clase `ClassB` de manera que: 1) tenga un método `setThread(Thread t)` que permita establecerle una referencia a otro hilo, 2) hacer que ejecuten el código del método `run()` de manera infinita (hasta que hayan sido interrumpidos). Después, dentro del bucle de ejecución infinita, los hilos harán lo siguiente: 1) esperar a ser notificados para ejecutar el método `enterAndWait()`, 2) ejecutar dicho método y 3) notificar al otro hilo.

c) **Main:** Implementa una clase principal con un método `main` en el que se crea un único objeto de la clase `ClassA` y dos objetos de la clase `ClassB` a los que se les pasa como parámetro el objeto de la clase `ClassA` creado. Después se crearán dos hilos (objetos de la clase `Thread`) para que ejecuten los objetos de tipo `ClassB` creados. A cada objeto de la clase `ClassB` se le debe de establecer una referencia al otro hilo (al que no se ejecuta).

Tras lanzar los dos hilos, el método principal debe esperar a que ambos se encuentren en estado de espera para despertar a uno de ellos y que comience el intercambio de ejecuciones del método. Pasado un tiempo, el método principal debe interrumpir los hilos.

2. **(P3: para entregar dentro de una semana)**:

Objetivo: en el código `enterAndWait()` de la semana pasada, cualquier hilo podía entrar en la sección crítica (con un poco más de restricción asociada a los hilos rojos/negros). Ahora, queremos que los hilos entren dado un orden específico. Haremos esto de dos formas diferentes: 1) utilizando `notifyAll()`, que es un método menos eficiente ya que despierta a todos que están esperando, y 2) notificando solo al hilo al que toca su turno. Consulta la figura a continuación para la notificación de llamada explícita en el orden de hilos:

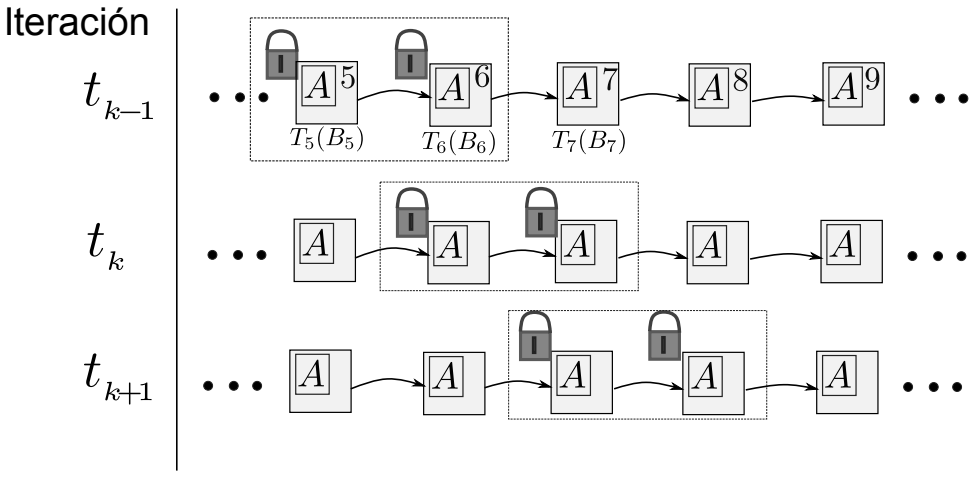
Sigue estos pasos:

a) Configuración del problema: Usa tu código del problema `enterAndWait()` de la semana pasada.

b) Orden secuencial (método de fuerza bruta): utilizando directamente el código de la semana pasada, añade el código necesario en los hilos para ejecutar la sección crítica en orden con llamadas usando `notifyAll()`. Pista: tendrás que asignar a los hilos un número que indique su orden, de manera que este orden pueda ser comprobado antes de ejecutar el método.

c) Orden secuencial (método explícito/eficiente): utilizando la idea de la Figura 2, escribe el código apropiado para que el orden de los hilos que ejecutan la sección crítica se realice explícitamente mediante llamadas al siguiente hilo en la lista.

d) Análisis: Compara el tiempo de ejecución del método de fuerza bruta (parte b) con el método explícito (parte c). Haz un plot y describe tus observaciones en palabras.



7. Práctica 7: Productor/Consumidor

From Wikipedia:

> En computación, el problema del productor-consumidor es un ejemplo clásico de problema de sincronización de multiprocesos. El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.
>
> La idea para la solución es la siguiente, ambos procesos (productor y consumidor) se ejecutan simultáneamente y se "despiertan" o "duermen" según el estado del buffer. Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a "dormir". Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo. Se puede encontrar una solución usando mecanismos de comunicación interprocesos, generalmente se usan semáforos.

1. **(P4: para entregar en grupo de práctica)**: Productor/ consumidor con `wait/notify`.
   Objetivo: Implementar un *bounded buffer* y estudiar el problema del Productor/Consumidor.

   a) **Configuración del problema:** El código consta de tres clases: `Productor`, `Consumidor` y una clase `Buffer`. La clase `Buffer` se usa para sincronizar dos operaciones, escribir y leer (implementadas como funciones miembro de la clase `Buffer`) que son utilizadas por los hilos `Productor` y `Consumidor`. La funcionalidad del código debe ser la siguiente:

      1) Desde el programa principal, lanzar los hilos Productor/Consumidor con un objeto compartido de tipo `Buffer` (llámelo buffer), que contiene una lista enlazada con tipos enteros.
      2) El productor debe agregar valores al búfer (utilizando el método `write()`) y el consumidor debe eliminar valores (utilizando el método `read()`).
      3) El búfer tiene una capacidad máxima que no se puede exceder (no puede contener más elementos que lo que indica dicha capacidad).
      4) Si el productor intenta agregar un valor cuando el búfer ha alcanzado su capacidad, debe esperar al Consumidor (sincronizado con la condición `notFull`).
      5) Si el consumidor intenta disminuir el valor cuando el búfer ha alcanzado su capacidad mínima, debe esperar al productor (sincronizado con la condición `notEmpty`).

   b) Este problema puede producir una situación llamada deadlock. Explica cómo puede suceder esto. Identifica la(s) línea(s) en tu código que podrían producir el punto muerto potencial.

2. **(P3: para entregar dentro de una semana)** Problema del consumidor/productor con Java *locks*:
   Parte 1

   a) Modifica la clase de `Buffer` del problema de arriba para utilizar objetos de `ReentrantLock()` de Java y la interfaz de `Condition`. Debe usar `lock/unlock` del objeto `Lock` y `await/signal` de la interfaz de `Condition` para lograr los mismos objetivos que la `notify/wait` con el método `synchronized`. (Ver la documentación de Java sobre el interfaz `Condition`).

   b) Modifica el código para que un productor nunca escriba dos líneas consecutivas.

3. **(P3: para entregar dentro de una semana)** Problema del consumidor/productor con Java *collection objects*:
   Parte 2

   a) Reemplace la clase `Buffer` con una cola de bloqueo (*blocking Queue*) de la colección Java.

   - Una cola de bloqueo hace que un hilo se bloquee (es decir, pasar al estado de espera) cuando intenta agregar un elemento a una cola completa, o eliminar un elemento de una cola vacía. Permanecerá allí hasta que la cola ya no esté llena o no esté más tiempo. Hay tres colas de bloqueo en Java: `ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`.

   b) Modifica tu código para utilizar este *buffer*.

8. Práctica 8: *Semaphores* y *Thread Executors*

Objetivos: Implementar un semáforo y estudiar su comportamiento dentro del modelo de consumidor productor

1. **(P4: para entregar en grupo de práctica)**: Variables locales del hilo e Interrupts

   Un semáforo es un objeto que controla el acceso a un recurso común. Antes de acceder al recurso, un hilo debe adquirir un permiso del semáforo. Después de terminar con el recurso, el hilo debe devolver el permiso al semáforo.

   a) Dado el siguiente código, completa el código necesario para los métodos `down()` y `up()`. Usa esta implementación en el código del productor-consumidor de la semana pasada.

```
public class MySemaphore {
  public MySemaphore(int initialValue) {
    // POR HACER: Implementar esto
  }
  // Initializa un semaforo con el valor zero.
  public MySemaphore() {
    this(0);
  }

  public void down() {
    // POR HACER: Implementar esto
  }
  public void up() {
    // POR HACER: Implementar esto
  }
}
```

   b) Vuelve a escribir el código del productor-consumidor (del código de la semana pasada), ahora con el tipo de datos `MySemaphore` para controlar la cola de bloqueo.

   c) La API de Java viene con una implementación de un semáforo. Reemplaza `MySemaphore` con la implementación de `Semaphore` de Java. Compara los resultados de cada una de las implementaciones. Explica el resultado ¿Se comporta como se esperaba?

2. **(P3: para entregar dentro de una semana)**: *Executors*.

El propósito de este problema es explorar algunas características muy básicas del servicio `Executor`. Para esto, exploramos dos versiones diferentes del uso del Ejecutor: una con `run()` y la otra con `call()`.

a) **Runnable:** Escriba una tarea ejecutable que sobreescribe el método `run()` para calcular el factorial. En la función `Main`, use las interfaces `Executor` y `Executors` junto con el método `execute` para calcular el factorial de N números, cada uno generado aleatoriamente. En este ejemplo, recuerde que un generador de números aleatorios se hace con el clase `Random`.

- Lanzar el `executor` con el número de procesadores disponible usando `availableProcessors()`.
- La función `factorial` se puede escribir de la siguiente manera, usando un modo de espera para simular más tiempo de cálculo.

```
// Si el numero es 0 o 1, devolver 1 valor
if( (num==0)||(num==1) ) {
  result=1;
}
else {
  // Else, calcular la factorial
  for(int i=2; i<=number; i++) {
    result*=i;
    Thread.sleep(20);
  }
}
```

- Cada tarea `Runnable` debe imprimir el valor o proporcionar un método para acceder a su resultado.
- Supervise el progreso de los hilos en el método `Main` (utilizando un bucle que esté activo mientras el los hilos siguen funcionando).
- Después de apagar el `Executor`, ¿qué sucede si se intenta ejecutar una nueva tarea?

b) **Callable; Hilos que devuelven valores:** Ahora la tarea `Runnable` devolverá los resultados de la factorial. Esto se hace con el método de `call()`. Esto es implementado por:

```
public class FactorialTask implements Callable<Integer>
```

De esta manera, el método `call()` de `FactorialTask` puede devolver el valor del factorial a un objeto `Future`.

- En particular, en el código principal `Main`, los resultados se pueden recopilar en una lista de objetos `Future`, definidos de la siguiente manera:
  `List<Future<Integer>> resultList=new ArrayList<>();`
- La llamada a `submit()` del executor devuelve un objeto `Future<Integer>` que se puede almacenar como:
  `Future<Integer> result=executor.submit(calculator);`
  y añadir a la lista de objetos del `Future`: `resultList.add(result);`
- Usando un thread-pool de tamaño fijo (usando el método `newFixedThreadPool()` del `ThreadPoolExecutor`), se debe lanzar las tareas de `FactorialTask`.
- Desde el código principal, verifique continuamente el estado de las tareas en bucle usando el método `getCompletedTaskNumber()` del executor:
- De esta manera, se debe recopilar la información de cada tarea para determinar si se haya completado o no (verdadero | falso). Escribe un mensaje indicando si las tareas que se gestionan han finalizado o no. Esto se puede hacer usando el método `isDone()`, por ejemplo:

```
for(int i=0; i<resultList.size(); i++) {
   Future<Integer> result=resultList.get(i);
   System.out.printf("Main: Task %d: %s\n",i,result.isDone());
}
```

- Escribe los resultados obtenidos por cada hilo/tarea. Para cada objeto `Future`, obtiene el valor de la factorial devuelto, utilizando el método `get()`. Como ejemplo, parte de la salida podría ser como lo siguiente:

```
Main: Number of Completed Tasks: 8
Main: Task 0: true
Main: Task 1: true
Main: Task 2: true
Main: Task 3: true
Main: Task 4: true
Main: Task 5: true
Main: Task 6: true
Main: Task 7: true
Main: Task 8: false
Main: Task 9: false
Main: Results
Core: Task 0: 120
Core: Task 1: 720
Core: Task 2: 1
Core: Task 3: 720
Core: Task 4: 5040
Core: Task 5: 1
Core: Task 6: 40320
Core: Task 7: 720
Core: Task 8: 2
Core: Task 9: 6
```