Prácticas Concurrencia y Distribución (17/18)

Arno Formella, Anália García Lourenço, Lorena Otero Cerdeira, David Olivieri

4 de mayo de 2018

Las prácticas de este curso están organizadas en actividades semanales para mantener un esfuerzo constante a lo largo del curso. Las entregas (¡cada semana!) se realizan mediante la herramienta FaiTIC que estará configurada para permitir la subida de ficheros.
Una primera entrega (contada como apartado P4 según la guía docente) consiste en el programa según pedido en la actividad que se ha publicado **que se entrega 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 profesor de prácticas puede usar otras medidas adicionales para monitorizar dicha asistencia.
Una segunda entrega (contado como apartado P3 según la guía docente) consiste en la resolución completa de la actividad (código fuente, informe y documentación adicional) **que se entrega como muy tarde al final de cada semana**.
Las entregas consisten en la subida de un **único** fichero simple o de tipo archivo (`.zip`, `.rar`, `.tgz`, etc.). Dichos ficheros **siempre** tendrán nombres que se forman de la siguiente manera:

`Apellido1_Apellido2_Nombre_Grupo_Apartado.Extensión`

donde

- el `Apellido1`, `Apellido2`, y `Nombre` se entienden como tales,
- `Grupo` es uno de CDI1 hasta CDI6,
- `Apartado` indica si es la entrega P4 que se sube en horario de clase práctica en la semana correspondiente o la entrega P3 de toda la resolución que se sube, como muy tarde, hasta el final de la semana correspondiente y
- `Extensión` según tipo de fichero que será para las entregas del apartado P4 solamente un fichero `java` con el código fuente, y para las entregas del apartado P3 un solo fichero como archivo que contiene la documentación en formato PDF 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. Semana 1 (05–11/02): 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 entregan para las actividades (siempre en la semana siguiente).

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. **Entrega P4 (en clase práctica)**

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

3. **Entrega P3 (hasta final de semana (domingo incluido))**

   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. Semana 2 (19–25/02): Comportamiento básico de los hilos

Objetivos: sincronización simple con los hilos al final de ejecución, medición de tiempo de ejecución

1. Determinación de la terminación del hilo **(P4: para entregar en grupo de práctica)**.

   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 *extends* `Thread` (o *implements* `Runnable`) e imprima el nombre del hilo de ejecución. (Ten en cuenta que tu versión podría tener una estructura ligeramente diferente y/o con diferentes nombres de clase, pero esencialmente lograr el mismo resultado).

   b) **Detalles:** en el método `main` de la clase principal), cree una lista (o 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. Usando 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 a este:

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

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 esto:

a) **Configuración del problema:** escriba una clase principal (que contiene `main`) y una clase que *extends* `Thread` como en el problema 1. 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 trascendentales. Para esto, podemos aplicar continuamente el tangente y su inverso, seguido por la raíz cubo. Una implementación es 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 vs 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 a un gran número de hilos, imprimiendo el tiempo total de ejecución. Sería útil ejecutar su código Java en un script bash (o en Windows, utilizando algo similar) y guarda tus resultados 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). Debes hacer el gráfico con distintos escalas: lineal, semi-logarítmica y log-log. A partir de estos gráficos, ¿qué conclusiones puedes sacar sobre el aumento de la cantidad de hilos?

3. Semana 3 (26/02–04/03): 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 e interrupciones

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

   a) **Configuración del problema:** crea una clase de hilo vacío (por el momento) (`MiThread`) que *extends* `Thread` (o *implements* `Runnable`). También crea una clase *principal* (por ejemplo, `MiProblema`) donde dentro de su método *main* crea una *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? Como todos los hilos comparten este mismo atributo, todos los hilos lo modificarán. A continuación, sobreescribe el método `run()` para sumar los números de 0 hasta algún número N y guardar el resultado dentro de `miSuma`. Una estructura muy esquemática para la clase del hilo sería:

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

```
    //   incrementar miSuma
    // }
    // imprime "finished",  threadID, miSuma
  }
}
```

Explica el resultado. ¿Se comporta como esperabas?

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). Demuestra 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 para calcular la constante π. Para ello, hay muchas fórmulas iterativas, pero una muy sencilla de implementar es la siguiente:

```
pi=0.0;
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 principal haya durmido 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 aplicar 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. Para hacer esto, sigue estos pasos:

a) **Configuración del problema:** crea una clase de hilo vacío (por el momento) (`MiThread`) que *extends* `Thread` (o *implements* `Runnable`). También crea una clase *main* (por ejemplo, `MiProblema`) con el método principal que crea una *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 él 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. Para el tratamiento cerca de los bordes de la matriz observa: si $M(i+k, j+l)$ resulta en un elemento fuera de la matriz puedes reflejar las coordinadas 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.

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 el trabajo a la lista de hilos. Las estrategias deberían incluir la división de matriz por filas, columnas, o bloques.

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 una subregión. 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) respecto a tamaño 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. Semana 4 (5/03–11/03): 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 que sincroniza el acceso de un grupo de hilos a un acumulador. Sigue estos pasos:

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

   b) **Clase Contador:** Elabora una clase `Contador` que tenga un método `Incrementar(n)` que ejecute un bucle de n iteraciones que incrementa una variable interna (un acumulador total) cada cierto tiempo, y al acabar devuelva el valor actual de dicha variable.

   c) **Threads:** Desde el método principal, lanza varios hilos que comparten un objeto de tipo `Contador`. Después de esperar un intervalo aleatorio de tiempo (para proporcionar una variación de salida), cada hilo debe llamar al método `Incrementar(n)` con `n` (también elegida de manera aleatoria), y obtener/imprimir el valor del acumulador total devuelto por ese método. Explica el comportamiento de la salida.

   d) **Bloques sincronizados:** Ahora construye un bloque sincronizado para la operación de `Incrementar` (el bloque con la palabra reservada `synchronized` debe actuar sobre el objeto contador compartido para ejecutar un conjunto/bloque de código). ¿Cómo cambia esto el comportamiento del contador?

   e) **Utilizando Java `AtomicInteger/LongAdder`:** Reemplaza 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)**: Más 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 concurrencia eficiente. En particular, aquí vas a utilizar el paquete `java.util.concurrent.locks`, que proporciona un mecanismo para proporcionar exclusión mutua 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 están. Modifica tu código para resolver este problema de calcular correctamente q, usando la sincronización según corresponda.

5. Semana 5 (12/03–18/03): 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 A 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.

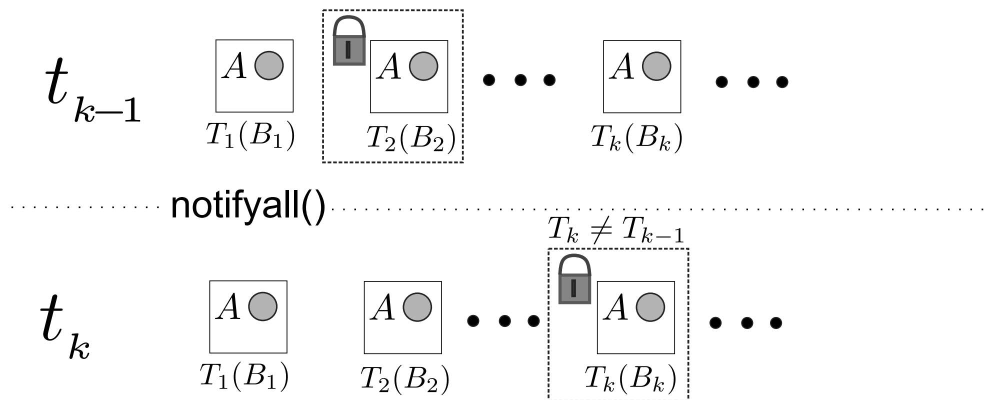
b) **Class B (MyThread):** Realiza una clase B que implemente `Runnable`, que reciba como parámetro un objeto de la clase A, y en el método `run()` simplemente llame al método `EnterAndWait()` de ese objeto.

c) **Main Class:** Una clase con un método `main()`, en el que se crea un objeto de la clase A, y varios objetos de la clase B a los que se les pasa a todos como parámetro el mismo objeto de la clase A. Es decir,

1) todos los objetos de la clase B compartirán el mismo objeto de la clase A.
2) Después se crearán y ejecutarán el mismo número de hilos que de objetos de tipo B tengamos, pasándoles como parámetros los objetos de la clase B, de forma que cada método `run()` de cada objeto de clase B 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, modifica el código para que solo un hilo pueda estar ejecutando el método `EnterAndWait()` en cualquier instante. Explica lo que observas.



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

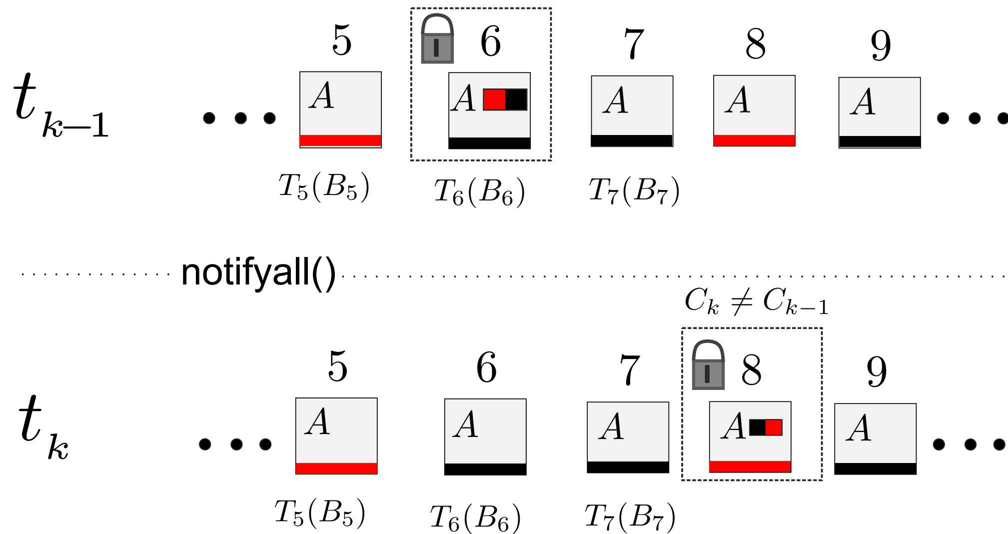
   Objetivo: El propósito de este ejercicio es comprender cómo controlar el acceso de hilos a *secciones críticas*. En el problema 1, el método sincronizado limitó el acceso a una sección crítica independientemente de cualquier otra condición. Mientras que todos los hilos compiten continuamente para entrar, es el sistema operativo el que finalmente decide cual es el siguiente hilo. Aquí queremos controlar ese acceso de forma explícita dadas ciertas condiciones mediante el mecanismo `wait`/`notifyAll` de Java.

   a) **Exclusión condicional:** La condición para este ejercicio es que solo un hilo puede ejecutar la sección crítica y una vez terminado, el siguiente hilo no puede ser el mismo hilo. Reutiliza el código del problema anterior con las modificaciones:

      1) **Modifica Class A:** agrega dos atributos enteros: `Nactual`, un contador que indica el número de veces que se ha entrado en `EnterAndWait()`, y `nhUltimo`, que indica el último hilo que ha accedido a la sección crítica (utilizado para garantizar que el siguiente hilo sea cualquier menos `nhUltimo`, y que luego debe actualizarse con el hilo actual que ingresa). `Nactual` debe comenzar en un valor inicial y disminuir a cero, lo que indica el final del programa.
      2) **Modifica Class B:** Modifica la sección de ejecución del subproceso para implementar la condición de exclusión condicional con `wait`/`notifyAll`. Los hilos que no están en la sección crítica, están en un estado de espera. Agrega cualquier otra variable o código necesario.
      3) **Modifica el programa principal:** Modifica el código del main de manera apropiada. Ejecuta el código con `M` hilos y `N` accesos totales a la sección crítica de `EnterAndWait()`; por ejemplo, `N=20` veces, mientras que `M=6` hilos. Comprueba que funciona como se espera.

   b) **Exclusión condicional con rojo/negro alternando:** En este problema, tu programa debería controlar los hilos basándose en una condición adicional: su color.

      1) Asigna un atributo de `Color` (C) (tipo `String`) a cada hilo (la clase B), que puede tomar el valor de ‘rojo’ o ‘negro’.
      2) Modifica el código del main para asignar a cada hilo un color, elegido al azar.
      3) Modifica Class A y Class B de manera apropiada. Para seleccionar el siguiente hilo (después de una llamada a `notifyAll()`), la regla es 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). Esto se ilustra en la siguiente figura, donde una posible solución es almacenar el atributo de color *anterior* y *actual* en la variable compartida A, que se utiliza para comparar quién puede ingresar a la sección crítica.

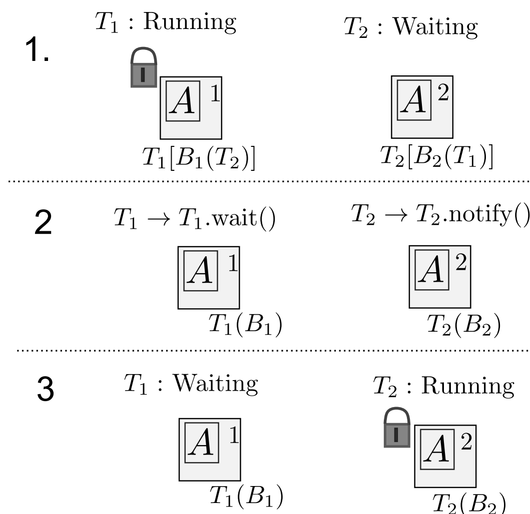


6. Semana 6 (19/03–25/03): Sincronización

La semana pasada, se probó como usar el mecanismo de `wait()/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, o 2) directamente usando una referencia al siguiente hilo y despertando solo este hilo mediante `notify()` para que sea el siguiente en acceder.
En el primer problema, se aprenderá cómo hacer referencia a un hilo directamente desde otro. En el segundo problema, se volverá al código `EnterAndWait()` de la semana pasada para controlar el orden de los hilos que entran en la sección crítica.

1. (**P4: para entregar en grupo de práctica**):

   Objetivo: este ejercicio pretende reiterar el concepto de compartir objetos entre hilos y utilizar la referencia del objeto compartido para controlar el acceso de los hilos a las secciones críticas. Consulta la Figura 1 a continuación:



a) **Configuración del problema:** Comienza usando el código de la semana pasada para la sincronización de `EnterAndWait()`. Ahora crea una clase principal (por ejemplo, `MiProblema`) donde, dentro de su método `main`, se creen dos threads t_1 y t_2. El hilo debe tener un método `set()` para que se pueda establecer la referencia de uno al otro.

b) **Sincronizar:** Tal y como se muestra en la secuencia de estados de la Figura 1, escribe el código apropiado, de tal forma que se *alternará* el estado de ejecución y espera entre los hilos t_1 y t_2. Después de que un hilo ejecuta la sección crítica (`EnterAndWait()`), debe notificar explícitamente al siguiente hilo.

c) **Ejecución e interrupción del programa:** Tu código debería ejecutarse en un bucle infinito. Añade el código necesario para que se interrumpan los hilos y así detener el programa.

d) **Lanzando un tercer hilo:** Modifica el código para crear e iniciar un tercer hilo t_3. La operación de alternar entre hilos solo debería tener lugar entre los dos primeros como antes. ¿Puedes observar que t_3 nunca ingresa en la sección crítica?

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:

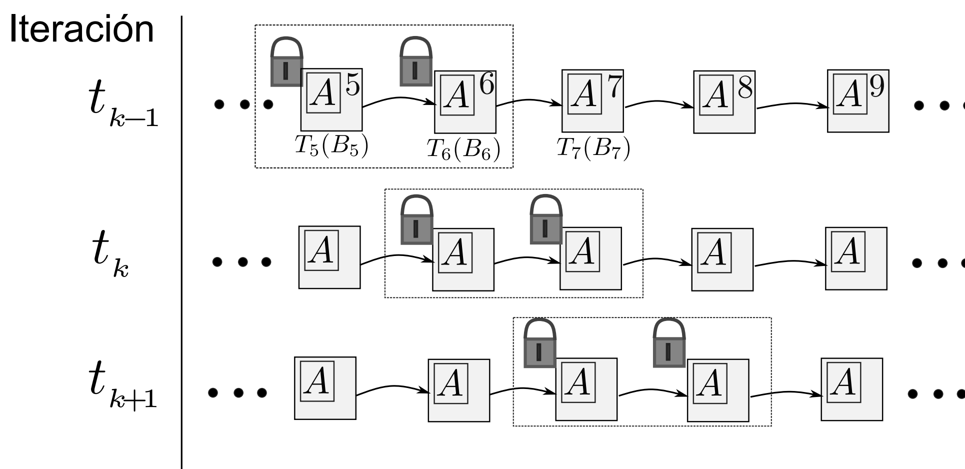
Sigue estos pasos:

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()`.

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. Semana 7 (02/04–08/04): 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 aumentar el valor del búfer (utilizando el método `write()`) y el consumidor debe disminuir el valor (utilizando el método `read()`).
      3) La variable compartida tiene una capacidad máxima que el valor no puede exceder (valor entero `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) La variable compartida tiene una capacidad mínima que el valor no puede aprobar (por ejemplo, puede tener el valor cero). 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 `Priority BlockingQueue`.

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

8. Semana 8 (09/04–15/04): *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)**: Semáforos.

   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.

```java
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)**: Estudio de dividir tareas concurrentes: dividir tareas en una matriz utilizando `Executor` y `ThreadPool`

Cuando trabajamos con matrices (Semana 3), el diseño parecía ideal. El programa era altamente paralelo, y los hilos ni siquiera tenían que sincronizarse. En la práctica, sin embargo, aunque este diseño podría funcionar bien para matrices pequeñas, tendría un rendimiento muy bajo para matrices lo suficientemente grandes como para ser interesante. La razón es que los hilos requieren memoria para el *stack* y otra información. Crear, programar y destruir hilos es costoso. Crear muchos hilos de corta duración es una forma ineficaz de organizar un cálculo multiproceso. Una forma más efectiva es con `ThreadPools`.

Java tiene servicios especializados e interfaces para ejecutar grupos de hilos de manera eficiente. Consulta la documentación provista en Faitic o de libros sobre este tema.

a) Escribe una rutina principal para usar el código asociado en este ejemplo, que implemente la adición y multiplicación de la matriz con el servicio de `Executor`.
- Explica cómo se dividen las tareas en este código. ¿Qué hace el objeto `Future` en este código?
- Haz una gráfica del tiempo de ejecución como una función de la dimensión de la matriz. Compara esta gráfica con la de un programa secuencial para la multiplicación de matrices.
- ¿Cuál es la limitación del método de división proporcionado? ¿Cómo podría mejorarse y generalizarse?

b) Modifica el código de la semana 3, problema 2, para usar el servicio de `Executor` de forma similar al que se usa en el código proporcionado para la suma y la multiplicación.

La clase `Matrix`:

```
import java.util.Random;

public class Matrix {
  double[][] data;
  int dim;
  int rowDisplace;
  int colDisplace;

  public Matrix(int d) {
    dim=d;
    data=new double[dim][dim];
  }
  private Matrix(double[][] matrix, int x, int y, int d) {
    data=matrix;
    dim=d;
    rowDisplace=x;
    colDisplace=y;
  }

  public void initialize() {
    Random rand=new Random();
    for(int i=0; i<dim; ++i){
      for(int j=0; j<dim; ++j){
        data[i][j]=(double)rand.nextInt(10);
      }
    }
  }

  public double get(int row, int col) {
```

```
    return data[row+rowDisplace][col+colDisplace];
  }
  public void set(int row, int col, double value) {
    data[row+rowDisplace][col+colDisplace]=value;
  }
  public int getDim() {
    return dim;
  }

  Matrix[][] split() {
    Matrix[][] result=new Matrix[2][2];
    final int newDim=dim/2;
    result[0][0]=
      new Matrix(data,rowDisplace,colDisplace,newDim);
    result[0][1]=
      new Matrix(data,rowDisplace,colDisplace+newDim,newDim);
    result[1][0]=
      new Matrix(data,rowDisplace+newDim,colDisplace,newDim);
    result[1][1]=
      new Matrix(data,rowDisplace+newDim,colDisplace+newDim,newDim);
    return result;
  }

  public String toString(){
    String ret="";
    for(int i=0; i<dim; ++i){
      for(int j=0; j<dim; ++j)
      ret+=data[i][j]+" ";
      ret+="\n";
    }
    return ret;
  }
}
```

La clase `MatrixTask`:

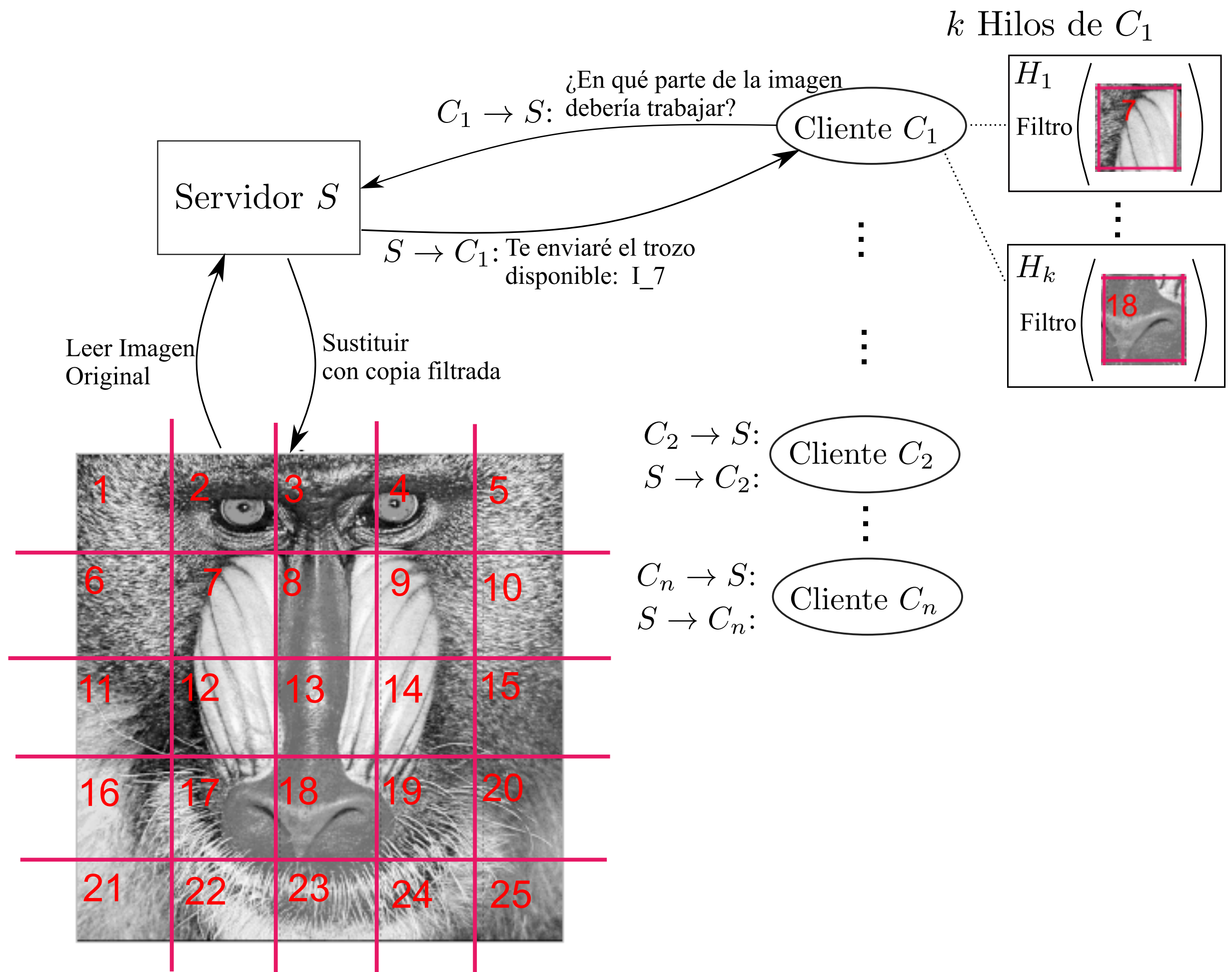
```
import java.util.concurrent.atomic.AtomicIntegerArray;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class MatrixTask {
  static ExecutorService exec=Executors.newCachedThreadPool();

  static Matrix add(Matrix a, Matrix b) throws ExecutionException {
    final int n=a.getDim();
    final Matrix c=new Matrix(n);
    Future<?> future=exec.submit(new AddTask(a,b,c));
    try {
      future.get();
```

```
    }
    catch(InterruptedException e) {
      e.printStackTrace();
    }
    catch(ExecutionException e) {
      e.printStackTrace();
    }

    // Finish the executor
    exec.shutdown();
    System.out.println("Exec: No more tasks");
    try {
      TimeUnit.SECONDS.sleep(1);
    }
    catch(InterruptedException e) {
      e.printStackTrace();
    }
    // The example finish
    System.out.println("Exec: return value");
    return c;
  }

  static class AddTask implements Runnable {
    Matrix a;
    Matrix b;
    Matrix c;

    public AddTask(Matrix a, Matrix b, Matrix c) {
      this.a=a;
      this.b=b;
      this.c=c;
    }
    public void run() {
      try {
        final int n=a.getDim();
        if(n==1) {
          c.set(0,0,a.get(0,0)+b.get(0,0));
        }
        else {
          final Matrix[][] aa=a.split();
          final Matrix[][] bb=b.split();
          final Matrix[][] cc=c.split();
          final Future<?>[][] future=(Future<?>[][]) new Future[2][2];
          for(int i=0; i<2; ++i)
            for(int j=0; j<2; ++j)
              future[i][j] =
                exec.submit(new AddTask(aa[i][j],bb[i][j],cc[i][j]));
          for(int i=0; i<2; ++i)
            for(int j=0; j<2; ++j)
              future[i][j].get();
        }
      }
      catch(Exception E) {
```

```
        E.printStackTrace();
      }
    }
  }

  static class MulTask implements Runnable {
    Matrix a;
    Matrix b;
    Matrix c;
    Matrix lhs;
    Matrix rhs;
    public MulTask(Matrix a, Matrix b, Matrix c) {
      this.a=a;
      this.b=b;
      this.c=c;
      this.lhs=new Matrix(a.getDim());
      this.rhs=new Matrix(a.getDim());
    }
    public void run() {
      try {
        if(a.getDim()==1) {
          c.set(0,0,a.get(0,0)*b.get(0,0));
        }
        else {
          final Matrix[][] aa=a.split();
          final Matrix[][] bb=b.split();
          final Matrix[][] cc=c.split();
          final Matrix[][] ll=lhs.split();
          final Matrix[][] rr=rhs.split();
          final Future<?>[][][] future=(Future<?>[][][])new Future[2][2][2];
          for(int i=0; i<2; ++i)
            for(int j=0; j<2; ++j) {
              future[i][j][0]=
                exec.submit(new MulTask(aa[i][0],bb[0][i],ll[i][j]));
              future[i][j][1]=
                exec.submit(new MulTask(aa[1][i],bb[i][1],rr[i][j]));
            }
          for(int i=0; i<2; ++i)
            for(int j=0; j<2; ++j)
              for(int k=0; k<2; ++k)
                future[i][j][k].get();
                Future<?> done=exec.submit(new AddTask(lhs,rhs,c));
                done.get();
        }
      }
      catch(Exception E) {
        E.printStackTrace();
      }
    }
  }
}
```

9. Semana 9 (16/04–22/04) *Sockets* y *streams*

Objetivos: Implementar una aplicación distribuida con *sockets* y *streams*



1. **(P4 y P3: para entregar en grupo de práctica y para entregar dentro de una semana)**:
   Objetivo: Retomamos la actividad 3 donde implementamos concurrencia simple para trabajar con una matriz, pero esta vez es una imagen. En vez de lanzar hilos en la misma máquina queremos usar o bien otros procesos en el mismo ordenador o bien otros procesos en otro(s) ordenador(es). Para ello, vamos a implementar un sistema cliente-servidor.

   - El servidor trabaja con una estructura de datos para las tareas.
   - Las tareas se distribuyen a los clientes cuando estos realizan peticiones.
   - Con los resultados recibidos como respuestas de los clientes el servidor compone la imagen resultante después de operaciones de suavización con el filtro. (Para entender más información sobre filtros de imagen del ver, por ejemplo, wikipedia)

   Configuración del problema:

   a) **Arquitectura:** Consultar la figura que demuestra los componentes y la arquitectura de esta práctica donde el Servidor/Cliente está actuando como un sistema *Master/Slaves*

b) **Conectividad entre Servidor y Clientes** (Master/Slaves): Implementa dos clases, `Cliente` y `Servidor` que contengan conexiones como las del ejemplo del anexo de esta práctica.

1) Como en el ejemplo, la comunicación entre servidor y clientes se realiza con sockets y flujos sobre estas conexiones tipo `ObjectOutputStream` y `ObjectInputStream`.
2) Copia y ejecuta el código del Anexo 1. Fíjate en la conexión con sockets y la utilización de `ObjectOutputStream` y `ObjectInputStream`.

c) **Servidor:** El servidor se ocupa de la lectura y escritura de la imagen (ver código en anexo para leer imágenes).

1) Se transmite un objeto desde el servidor al cliente para indicar los parámetros para calcular (básicamente la región de la imagen).
2) El objeto que se transmite al cliente contiene los parámetros de la región sobre la cual este cliente debe actuar, es decir, coordenadas, zona de la imagen, y umbral.

d) **Cliente:** Para cada cliente que se conecta al servidor, se usa un hilo (en el servidor) para realizar la gestión, es decir, para mandar la tarea y esperar la respuesta.

1) En vez de crear los hilos trabajadores en un bucle fijo, se espera para que se conecten los clientes y se crea para cada cliente el hilo que se dedica a la comunicación.
2) Se devuelve un objeto desde el cliente al servidor con el resultado.
3) Tarea de *worker* hilos: Como antes, cada hilo (de clase `Cliente`), realiza una suavización de la imagen con la fórmula dada en Semana 3, problema 2, pero esta vez de forma distribuida. La fórmula por implementar (como antes) es:

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

4) El objeto que se transmite al servidor contiene en la región los valores del resultado de la operación, que el hilo de comunicación (del servidor) copia a la imagen de resultado.

e) **Sincronización final:** Para realizar la sincronización al final, es decir, que el servidor determina que todos los hilos trabajadores y sus respectivos clientes, hayan terminado se usa un `CountDownLatch` de forma adecuada. (Recomendación: usa el `CountDownLatch` primero en una modificación de la actividad 2 para sustituir el uso de `join`.)

```java
import java.net.*;
import java.io.*;

class MySockets {
  public static void main(String args[]) {
    new Server().start();
    new Client().start();
  }
}

class Server extends Thread {
  Socket socket;
  ObjectInputStream ois;
  ObjectOutputStream oos;

  public void run() {
```

```
    try {
      ServerSocket server=new ServerSocket(4444);
      while(true) {
        socket=server.accept();
        ois=new ObjectInputStream(socket.getInputStream());
        String message=(String) ois.readObject();
        System.out.println("Server Received: "+message);
        oos=new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject("Server Reply");
        ois.close();
        oos.close();
        socket.close();
      }
    }
    catch(Exception e) {
    }
  }
}

class Client extends Thread {
  InetAddress host;
  Socket socket;
  ObjectOutputStream oos;
  ObjectInputStream ois;

  public void run() {
    try {
      for(int x=0; x<5; ++x) {
        host=InetAddress.getLocalHost();
        socket=new Socket(host.getHostName(), 4444);
        oos=new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject("Client Message "+x);
        ois=new ObjectInputStream(socket.getInputStream());
        String message=(String) ois.readObject();
        System.out.println("Client Received: "+message);
        ois.close();
        oos.close();
        socket.close();
      }
    }
    catch(Exception e) {
    }
  }
}
```

```
import java.awt.image.*;
import java.io.File;
import javax.imageio.*;

class Gray {
  static BufferedImage img;

  public static void main(String[] args ) {
    try {
      File file_in=new File("image.png");
      img=ImageIO.read(file_in);

      final int width=img.getWidth();
      final int height=img.getHeight();
      int[][] data=new int[width][height];
      Raster raster_in=img.getData();
      for(int i=0; i<width; ++i) {
        for(int j=0; j<height; ++j) {
          final int d=raster_in.getSample(i, j, 0);
          data[i][j]=d;
        }
      }
      WritableRaster raster_out=img.getRaster();
      for(int i=0; i<width; ++i) {
        for(int j=0; j<height; ++j) {
          raster_out.setSample(i,j,0,data[i][j]/2);
        }
      }
      img.setData(raster_out);
      File file_out=new File("out.png");
      ImageIO.write(img,"png",file_out);
    }
    catch(Exception E) {
    }
  }
}
```

10. Semana 10 (23/04–29/04) Conjuntos de Mandelbrot distribuidos

Objetivos: Calcular un conjunto de Mandelbrot en un orden específico utilizando métodos concurrentes y distribuidos con una nube de ordenadores organizados como una arquitectura cliente-servidor (con *sockets* como la semana pasada)

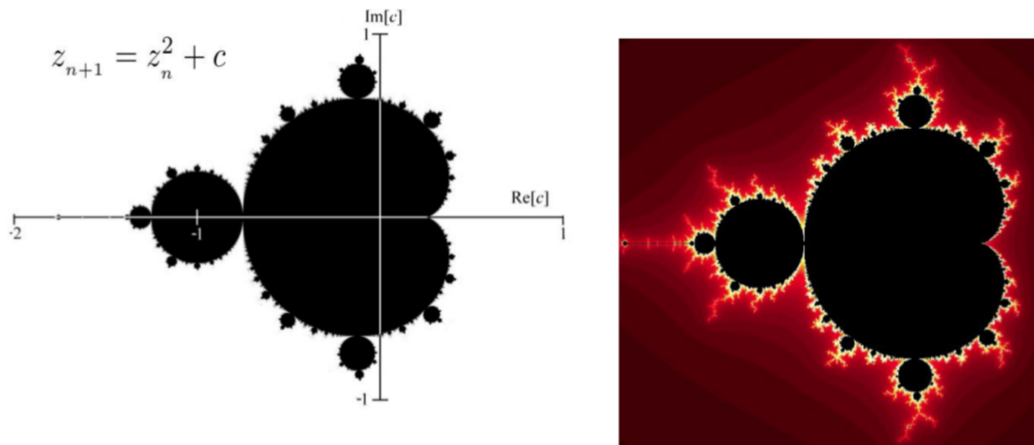
1. **(P4 y P3: para entregar en grupo de práctica y para entregar dentro de una semana)**:

   ¿Qué es un conjunto de Mandelbrot? El conjunto de Mandelbrot es el más conocido de los conjuntos fractales y el más estudiado. El conjunto de Mandelbrot es el conjunto de valores de c en el plano complejo para el cual la órbita de 0 bajo la iteración del mapa cuadrático

   $$z_{n+1} = z_n^2 + c$$

   queda acotado. Si esta sucesión queda acotada, entonces se dice que c pertenece al conjunto de Mandelbrot, y si no, queda excluido del mismo.

Ver información en `https://en.wikipedia.org/wiki/Mandelbrot_set`



¿Qué significa para una serie queda acotoda? Considera estos ejemplos:

- Por ejemplo, si $c = 1$ obtenemos la sucesión $0, 1, 2, 5, 26 \cdots$ que diverge. Como no está acotada, 1 no es un elemento del conjunto de Mandelbrot.
- En cambio, si $c = -1$ obtenemos la sucesión $0, -1, 0, -1, \cdots$ que sí es acotada, y por tanto, -1 sí pertenece al conjunto de Mandelbrot.

Sobre la figura:

- Se representa el conjunto mediante el algoritmo de tiempo de escape: los colores de los puntos que no pertenecen al conjunto indican la velocidad con la que diverge (tienden al infinito, en módulo) la sucesión correspondiente a dicho punto.
- el rojo oscuro indica que a cabo de pocos cálculos el punto no está en el conjunto; mientras que el blanco indica que se ha tardado mucho más para que diverja. Más detalles del algoritmo están dado en el anexo. En modo más sencillo, bastaría generar ficheros en blanco y negro, donde por ejemplo un píxel negro indica que las coordenadas correspondientes pertenecen al conjunto (la sucesión queda acotada), un píxel blanco que no.

Configuración del problema:

a) **Arquitectura** de Codigo Server/cliente de la semana pasada: el servidor esté a la espera de recibir peticiones de trabajo de clientes. Como en el código de la semana pasada, el cliente le pedirá al servidor bloques para calcular. Considera en el uso del patrón de diseño productor/consumidor para la implementación del control del flujo de datos.

b) **Cliente:** Cuando un cliente pide trabajo al servidor, el servidor asigna a tal cliente un bloque de la imagen de tamaño adecuado para que el cliente calcule la parte correspondiente de la imagen, es decir, el cliente devuelve los píxeles calculados según la especificación del servidor (coordinadas de la ventana y parámetros necesarios)

c) **Servidor:** El servidor colecciona todos los datos devueltos por los clientes hasta que la imagen esté completada, en cual momento escribe el fichero resultante.

d) Al lanzar el servidor se debe especificar por lo menos la región del conjunto Mandelbrot por calcular, la resolución de la misma, el número máximo de iteraciones, y el nombre del fichero de salida.

e) Al lanzar el cliente se debe especificar por lo menos los datos necesarios para contactar al servidor.

f) (Mejora) Una vez funcionando el sistema de cálculo en sí, intenta modificar el sistema para que sea robusto a fallos en un cliente, es decir, si un cliente todavía no ha devuelto el resultado y llega una nueva petición de trabajo, el cálculo faltante se puede asignar otra vez a tal nueva petición.

g) Construyendo diferentes niveles de conjuntos de Mandelbrot. Para cada punto $c = (x, y) = (\mathrm{Re}[c], \mathrm{Im}[c])$ en el plano complejo, el hilo debe determinar si la relación de recurrencia permanece finita o diverge a infinito. En el caso más simple, si la recurrencia permanece acotada, entonces el punto c se pinta de negro; de lo contrario, se pinta de blanco. Para casos más complejos, la escala de color puede indicar el ritmo al que se repite.

h) **Rendimiento:** ejecuta tu código para diferentes niveles de zoom. Con tu código, ¿cuánto tiempo tomaría calcular la profundidad de un conjunto de Mandelbrot con una resolución de 10^{-220} dando pasos de 10^{-1}?