**Clase del día - 26/02/2021**

La clase anterior vimos el programa [**Servidor.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1674) el cual invoca el método **accept** para esperar una conexión del cliente, debido a que este método es bloqueante el programa queda en espera pasiva hasta que el cliente se conecta.

Cuando el servidor recibe una conexión, el método **accept** regresa un socket. Entonces el cliente y el servidor podrán intercambiar datos. Generalmente el servidor procesa los datos que recibe del cliente y al terminar vuelve a invocar el método **accept** para esperar otra conexión.

Sin embargo, mientras el servidor procesa los datos que recibe del cliente, no puede recibir otra conexión. Para resolver este problema los servidores se construyen utilizando threads.

En la clase de hoy veremos cómo construir un servidor multithread.

**Programación multithread en Java**

Supongamos que tenemos una clase principal llamada **P**.

Dentro de la clase **P** definimos una clase interior (*nested class*) llamada **Worker** la cual es subclase de la clase Thread:

class P

{

static class Worker extends Thread

{

public void run()

{

}

}

public static void main(String[] args) throws Exception

{

}

}

Podemos ver que hemos incluido en la clase **Worker** un método público llamado **run** el cual no tiene parámetros ni regresa un resultado.

En el curso de Sistemas Operativos se explicó que un thread (hilo) es una secuencia de instrucciones que ejecutan en una computadora. Si la computadora tiene un CPU *dual core*, entonces el CPU podrá ejecutar en paralelo (al mismo tiempo) dos threads, si el CPU es *quad core* entonces podrá ejecutar en paralelo cuatro threads, y así sucesivamente.

Por otra parte, si un programa crea un número de threads mayor al número de procesadores físicos (*cores*) disponibles en la computadora, entonces los threads ejecutarán en forma concurrente (por turnos).

**Crear un thread e iniciar su ejecución**

Para iniciar la ejecución de un thread, debemos crear una instancia de la clase **Worker** e invocar el método **start** (este método se hereda de la clase Thread):

Worker w = new Worker();

w.start();

Entonces se crea un hilo que inicia invocando el método **run** que hemos definido en la clase **Worker**.

Un thread finaliza su ejecución cuando el método **run** termina. Cuando un thread finaliza, no puede volver a ejecutarse.

**El método join**

Supongamos que el thread principal (el thread que invocó el método **start**) requiere esperar que el thread w termine su ejecución, entonces el thread principal deberá invocar el método **join**:

Worker w = new Worker();  
w.start();  
w.join();

El método **join** queda en un estado de espera pasiva mientras el thread "w" se encuentra ejecutando, cuando el thread "w" termina, el método **join** regresa, entonces el thread principal continua su ejecución.

Ahora supongamos que el thread principal requiere crear dos threads y esperar a que terminen su ejecución. Entonces creamos dos instancias de la clase **Worker** e invocamos los métodos **start** y **join** para cada thread:

Worker w1 = new Worker();  
Worker w2 = new Worker();

w1.start();  
w2.start();  
w1.join();  
w2.join();

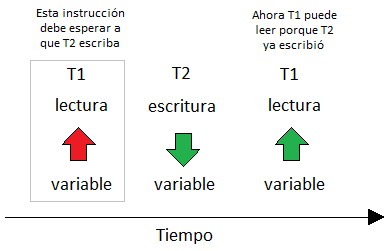
Cuando un thread (en este caso el thread principal) espera la terminación de uno o más threads para continuar su ejecución, se dice que se implementa una **barrera**. En este caso estamos implementando una barrera mediante dos métodos **join**.

**Sincronización de threads**

Cuándo dos o más threads acceden a una misma variable, y al menos uno de los threads modifica (escribe) la variable, entonces es necesario sincronizar el acceso de los threads a la variable.

Sincronizar el acceso de los threads significa ordenar las operaciones de escritura y de lectura que realizan los threads.

Por ejemplo, si el thread T1 va a leer una variable que escribe el thread T2, entonces el thread T1 debe esperar a que el thread T2 escriba la variable; evidentemente el thread T1 no puede leer un valor que no ha sido escrito todavía.



Para ordenar las lecturas y escrituras que realizan los threads dentro de un proceso, se utiliza la instrucción **synchronized**:

synchronized(*objeto*)  
{  
 *instrucciones*  
}

La instrucción **synchronized** funciona de la siguiente manera:

* Primero se verifica si el lock del *objeto* está bloqueado (en Java todos los objetos tienen un lock asociado), si el lock está bloqueado entonces el thread espera a que el lock se desbloquee.
* Por otra parte, si el lock está desbloqueado entonces el thread lo bloquea (se dice que "el thread adquiere el lock") y ejecuta las instrucciones dentro del bloque.
* Al terminar de ejecutar las instrucciones el thread desbloquea el lock (se dice que "el thread libera el lock"), entonces el sistema operativo notifica a alguno de los threads que se encuentran esperando el lock  para que adquiera el lock y ejecute las instrucciones dentro del bloque.
* Al terminar de ejecutar las instrucciones, el thread desbloquea el lock  y nuevamente el sistema operativo notifica a alguno de los threads que esperan.

Como podemos ver, la instrucción **synchronized** evita que dos o más threads ejecuten simultáneamente un bloque de instrucciones. Al bloque de instrucciones que solo puede ser ejecutado por un thread se le llama **sección crítica**.

Veamos un ejemplo.

Supongamos que tenemos dos threads que incrementan una variable estática llamada "n" dentro de un ciclo for. La variable estática "n" es "global" a todas las instancias de la clase, por tanto los threads pueden leer y escribir esta variable.

class A extends Thread  
{  
  static long n;  
  public void run()  
  {  
    for (int i = 0; i < 100000; i++)  
        n++;  
  }  
  public static void main(String[] args) throws Exception  
  {  
    A t1 = new A();  
    A t2 = new A();  
    t1.start();  
    t2.start();  
    t1.join();  
    t2.join();  
    System.out.println(n);  
  }  
}

En este caso, la clase principal A es subclase de la clase Thread, por tanto hereda los métodos run, start y join (entre otros).

El programa debería desplegar 200000 ya que cada thread incrementa 100000 veces la variable "n".

¿Por qué despliega un número menor a 200000?

¿Por qué cada vez que se ejecuta el programa despliega un número diferente?

El problema es que los dos threads ejecutan al mismo tiempo la instrucción n++.

El incremento de la variable se compone de tres operaciones: la lectura a la variable, el incremento del valor y la escritura del nuevo valor. Sin embargo, los dos threads ejecutan al mismo tiempo las instrucciones de lectura y escritura sobre la misma variable, lo cual ocasiona que algunos incrementos "se pierdan" (no se escriban sobre la variable "n").

Entonces debemos impedir que ambos threads ejecuten al mismo tiempo la instrucción n++. Justamente esta instrucción es la sección crítica.

Ahora vamos a ejecutar la instrucción n++ dentro de una instrucción synchronized, Notar que utilizamos el objeto "obj" para sincronizar los threads:

class A extends Thread  
{  
  static long n;  
  static Object obj = new Object();  
  public void run()  
  {  
    for (int i = 0; i < 100000; i++)  
**synchronized(obj)  
    {  
      n++;  
    }**  }  
  public static void main(String[] args) throws Exception  
  {  
    A t1 = new A();  
    A t2 = new A();  
    t1.start();  
    t2.start();  
    t1.join();  
    t2.join();  
    System.out.println(n);  
  }  
}

En este caso el programa siempre despliega 200000.

Si bien es cierto que es necesario sincronizar los threads para que el programa funcione correctamente, la sincronización hace más lento el programa, ya que obliga a que ciertas partes del programa se ejecuten en serie (una tras otra) y no en paralelo (al mismo tiempo).

[**Servidor2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1676)

Ahora vamos a implementar el servidor de sockets multithread.

La idea es que el servidor multithread espere conexiones y para cada conexión cree un thread que procese los datos que envía el cliente.

Vamos a invocar el método **accept** dentro de un ciclo, y para cada conexión vamos a crear un thread.

class Servidor2

{

static class Worker extends Thread

{

Socket conexion;

Worker(Socket conexion)

{

this.conexion = conexion;

}

public void run()

{

}

}

public static void main(String[] args) throws Exception

{

ServerSocket servidor = new ServerSocket(50000);

for (;;)

{

Socket conexion = servidor.accept();

Worker w = new Worker(conexion);

w.start();

}

}

}

Este código será la base para los programas que desarrollaremos en el curso.

Ahora el constructor de la clase **Worker** pasa como parámetro el socket que crea el método **accept**, ya que el método **run** requiere el socket para recibir y enviar datos al cliente.

La implementación completa del servidor se puede encontrar en el programa [**Servidor2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1676).

Podemos ver en el programa [**Servidor2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1676) que el método **run** crea los streams que se utilizarán para enviar y recibir datos del cliente. Notar que el programa [**Servidor2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1676) es completamente compatible con el programa [**Cliente.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1673)

**Un cliente con re-intentos de conexión**

Como vimos la clase pasada, para que el cliente se conecte al servidor, es necesario que el servidor inicie su ejecución antes que el cliente, sin embargo para algunas aplicaciones el cliente debe esperar a que el servidor inicie su ejecución.

En el programa [**Cliente2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1675) podemos ver cómo implementar el re-intento de conexión cuando el servidor no está ejecutando.

Socket conexion = null;

for(;;)

try

{

conexion = new Socket("localhost",50000);

break;

}

catch (Exception e)

{

Thread.sleep(100);

}

Como podemos ver, cada vez que el cliente falla en establecer la conexión con el servidor, espera 100 milisegundos y vuelve a intentar la conexión. Cuando el cliente logra conectarse con el servidor entonces sale del ciclo for.

**Actividades individuales a realizar**

1. Compile los programas [**Cliente2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1675) y [**Servidor2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1676)
2. Ejecute el programa [**Cliente2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1675) en una ventana de comandos de Windows (o terminal de Linux) y ejecute el programa [**Servidor2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1676) en otra ventana de comandos de Windows (o terminal de Linux).
3. Ejecute repetidamente el programa [**Cliente2.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1675) en la ventana de comandos, como puede ver el servidor sigue en ejecución recibiendo las conexiones de los clientes y procesando los datos.
4. Compile la clase A que no utiliza sincronización y la clase que utiliza sincronización
   * ¿Por qué el programa sin sincronización despliega un valor incorrecto?
   * ¿Por qué cada vez que se ejecuta el programa sin sincronización despliega un valor diferente?
   * ¿Por qué el programa con sincronización es más lento?