25-3-2021

Actividad 4 - Programas en Java [Concurrencia]

Jerez de García Salinas

Alumno:
Mario Alberto Loya Rodríguez
Carrera:
Ingeniería en Sistemas Computacionales
Semestre 4

Materia:
Tópicos Avanzados de Programación
Tema:
3.- Programación concurrente (MultiHilos)

No. de control:
16070135
Profesor:
ISC Salvador Acevedo Sandoval

Ciclo de vida y estados de un hilo en Java

Existe un hilo en Java en cualquier punto del tiempo en cualquiera de los siguientes estados. Un hilo se encuentra solo en uno de los estados mostrados en cualquier instante: 1.- Nuevo

2.- Runnable

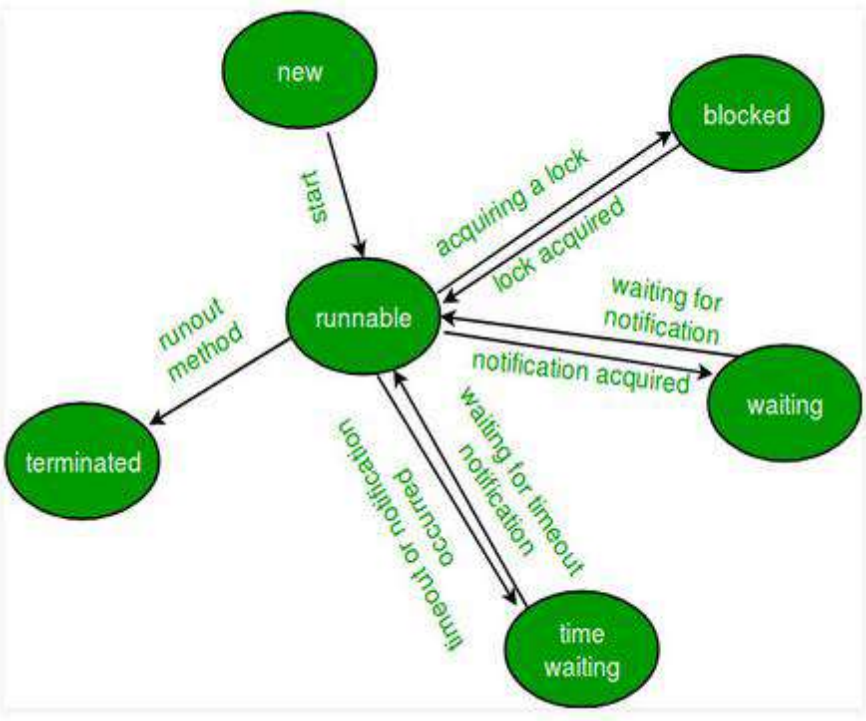
3.- Bloqueado

4.- Esperando

5.- Tiempo de espera

6.- Terminado

El diagrama que se muestra a continuación representa varios estados de un hilo en cualquier instante de tiempo.



Ciclo de vida de un hilo

1.- Nuevo subproceso: cuando se crea un nuevo subproceso, está en el nuevo estado. El subproceso aún no se ha iniciado cuando el subproceso está en este estado. Cuando un hilo se encuentra en el nuevo estado, su código aún no se ha ejecutado y no ha comenzado a ejecutarse.

2.- Estado ejecutable: un hilo que está listo para ejecutarse se mueve al estado ejecutable. En este estado, un hilo podría estar ejecutándose o podría estar listo para ejecutarse en cualquier momento. Es responsabilidad del programador de hilos dar el hilo y el tiempo de ejecución.

Un programa de subprocesos múltiples asigna una cantidad de tiempo fija a cada subproceso individual. Todos y cada uno de los subprocesos se ejecutan durante un breve período de tiempo y luego hacen una pausa y renuncian a la CPU a otro subproceso, para que otros subprocesos tengan la oportunidad de ejecutarse. Cuando esto sucede, todos los subprocesos que están listos para ejecutarse, a la espera de que la CPU y el subproceso en ejecución se encuentren en estado ejecutable. 3.- Estado bloqueado / en espera: cuando un hilo está temporalmente inactivo, está en uno de los siguientes estados:

* Bloqueado

* Esperando

Por ejemplo, cuando un hilo está esperando a que se complete la E / S, se encuentra en estado bloqueado. Es responsabilidad del programador de hilos reactivar y programar un hilo bloqueado / en espera. Un hilo en este estado no puede continuar su ejecución más hasta que se mueva a estado ejecutable. Cualquier hilo en estos estados no consume ningún ciclo de CPU.

Un hilo está en el estado bloqueado cuando intenta acceder a una sección protegida de código que actualmente está bloqueada por algún otro hilo. Cuando la sección protegida está desbloqueada, el programa selecciona uno de los hilos bloqueados para esa sección y lo mueve al estado ejecutable. Mientras que, un hilo está en estado de espera cuando espera otro hilo en una condición. Cuando se cumple esta condición, se notifica al planificador y el hilo de espera se mueve a estado ejecutable.

Si un hilo actualmente en ejecución se mueve al estado bloqueado / en espera, otro hilo en el estado ejecutable es programado por el programador de hilos para ejecutarse. Es responsabilidad del planificador de hilos determinar qué hilo ejecutar.

4.- Espera temporizada: un hilo se encuentra en el estado de espera temporizada cuando llama a un método con un parámetro de tiempo de espera. Un hilo se encuentra en este estado hasta que se complete el tiempo de espera o hasta que se reciba una notificación. Por ejemplo, cuando un hilo llama a suspensión o espera condicional, se mueve al estado de espera temporizada.

5.- Estado terminado: un hilo termina por cualquiera de los siguientes motivos:

- Porque existe normalmente. Esto sucede cuando el código del hilo ha sido ejecutado por completo por el programa.

- Porque se produjo un evento erróneo inusual, como un error de segmentación o una excepción no controlada.

Un hilo que se encuentra en estado terminado ya no consume ningún ciclo de CPU.
Implementando Estados de subprocesos en Java

En Java, para obtener el estado actual del hilo, use el método Thread.getState () para obtener el estado actual del hilo. Java proporciona la clase java.lang.Thread.State que define las constantes ENUM para el estado de un hilo, como se resume a continuación:

1.- Tipo constante: Nuevo

```
Declaration: public static final Thread.State NEW
```

Descripción: estado del hilo para un hilo que aún no ha comenzado.

2.- Tipo constante: Runnable

```
Declaration: public static final Thread.State RUNNABLE
```

Descripción: estado del hilo para un hilo ejecutable. Se está ejecutando un subproceso en el estado ejecutable en la máquina virtual Java, pero puede estar esperando otros recursos del sistema operativo, como el procesador.

3.- Tipo constante: bloqueado

```
Declaration: public static final Thread.State BLOCKED
```

Descripción: estado del hilo para un hilo bloqueado esperando un bloqueo del monitor. Un hilo en el estado bloqueado está esperando que un bloqueo de monitor ingrese a un bloque / método sincronizado o vuelva a ingresar un bloque / método sincronizado después de llamar a Object.wait ().

4.- Tipo constante: esperando

```
Declaration: public static final Thread.State WAITING
```

Descripción: estado del hilo para un hilo de espera. Estado del subproceso para un hilo de espera. Un hilo está en estado de espera debido a que llama a uno de los siguientes métodos:

- Object.wait sin tiempo de espera

- Thread.join sin tiempo de espera

- LockSupport.park

Un hilo en el estado de espera está esperando que otro hilo realice una acción en particular.

5.- Tipo constante: tiempo de espera

```
Declaration: public static final Thread.State TIMED_WAITING
```

Descripción: estado del hilo para un hilo en espera con un tiempo de espera especificado. Un hilo está en estado de espera temporizada debido a que llama a uno de los siguientes métodos con un tiempo de espera positivo especificado:

- Thread.sleep

- Object.wait con timeout

- Thread.unirse con tiempo de espera

- LockSupport.parkNanos

- LockSupport.parkUntil

6.- Tipo constante: terminado

```
Declaration: public static final Thread.State TERMINATED
```

Descripción: estado del subproceso para un subproceso finalizado. El hilo ha completado la ejecución.

```
// Java program to demonstrate thread states
class thread implements Runnable
{
    public void run()
    {
        // moving thread2 to timed waiting state
        try
        {
            Thread.sleep(1500);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        try
        {
            Thread.sleep(1500);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println("State of thread1 while it called join() method on thread2 -"+
            Test.thread1.getState());
        try
        {
            Thread.sleep(200);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

```java
public class Test implements Runnable
{
    public static Thread thread1;
    public static Test obj;

    public static void main(String[] args)
    {
        obj = new Test();
        thread1 = new Thread(obj);

        // thread1 created and is currently in the NEW state.
        System.out.println("State of thread1 after creating it - " + thread1.getState());
        thread1.start();

        // thread1 moved to Runnable state
        System.out.println("State of thread1 after calling .start() method on it - " +
            thread1.getState());
    }

    public void run()
    {
        thread myThread = new thread();
        Thread thread2 = new Thread(myThread);

        // thread1 created and is currently in the NEW state.
        System.out.println("State of thread2 after creating it - "+ thread2.getState());
        thread2.start();

        // thread2 moved to Runnable state
        System.out.println("State of thread2 after calling .start() method on it - " +
            thread2.getState());

        // moving thread1 to timed waiting state
        try
        {
            //moving thread2 to timed waiting state
            Thread.sleep(200);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
```

```
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println("State of thread2 after calling .sleep() method on it - "+
            thread2.getState() );


        try
        {
            // waiting for thread2 to die
            thread2.join();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println("State of thread2 when it has finished it's execution - " +
            thread2.getState());
    }

}
```

Salida:

```
State of thread1 after creating it - NEW
State of thread1 after calling .start() method on it - RUNNABLE
State of thread2 after creating it - NEW
State of thread2 after calling .start() method on it - RUNNABLE
State of thread2 after calling .sleep() method on it - TIMED_WAITING
State of thread1 while it called join() method on thread2 -WAITING
State of thread2 when it has finished it's execution - TERMINATED
```

Explicación: cuando se crea un nuevo hilo, el hilo está en el estado NUEVO. Cuando se llama al método .start () en un subproceso, el planificador de subprocesos lo mueve a estado Runnable. Cuando se invoca el método join () en una instancia de subproceso, el subproceso actual que ejecuta esa instrucción esperará a que este subproceso pase al estado Terminated. Entonces, antes de que la declaración final se imprima en la consola, el programa llama a join () en thread2 haciendo que thread1 espere mientras thread2 completa su ejecución y se mueve al estado Terminated. thread1 pasa al estado de espera porque lo está esperando para que thread2 complete su ejecución como lo ha llamado join en thread2.

Referencias utilizadas: Oracle, Core Java Vol 1, novena edición, Horstmann, Cay S. y Cornell, Gary_2013

Este artículo es contribuido por Mayank Kumar. Si te gusta GeeksforGeeks y te gustaría contribuir, también puedes escribir un artículo usando contribute.geeksforgeeks.org o enviar tu artículo por correo electrónico a contribute@geeksforgeeks.org. Vea su artículo que aparece en la página principal de GeeksforGeeks y ayude a otros Geeks.

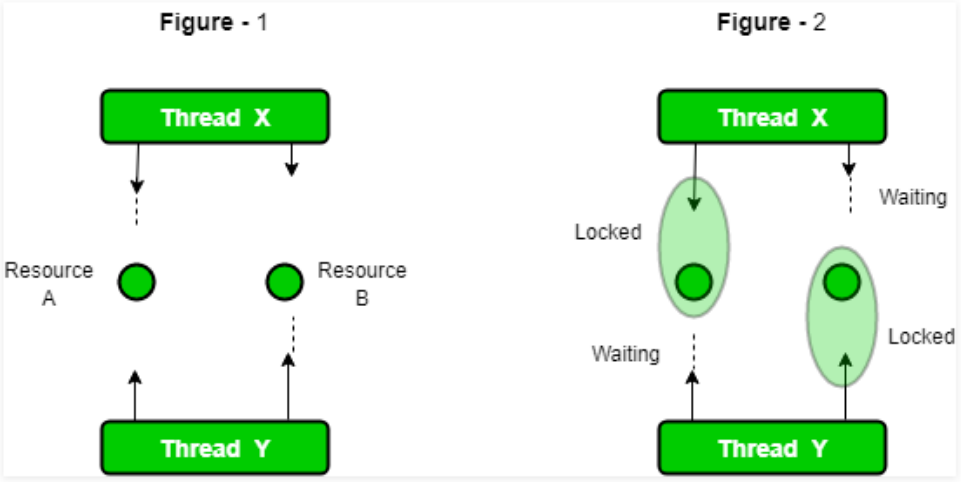
Por favor, escriba comentarios si encuentra algo incorrecto, o si desea compartir más información sobre el tema discutido anteriormente.

//PAGINA 1

Punto muerto en Java Multihilo

La palabra clave synchronized se usa para hacer que la clase o método sea seguro para subprocesos, lo que significa que solo un subproceso puede tener un métodosincronizado y usarlo, otros subprocesos tienen que esperar hasta que el bloqueo se libere y cualquiera de ellos adquiera ese bloqueo.

Es importante usarlo si nuestro programa se ejecuta en un entorno de subprocesos múltiples donde dos o más subprocesos se ejecutan simultáneamente. Pero a veces también causa un problema que se llama punto muerto. A continuación se muestra un ejemplo simple de condición de punto muerto.



```java
// Java program to illustrate Deadlock
// in multithreading.
class Util
{
    // Util class to sleep a thread
    static void sleep(long millis)
    {
        try
        {
            Thread.sleep(millis);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}


// This class is shared by both threads
class Shared
{
    // first synchronized method
    synchronized void test1(Shared s2)
    {
        System.out.println("test1-begin");
        Util.sleep(1000);

        // taking object lock of s2 enters
        // into test2 method
        s2.test2(this);
        System.out.println("test1-end");
    }

    // second synchronized method
    synchronized void test2(Shared s1)
    {
        System.out.println("test2-begin");
        Util.sleep(1000);

        // taking object lock of s1 enters
        // into test1 method
        s1.test1(this);
        System.out.println("test2-end");
    }
}
```

```
class Thread1 extends Thread
{
    private Shared s1;
    private Shared s2;

    // constructor to initialize fields
    public Thread1(Shared s1, Shared s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    // run method to start a thread
    @Override
    public void run()
    {
        // taking object lock of s1 enters
        // into test1 method
        s1.test1(s2);
    }
}


class Thread2 extends Thread
{
    private Shared s1;
    private Shared s2;

    // constructor to initialize fields
    public Thread2(Shared s1, Shared s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    // run method to start a thread
    @Override
    public void run()
    {
        // taking object lock of s2
        // enters into test2 method
        s2.test2(s1);
    }
}
```

```
public class GFG
{
    public static void main(String[] args)
    {
        // creating one object
        Shared s1 = new Shared();

        // creating second object
        Shared s2 = new Shared();

        // creating first thread and starting it
        Thread1 t1 = new Thread1(s1, s2);
        t1.start();

        // creating second thread and starting it
        Thread2 t2 = new Thread2(s1, s2);
        t2.start();

        // sleeping main thread
        Util.sleep(2000);
    }
}
```

```
Output : test1-begin
test2-begin
```

No se recomienda ejecutar el programa anterior con IDE en línea. Podemos copiar el código fuente y ejecutarlo en nuestra máquina local. Podemos ver que se ejecuta por tiempo indefinido, porque los hilos están en condición de punto muerto y no permite que se ejecute el código. Ahora veamos paso a paso lo que está sucediendo allí.

1.- El hilo t1 se inicia y llama al método test1 tomando el bloqueo del objeto de s1.

2.- El hilo t2 se inicia y llama al método test2 tomando el bloqueo del objeto de s2.

3.- t1 imprime test1-begin y t2 prints test-2 comienzan y ambos esperan 1 segundo, por lo que ambos hilos pueden iniciarse si alguno de ellos no lo está.

4.- t1 intenta tomar el bloqueo de objeto de s2 y llama al método test2, pero ya lo adquirió t2, por lo que espera a que se libere. No liberará el bloqueo de s1 hasta que consiga el bloqueo de s2.

5.- Lo mismo pasa con t2. Intenta tomar el bloqueo de objetos de s1 y llama al método test1, pero ya lo adquirió t1, por lo que debe esperar hasta que t1 libere el bloqueo. t2 tampoco liberará el bloqueo de s2 hasta que consiga el bloqueo de s1.

6.- Ahora, ambos hilos están en estado de espera, esperando que los demás liberen bloqueos. Ahora hay una carrera en torno a la condición de quién liberará el candado primero.

7.- Como ninguno de ellos está listo para liberar el bloqueo, esta es la condición de Bloqueo Muerto.

8.- Cuando ejecute este programa, parecerá que la ejecución está en pausa.

Detecta la condición de bloqueo muerto También podemos detectar interbloqueo ejecutando este programa en cmd. Tenemos que recolectar Thread Dump. El comando para recopilar depende del tipo de sistema operativo. Si estamos usando Windows y Java 8, el comando es jcmd $ PID Thread.print

Podemos obtener PID ejecutando el comando jps. El volcado de subprocesos para el programa anterior está a continuación:

```
5524:
2017-04-21 09:57:39
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.25-b02 mixed mode):

"DestroyJavaVM" #12 prio=5 os_prio=0 tid=0x0000000002690800 nid=0xba8 waiting on condition [0x0
   java.lang.Thread.State: RUNNABLE

"Thread-1" #11 prio=5 os_prio=0 tid=0x0000000018bbf800 nid=0x12bc waiting for monitor entry [0x
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Shared.test1(GFG.java:15)
        - waiting to lock  (a Shared)
        at Shared.test2(GFG.java:29)
        - locked  (a Shared)
        at Thread2.run(GFG.java:68)

"Thread-0" #10 prio=5 os_prio=0 tid=0x0000000018bbc000 nid=0x1d8 waiting for monitor entry [0x0
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Shared.test2(GFG.java:25)
        - waiting to lock  (a Shared)
        at Shared.test1(GFG.java:19)
        - locked  (a Shared)
        at Thread1.run(GFG.java:49)

"Service Thread" #9 daemon prio=9 os_prio=0 tid=0x000000001737d800 nid=0x1680 runnable [0x00000
   java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #8 daemon prio=9 os_prio=2 tid=0x000000001732b800 nid=0x17b0 waiting on co
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #7 daemon prio=9 os_prio=2 tid=0x0000000017320800 nid=0x7b4 waiting on con
   java.lang.Thread.State: RUNNABLE
```

```
"C2 CompilerThread0" #6 daemon prio=9 os_prio=2 tid=0x000000001731b000 nid=0x21b0 waiting on c
   java.lang.Thread.State: RUNNABLE

"Attach Listener" #5 daemon prio=5 os_prio=2 tid=0x0000000017319800 nid=0x1294 waiting on cond
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os_prio=2 tid=0x0000000017318000 nid=0x1efc runnable [0x0
   java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os_prio=1 tid=0x0000000002781800 nid=0x5a0 in Object.wait() [0x00
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on  (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        - locked  (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x000000000277a800 nid=0x15b4 in Object.wa
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on  (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Unknown Source)
        at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)
        - locked  (a java.lang.ref.Reference$Lock)

"VM Thread" os_prio=2 tid=0x00000000172e6000 nid=0x1fec runnable

"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x00000000026a6000 nid=0x21fc runnable

"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x00000000026a7800 nid=0x2110 runnable
```

```
"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x00000000026a9000 nid=0xc54 runnable

"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x00000000026ab800 nid=0x704 runnable

"VM Periodic Task Thread" os_prio=2 tid=0x0000000018ba0800 nid=0x610 waiting on condition

JNI global references: 6


Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x0000000018bc1e88 (object 0x00000000d5d645a0, a Shared),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x0000000002780e88 (object 0x00000000d5d645b0, a Shared),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at Shared.test1(GFG.java:15)
        - waiting to lock  (a Shared)
        at Shared.test2(GFG.java:29)
        - locked  (a Shared)
        at Thread2.run(GFG.java:68)
"Thread-0":
        at Shared.test2(GFG.java:25)
        - waiting to lock  (a Shared)
        at Shared.test1(GFG.java:19)
        - locked  (a Shared)
        at Thread1.run(GFG.java:49)
```

```
Found 1 deadlock.
```

Como podemos ver, se menciona claramente que encontró 1 punto muerto. Es posible que aparezca el mismo mensaje cuando lo pruebe en su máquina.

Evitar la condición de bloqueo muerto Podemos evitar la condición de bloqueo muerto al conocer sus posibilidades. Es un proceso muy complejo y no es fácil de atrapar. Pero aun así, si lo intentamos, podemos evitar esto. Hay algunos métodos por los cuales podemos evitar esta condición. No podemos eliminar completamente su posibilidad, pero podemos reducirla.

- Evite los bloqueos anidados: esta es la razón principal del bloqueo muerto. Dead Lock ocurre principalmente cuando damos bloqueos a múltiples hilos. Evite dar bloqueo a múltiples hilos si ya se lo hemos dado a uno.

- Evite bloqueos innecesarios: deberíamos haber bloqueado solo los miembros que se requieren. Tener el bloqueo innecesariamente puede conducir a un bloqueo muerto.

- Usar unión de hilo: la condición de bloqueo muerto aparece cuando un hilo espera otro para finalizar. Si se produce esta condición, podemos usar Thread.join con el tiempo máximo que cree que tomará la ejecución.

Puntos importantes:

- Si los hilos están esperando el uno al otro para terminar, entonces la condición se conoce como interbloqueo.

- La condición de punto muerto es una condición compleja que ocurre solo en caso de múltiples hilos.

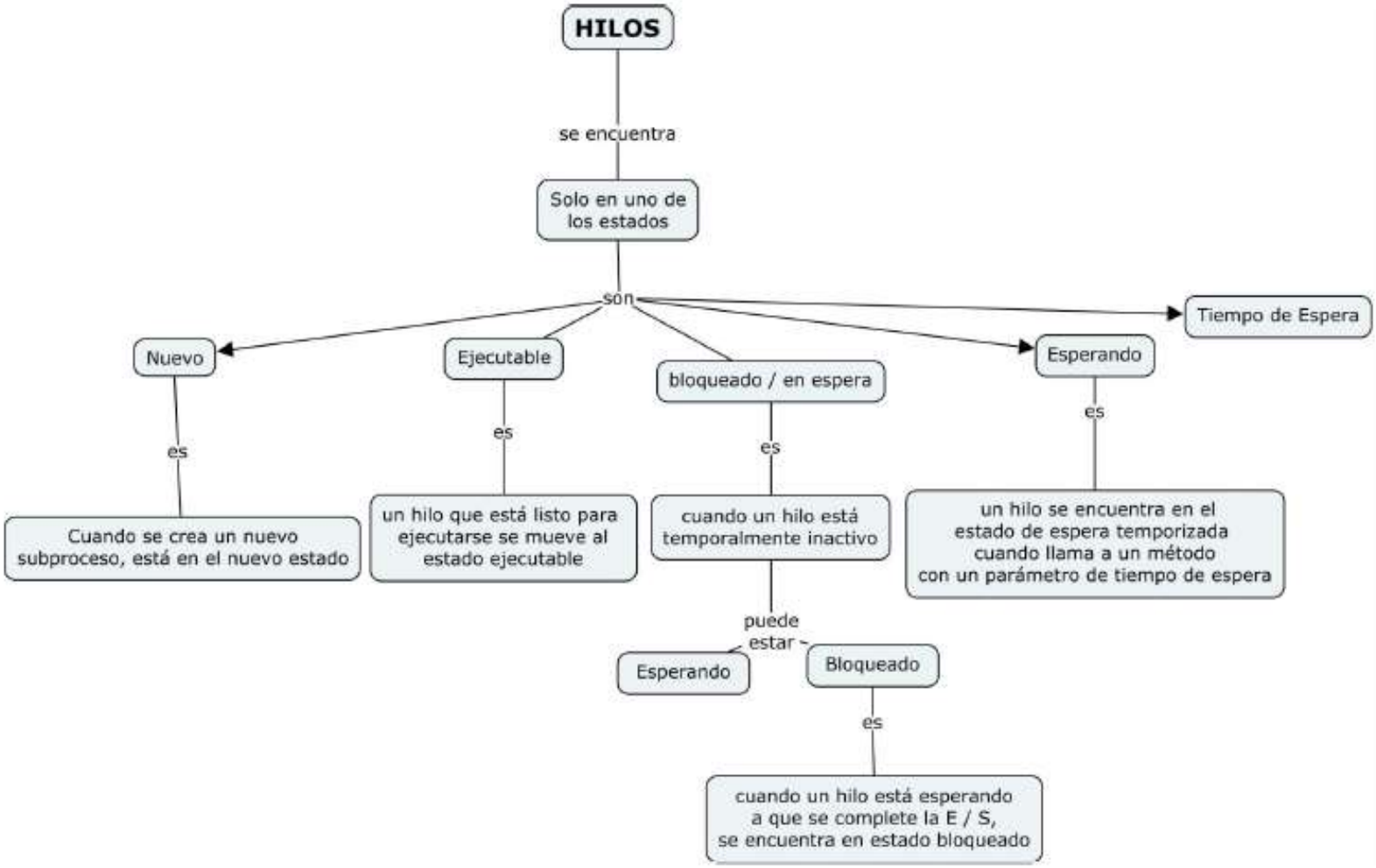
- La condición de interbloqueo puede romper nuestro código en tiempo de ejecución y puede destruir la lógica comercial.

- Debemos evitar esta condición tanto como podamos.

Este artículo es contribuido por Vishal Garg. Si te gusta GeeksforGeeks y te gustaría contribuir, también puedes escribir un artículo usando contribute.geeksforgeeks.org o enviar tu artículo por correo electrónico a contribute@geeksforgeeks.org. Vea su artículo que aparece en la página principal de GeeksforGeeks y ayude a otros Geeks.

Escriba comentarios si encuentra algo incorrecto o si desea compartir más información sobre el tema discutido anteriormente.

Mapa Conceptual



Referencias:

https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-java/

https://www.geeksforgeeks.org/deadlock-in-java-multithreading/